TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Rechnertechnik und Rechnerorganisation

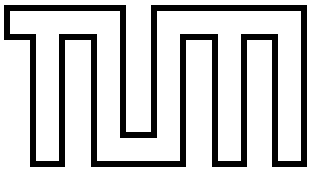
Automated Design of Computer Clusters

Konstantin S. Solnushkin

Vollständiger Abdruck der von der Fakultät für **Informatik**
der Technischen Universität München zur Erlangung des akademischen Grades eines
Doktors **der Naturwissenschaften**
genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. Martin Bichler
Prüfer der Dissertation:	Univ.-Prof. Dr. Arndt Bode
	Univ.-Prof. Dr. Dieter Kranzlmüller,
	Ludwig-Maximilians-Universität München

Die Dissertation wurde am 17.09.2014 bei der Technischen Universität München
eingereicht und durch die Fakultät für Informatik am 16.12.2014 angenommen.



FAKULTÄT FÜR INFORMATIK

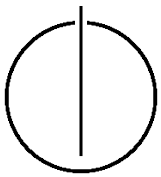
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Dissertation

Automated Design of Computer Clusters

Automatisierter Entwurf von Rechnerclustern

Author:	Konstantin S. Solnushkin
Supervisor:	Univ.-Prof. Dr. Arndt Bode
Advisor:	Univ.-Prof. Dr. Dieter Kranzlmüller (LMU München)
Date:	September 17, 2014



Ich versichere, dass ich diese Dissertation selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 17. September 2014 Konstantin S. Solnushkin

Acknowledgements

Ever since my childhood, I wanted to become a scientist. On my way to this ambitious goal, I met a lot of wonderful people who provided me with help and guidance. In this section, I try to give them a due credit.

It all started in 2005 when I became a PhD student at the Saint Petersburg State Polytechnic University. Professor Valery D. Masin, who supervised my MSc thesis, agreed to become my PhD advisor. By that time, I have already worked as a system administrator at the university's computing centre for a couple of years. We were actively procuring computer clusters but had difficulties determining an optimal configuration of these machines to suit our tasks. Therefore, a topic of the dissertation emerged naturally: designing cost-efficient computer clusters.

I started to work tirelessly, presenting at conferences and publishing in journals. My colleagues at the computing centre gave me enough time to pursue science. At my department, I participated in PhD seminars, receiving useful feedback from fellow students and professors. I also drew a lot from lectures on the philosophy of science and scientific discovery, delivered by wise professors: Konstantin K. Gomoyunov, Alexander A. Krauze, and Vadim G. Knorring.

But despite all the hard work, the progress was not very visible. My grandmother always met me with the question, when will I produce the first page of my thesis? I evaded a direct answer, quoting the number of papers and conference proceedings. The figures were impressive, but I was only a bit closer to my goal than in the beginning, because I apparently tried to "embrace the unembraceable".

Cruel fate dealt me a severe blow when, in the beginning of 2008, two and a half years into my PhD term, my grandmother passed away. I was condemning myself, since I was unable to produce even the first page of the thesis that she so frequently had asked me about. My research work only continued through inertia. I tried to find solace in my everyday job. The prospects were becoming grim, or so it appeared.

It was only in the December of 2009 when I saw the light at the end of the tunnel. Professor Vyacheslav P. Shkodyrev invited me to participate in the "School on Granular Computing" in Erice, Italy, organised by Prof. Dr. Bruno Apolloni and his younger colleague, Dr. Simone Bassis, both from the University of Milan. This event was an eye-opener, a week of pure joy, excitement and scientific discovery. I met many new people and learned many new things. Quotation of Prof. Apolloni has become an epigraph to Chapter 1, "Introduction". An amazing class on interval computation by Prof. Dr. Vladik Kreinovich from the University of Texas at El Paso inspired me to use the methods of interval arithmetic in this thesis: see section 6.4.

The School breathed a new life into me. I saw how international – and European – science operates, and wanted to be a part of it. I revived my research work, and in April of 2011 I visited a PRACE conference in Helsinki, Finland. There I met Dr. Markus Rampp from Rechenzentrum Garching and shared my concerns about completing my PhD in Europe. Markus proposed that I ask Prof. Dr. Arndt Bode to become my PhD advisor. That conversation became another turning point in my life.

Luckily, upon seeing my existing work, Prof. Bode agreed to become my advisor for the next three long years. I promised to complete the PhD as soon as possible, perhaps in a few months, but many times I found new branches worth exploring. I am deeply grateful to Prof. Bode for not urging me with my work and being patient and considerate.

Presenting at international conferences, I met many colleagues, young and old, and these acquaintances helped me see new directions in my scientific work and get in touch with knowledgeable senior colleagues. I am thankful to Dr. José Duato who provided insightful comments on the fat-tree design algorithm in Chapter 13.

Prof. Dr. Dieter Kranzlmüller from Ludwig-Maximilians-Universität München became a thesis co-advisor. Finally, it was time to complete the activities and submit the thesis. Here, invaluable help was provided by Mrs. Manuela Fischer, the secretary of the dean of the Informatics faculty. She supported me in all administrative matters, and we exchanged over one hundred e-mails since my enrolment in the PhD program.

Last but not least, I want to mention my parents and relatives – especially my mother and grandmother – who believed in what I was doing during all these years. At times, I lost hope, but my whole family supported me, both psychologically and financially, no matter what. I dedicate this thesis to the memory of my grandmother; I wish she could be here and share my success with me.

Abstract

Computer clusters made of mass-produced, off-the-shelf components are able to deliver the same real-life performance on a variety of tasks as other types of supercomputers, but at a lower cost. Current ad hoc design practices are characterised by only partial exploration of design space and inability to accurately predict capital and operating expenses. We provide the framework for a more comprehensive design space exploration, paving the way to a future CAD system for computer clusters and warehouse-scale computers with decision support and "what if" analysis capabilities.

We formulate the design task as a discrete combinatorial optimisation problem, with the non-linear objective function being the ratio of total cost of ownership to real-life performance. Although complex, the objective function allows for an unbiased assessment of proposed design alternatives. Various constraints can be imposed on technical and economic characteristics of the computer cluster, including minimal performance, maximal capital and operating expenses, power consumption, occupied space, etc.

Obtaining the value of the objective function is only possible after several consecutive stages of design process are completed: choosing an optimal configuration of a single compute node, designing interconnection network and a power supply system, etc. We provide a prototype CAD tool that implements these stages. We also propose heuristics to deal with combinatorial explosion at various stages.

Zusammenfassung

Automatisierter Entwurf von Rechnerclustern

Rechnercluster aus Standardkomponenten können für eine Vielzahl von Anwendungen vergleichbare Rechenleistung bieten wie spezielle Supercomputer, jedoch zu einem besseren Preis-Leistungsverhältnis. Heutige Adhoc-Entwürfe sind charakterisierbar durch beschränkte Auswertung des Entwurfsraums und die Unfähigkeit, Investitionskosten und Betriebskosten korrekt vorherzusagen. Im Rahmen dieser Arbeit wird eine Methode zur besseren Auswertung des Entwurfsraums vorgeschlagen, die die Basis für künftige automatisierte Entwurfssysteme für Rechnercluster auf Basis von Standardkomponenten darstellen können und entsprechende Entscheidungsunterstützungssysteme beinhalten.

Die Aufgabe des Entwurfs wird dabei als diskretes kombinatorisches Optimierungsproblem formuliert, wobei die nichtlineare Zielfunktion das Verhältnis zwischen Gesamtkosten und Rechenleistung für reale Anwendungen beschreibt. Trotz hoher Komplexität erlaubt die Zielfunktion eine klare Analyse möglicher Entwurfsalternativen. Verschiedenste Randbedingungen für technische und wirtschaftliche Eigenschaften des Rechenclusters können formuliert werden, wie z.B. minimale Rechenleistung, maximale Herstellungskosten und Betriebskosten, Leistungsaufnahme, Raumanforderungen, usw.

Der Wert der Zielfunktion kann erst bestimmt werden, wenn mehrere konsekutive Schritte des Entwurfsprozesses fertiggestellt sind: Auswahl der optionalen Konfiguration des einzelnen Rechenknotens, Entwurf des Verbindungsnetzwerks, der Stromversorgung usw. Im Rahmen der Arbeit wird der Prototyp eines automatisierten Entwurfswerkzeuges realisiert, der diese Arbeitsschritte implementiert. Weiterhin werden Heuristiken entwickelt, die mit der kombinatorischen Explosion die verschiedenen Schritte eingrenzt.

Contents

Part I.

Introduction and Problem Statement

1. Introduction

> To those who are called upon to make decisions, practically the whole of mankind, politicians included. Faced with necessarily granular information, we don't expect people to arrive at the optimum decision. But we demand that they make reasonable choices.
>
> *Prof. Dr. Bruno Apolloni*
> University of Milan

In this chapter we introduce the reader to the task of synthesis of cluster supercomputers. Design procedures result in good, and under certain conditions even mathematically optimal solutions. Extended discussion of optimality is available in section 11.4.

We also explain why the design process should be automated, and show the connection between our task and the field of Electronic Design Automation (EDA).

1.1. Motivation

Computer clusters made of mass-produced, off-the-shelf components have been successful since their emergence in the early 1990s. Having continuously earned high rankings in the TOP500 list [81], they are able to deliver the same real-life performance on a variety of tasks as other types of supercomputers — but at a lower cost.

The off-the-shelf components that serve as the basis for computer clusters are available in great assortment. This means that a basic building block – a compute node – can have many different configurations. Every component, like Central Processing Unit (CPU), memory, local storage device or hardware accelerator, can be present in a compute node in different types and quantities. Even if two nodes have similar sets of inside components, they still can vary greatly in their mechanical characteristics, such as size and weight, which depends on packaging: rack-mounted versus blade servers.

For every distinct compute node configuration, building a compute cluster out of these nodes requires several stages. The first is choosing the number of compute nodes: it should be large enough to satisfy performance constraints, but still should not violate budget constraints. Then storage is added, and a network is designed to connect compute nodes together and to the storage. Finally, equipment is placed into racks, and cables are routed. Every of this stages can be implemented in several ways, which essentially yields a combinatorial optimisation problem. Additionally, a solution to this problem is subject to many constraints, e.g., physical size and power consumption of the future supercomputer are often limited.

Therefore, a challenge exists to build an optimal supercomputer – the one that brings optimality to a certain criterion function while satisfying a set of design constraints.

Supercomputer vendors must be able to meet that challenge. The ability to accurately predict characteristics of a supercomputer is especially important for the bigger systems, when only a small prototype system can be built and evaluated before the bidding procedure with a prospective customer. Additionally, the biggest supercomputers are often based on novel ideas. In this case, many unconventional solutions can be proposed by designers, and evaluating their characteristics should be done automatically.

Currently government organisations are the primary consumers of the largest supercomputers. In the USA, supercomputers sponsored by the National Science Foundation should be procured according to the guidelines outlined in [33]. According to the document, requirements specification for the supercomputer issued by the procuring body should specify the minimum performance that a system has to achieve on a set of benchmarks.

The vendor, in turn, proposes a system that fits within performance, reliability and budgetary constraints. The figures should be obtained from prototype systems or be "estimated by well-justified extrapolation from analogous systems", as the guidelines term it.

The European entity, PRACE, summarised its procurement strategies in [76], citing similar requirements for the bidding process.

A vendor's ability to optimise hardware structure is crucial for winning a contract. For example, a poor initial choice of a CPU which is too expensive can lead to the necessity to choose a low-budget interconnection network, yielding a low performance of the whole system. Choosing a less expensive CPU leaves more funds for the network, and the resulting system may have a higher performance. Such decisions can only be made if designs can be quickly evaluated and compared, which calls for automation.

For scientists, who are today's prevalent users of supercomputers, the existence of such framework means better systems within the same budget.

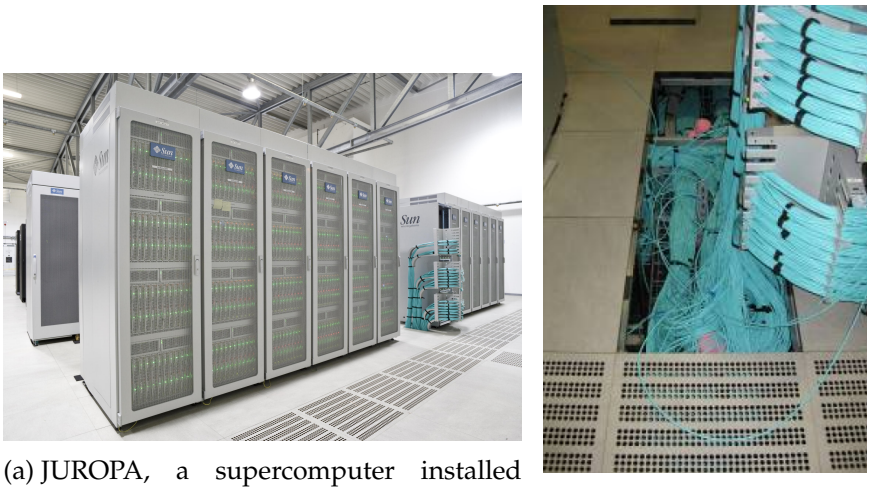
1.2. Benefits of Automated Design Space Exploration

There is a complex interdependence between many factors that influence the design process. A set of components that looked promising in the beginning may eventually result in a design with unacceptable characteristics.

Given the wide assortment of off-the-shelf components, a single compute node can have tens of favourable configurations, and further design stages quickly add to thousands of combinations, each representing a different design, characterised by a number of technical figures (performance, power consumption, size, weight, etc.) and economic figures (procurement costs, total cost of ownership).

The amount of designs that need to be analysed is big, and major technical and economic characteristics for each design must be predicted using complex mathematical models. Existing vendor software tools [38, 44] do not provide required functionality. They check compatibility of user-supplied components and can calculate cost, size and power consumption of a design (but don't try to predict performance), and are unable to automatically iterate through many possible combinations.

Hence, these tools solve the direct problem: given a hardware configuration, assess its



(a) JUROPA, a supercomputer installed in the Jülich Supercomputing Centre. (Source: fz-juelich.de)

(b) Underfloor cables of JUROPA. (Source: fci.com)

Figure 1.1.: Cabling plans are essential for large installations

technical and economic characteristics. We need, however, to solve an inverse problem: given a set of constraints, determine a hardware configuration that is optimal in a certain sense. It is evident that the solution of the inverse problems involves solving a direct problem many times.

Therefore, a framework for automated design of computer clusters is required. In the absence of such a framework, ad hoc design practices are utilised. They are characterised by only partial exploration of design space and inability to accurately predict capital and operating expenses.

For example, a human designer may restrict one's attention to a specific part of design space due to reasons such as bias, personal preferences or simply lack of time.

Employing a Computer-Aided Design (CAD) system to automatically solve a design problem has many benefits. First, a complete exploration of the design space can be ensured, which eventually leads to better designs. The CAD system can evaluate architectures that a human designer would never consider due to one's stereotypes; can do it in less time, with less paperwork and without mistakes.

Second, all infrastructure components – storage, power, cooling – will be automatically taken care of. Tedious task of selecting a proper power supply and cooling systems for every candidate design can be commissioned to a CAD system. Choosing a different CPU model for a compute node can lead to a change in power consumption and therefore required cooling capacity for the entire system in the range of tens of percent. A CAD system will track these changes automatically.

Third, automated design leads to the capability of precise documentation being automatically generated. Not only will a resultant solution meet all design constraints, it will also have its network layout and floor plan available. All power and network cables, as well as cooling water pipes, can be automatically routed in a non-conflicting manner. Figure 1.1 illustrates benefits of cabling plans.

Fourth, automation is the only possible way to track a rapidly changing market situa-

tion. New components frequently appear on the supercomputing market. It also means that existing components become outdated, and their prices decrease. However, the decrease is temporary: when components are out of stock, they become hard to find, and in this situation of low supply the price increases according to market laws. Automation allows to choose an optimal hardware configuration which has the lowest price at a particular moment of time.

Finally, automation makes it possible, in principle, to estimate many auxiliary useful metrics, such as labour expenditures during the system installation.

1.3. A System-Level Complement to EDA

Engineers that design new microprocessors are always challenged with constraints and trade-offs. They can put more cache memory onto a die, but this will increase die size and production costs and decrease yield. They can make cores with a simple instruction set architecture and connect them with a simpler on-chip network, but this may have performance implications. Inflating clock frequency causes heat dissipation problems.

Designers of servers face similar challenges, although on a higher level. Instead of die surface constraints, they have size constraints of a printed circuit board (PCB). Traces on a PCB have to be carefully routed to reduce electromagnetic interference. Heat rejection issues stay important as well.

In this sense, design of supercomputers is a system-level complement to the above mentioned problems of electronic design automation. Even more interesting is that EDA approaches can be utilised to solve problems of cluster computer design. For example, partitioning components between boards in EDA is governed by the same algorithms as partitioning equipment between racks in cluster design. Routing of PCB traces is very similar to routing network cables in clusters. This continuity of ideas and approaches when moving to higher abstraction levels is remarkable. (Note that the placement problem – the problem of placing blocks on a VLSI die or placing components on a PCB – also has a correspondence in cluster design, in the form of placing racks in a machine room. It has, however, a much simpler formulation due to the fact that racks usually have identical dimensions).

In 2010, HiPEAC, the European initiative on architecture and compilation, described in its roadmap [26] the forthcoming shift of Electronic Design Automation to a new abstraction level, Electronic System Level, with one of key issues being "component-based design, from the basic building blocks up to the complete datacenter". This has much in common with our vision.

It is not far away from now when we will apply the principles of silicon compilers – used to automatically design silicon chips with particular structure and functions – to a whole new field of automated design of *complete machines* with particular structure and functions. We even might need new languages for this, like Verilog and VHDL that are used for silicon chips. The proposed framework moves us one step further to this future.

The quest for exascale systems is changing the landscape of cluster computing in yet another way. While current petascale systems have on order of $O(10^4)$ individual CPU chips, future exascale systems are estimated [24] to have two orders of magnitude more chips. Current cluster computers have to refer to off-the-shelf components to keep costs low: mass-produced CPUs are used because they are cheap, not because they are perfect.

In contrast, in future supercomputers the number of CPU chips will be so big that it would justify their custom development. In view of the approach to system development outlined above – from chips to servers to entire supercomputers – this future mass-production not only poses challenges but also opens new prospects. It can signify the convergence between custom supercomputers and the ones based on off-the-shelf components.

1.4. Generality of Approach

The framework that we develop can be applied to more general problems, like the design of "warehouse-scale" computing facilities – the big data centres, as the ones used for cloud computing. Indeed, the distinctive feature of computer clusters is that they are built from a small variety of types of compute nodes, usually just one or two types, and the notion of performance of a supercomputer is clearly defined.

In contrast, big data centres can utilise a diversity of equipment types, and the concept of "performance" of a data centre is defined differently. Aside from that, problems remain the same and are solved in the same way: assessing technical and economic characteristics of a given hardware configuration, designing interconnection networks, placing equipment into racks and positioning racks on the floor, routing cables and so on.

2. Related Work

> If I have seen further it is by standing on the shoulders of giants.
>
> *Isaac Newton*

The process of designing a cluster supercomputer basically consists of several steps: (1) exploring design space to find favourable configurations, (2) estimating performance of each configuration, (3) adding infrastructure components, such as network, power, cooling, etc., (4) assessing economic and technical characteristics of resulting designs.

The configuration step is of particular importance: using domain-specific knowledge, it combines components in a compatible fashion, and simultaneously filters out poor designs. This significantly reduces the number of designs that have to participate in subsequent procedures.

In this chapter we review a number of works: papers [60, 62, 43] address just the configuration problem in the field of computer design, while papers [22, 112] try to solve the whole problem of choosing an optimal computer cluster. (Literature on other tasks is reviewed in relevant chapters, e.g., literature on networks is discussed in Chapters 13 and 14).

Additionally, article [57] contains an extensive review of different combinatorial mathematics formulations of system configuration task. It shows that a problem of finding a set of distinct representatives, shortest path problem, knapsack problem, assignment problem and morphological analysis – both in one-criterion and multi-criterion formulations – can be successfully used to describe real-world configuration tasks.

2.1. McDermott, 1980

R1 is an expert system created by John P. McDermott in late 1970s [60]. Its main task was to configure VAX-11/780 minicomputers made by Digital Equipment Corporation. As VAX computers supported a large assortment of peripheral devices, the total number of possible configurations was very big.

A bus called Unibus was used to connect low-speed peripheral modules to a minicomputer. There were lots of mechanical and power constraints that directed placement of Unibus modules into backplanes, backplanes into boxes, and boxes into cabinets.

Unibus modules should be put into backplanes in a specific precomputed optimal sequence. If, after placing a module, the remaining space in the backplane is not sufficient to accommodate the next module in the sequence, the designer has to consider two choices: (1) either add a next backplane, possibly increasing the volume taken by hardware, and eventually the occupied floorspace, or (2) deviate from the optimal sequence of modules.

Similar rules and constraints apply to configuring another bus, the Massbus, used to connect disk and tape drives.

Working with human designers, McDermott was able to initially extract 480 rules that represented domain knowledge. Then, *R1*, an expert system, was implemented as a production rule system.

R1 was designed to check customers' orders for validity and complement them if necessary. Given a partially defined order, the system would extend the configuration using the design rules. For example, if one backplane was not enough to accommodate all peripherals specified in the order, the system would add as many backplanes as required.

The system didn't try to iterate through different models of components, assessing cost and performance of resulting designs, simply because it was created to solve a different problem. Therefore, the *R1*'s approach could not be directly applied to design cluster supercomputers. However, integrating an expert system into a CAD tool for cluster supercomputers appears beneficial, and this approach is detailed in chapter 18.

In a certain sense, *R1* set the standards for future automated configurers of computers. Indeed, the system was able to determine spatial location of components in cabinets, position of cabinets on the floor, length of cables (and produced a wiring table), and also reported unused capacity – i.e., what other components could potentially be added to expand a computer in the future.

Additionally, *R1*, as a CAD system, was able to produce more detailed documentation for technicians performing the physical assembly than human designers could do. This feature, implemented in a future CAD system for supercomputers, would be especially useful for larger installations.

2.2. Mittal and Frayman, 1989

In 1989, Sanjay Mittal and Felix Frayman revisited a general configuration task [62]. They pointed at the deficiency of then-current aproaches: namely, the reliance on naïve definition of configuration as a design activity, which didn't allow for comparison between approaches. As a uniformly accepted formal definition seemed to be lacking, Mittal and Frayman introduced it.

According to their general definition, there exists a set of components, and each component has a set of ports to connect to other components. The general version of the configuration task – any component can be connected to any other compatible one – has an exponential complexity.

They further introduce two restriction. The first relates to *"functional architecture"*: instead of trying all possible arbitrary combinations of compatible components, real-world configuration tasks usually connect components according to a certain set of rules which together define an *architecture*, intended to fulfil a specific *function*. All other combinations, although valid, should not be considered, which reduces search.

The second restriction concerns "key components": if a certain functionality must be available, there is usually a key component that is crucial in providing that functionality, and which also entails a set of auxiliary components.

Example 2.1 *Cluster supercomputers are commonly built using commercially available servers. Each server has a video display interface which can be used for local debugging. However, the*

architecture *of cluster supercomputers does not assume that displays would be connected to every cluster node in the resulting machine: it is a computing farm, not a visualisation wall. Hence, although the connection is formally valid, it contradicts the architecture and shall not be considered during the automated configuration.*

Similarly, to implement a function "interconnection network", several key components are required: network adapters in cluster nodes, network switches and network cables. Network switches additionally entail an auxiliary sub-component: power cables.

Together, these two restrictions, based on domain-specific knowledge, not only reduce search space, but also help to partition the global configuration task into a number of relatively independent subtasks.

Although the above considerations of knowledge representation may seem obvious for any particular domain, the merit of the cited work is in formalising the ideas for the most general abstract case. We will often refer to the concepts outlined above throughout this thesis.

The authors also proposed an algorithm to search for configurations. They advised to check compatibility constraints as early as possible during the design process to reduce the need to backtrack.

2.3. Hsiung et al., 1998

ICOS – an Intelligent Concurrent Object-Oriented Synthesis methodology – was proposed in 1998 by Pao-Ann Hsiung *et al.* [43] and focuses on design of multiprocessor systems. In this object-oriented methodology, system components are modelled as classes with hierarchical relationships between them.

Previously synthesised subsystems are reused as building blocks of new designs. To achieve this, *ICOS* applies machine learning techniques. It compares specifications of previously learnt designs with current specifications, using fuzzy logic, and in case of a match, a previous design is reused.

Several design alternatives can be evaluated concurrently (in parallel), with the aim of further reducing time to solution. Performance, cost, scalability, reliability and fault-tolerance constraints can all be specified. Logic rules are used to detect incorrect or contradictory input specifications.

Unsuccessfully synthesised components (violating constraints) cause the rollback procedure by propagating messages in the class hierarchy and subsequent re-synthesis. Of several system designs that match constraints, the one with the best performance is selected.

Overall, *ICOS* provides a way to represent design data and candidate solutions in the memory of a CAD tool, in the form of a class hierarchy. It is theoretically capable of designing cluster supercomputers, which can be specified therein as "MIMD hybrid architectures", i.e., a number of multiprocessor compute nodes, each with a shared memory, connected together by an interconnection network.

However, *ICOS*, being a rather general tool, cannot evaluate performance on different workloads, doesn't take infrastructural component of cluster supercomputers (power, cooling, etc.) into account and is unable to calculate the total cost of ownership (TCO). The

approach that we propose in this thesis overcomes these deficiencies, while utilising certain ideas from *ICOS* – namely, representing components as objects and ability to specify an array of practically meaningful constraints.

2.4. Dieter and Dietz, 2005

At Supercomputing 2002 conference, William R. Dieter and Henry G. Dietz presented the tutorial on *Cluster Design Rules (CDR)*, the methodology to design computer clusters suited for specific workloads. By 2005, their web-based CAD tool, *CDR*, was used many times, and certain patterns started to emerge in the tool's output. The findings were summarised in 2005 in the technical report [22].

CDR is perhaps the first documented attempt to create a CAD tool for clusters, specific enough to take care of necessary details, such as infrastructural equipment and component prices, and at the same time general enough to allow for a wide variety of components.

Dieter and Dietz found an important regularity: for the criterion functions they were using, there were no simple ways to derive a globally-optimal model of a component from its parameters. For example, the fastest (and hence the most expensive) CPU as well as the cheapest (and hence the slowest one) did not necessarily deliver optimality to the resultant design. Similarly, the CPU with the lowest price to performance ratio did not yield optimality, too.

It means that global optimality of the entire system cannot be ensured via using locally optimal components. In this example, the correct choice of a CPU can only be made through exhaustive search, trying every candidate CPU and evaluating the global criterion function. Additionally, when market prices change, the design procedure has to start anew, because now it could be a different model that would bring global optimality.

During the execution, the tool presents a designer with a series of questions that mainly describe workload characteristics. Of particular importance are (1) main memory bandwidth within a compute node, measured in GBytes/second per a GFLOPS, that an application needs, and (2) network parameters – namely, latencies of ordinary and collective operations, bisection bandwidth per a processor core, and the number of neighbouring compute nodes that an application will typically communicate with.

Unfortunately, there are problems with these parameters. First, they are difficult to quantify (although a link is given to the paper that suggests to determine them using hardware counters). Second, application's needs may change during different phases of its execution. It is unclear whether we should engineer the cluster for the worst case, or for the average case, or otherwise. Third, in current cases of shared usage of big supercomputers, applications come and go, and requirements of the same application change with code updates, introduction of new mathematical models, or with different input data. In this case, precise tracking of the above parameters becomes useless.

It makes sense to determine workload requirements when workload is rather stable, such as day-to-day weather predictions in national weather forecasting facilities. In other cases, instead of the fairly low-level parameters, performance modelling appears a viable alternative. *CDR* allows to specify the minimum required performance of the supercomputer on High Performance Linpack and SWEEP3D benchmarks (the latter is a particle transport simulation).

Additionally, physical constraints (rack space and available cooling) and budgetary constraints (acquisition cost, operating costs: electricity and floor space rent) can be specified.

The criterion function is a weighted linear combination of metrics, sought to be maximised. (Another criterion function, aimed to optimise for the minimal total cost of ownership (TCO), is marked as "experimental" and does not work). Weights can be specified as zero (to disregard values of certain metrics) or as 10^n, where $0 \leq n \leq 6$. Metrics are either ordinary (memory space size, memory bandwidth, disk space size, network bandwidth and raw performance) or inverse (the lower the value, the better: network latency, acquisition cost, operating cost per year).

The difficulty with the weightings is their voluntaristic assignment by the user. There is no "ideal" assignment, so different assignments result in different "optimal" designs. This depreciates the whole idea of optimisation.

CDR builds network structures for the supercomputers it designs, using the auxiliary *NetWires* tool [23]. *NetWires* is capable of designing and visualising network topologies; however, it is unable to calculate network cost, or design networks according to constraints (such as expandability).

The results in [22] indicated the complex nature of design space when pricing is taken into account. For example, when setting varying constraints for network performance and cost, *CDR* suggested radically different network solutions, in terms of hardware and topologies. Similarly, for smaller clusters, uniprocessor compute nodes were found to have a higher performance within the same budget than the multiprocessor ones – a non-intuitive conclusion. These examples once again underline that a thorough automated search is required.

Overall, *CDR* was a successful project that pointed to new directions for research in its field.

2.5. Venkateswaran et al., 2009

The problem of automated design of cluster supercomputers was attacked again in 2009 by Nagarajan Venkateswaran et al. [112]. Their methodology, *"Modeling and integrated design automation of supercomputers (MIDAS)"*, tried to analyse and model cluster supercomputers via the use of simulation. Although the methodology is aimed at Supercomputers-on-a-Chip (SCOC), it can be generalised to wider areas as well.

MIDAS builds a dependency graph of many parameters that determine performance and power consumption of a cluster supercomputer. Then the relationships are expressed as analytical mathematical models, in form of equations. Separate components of a computer are optimised using a simulated annealing procedure, to parametrise each model. Then, high-level characteristics, such as performance, power, performance per watt are inferred from separate models.

As *MIDAS* is intended for Supercomputers-on-a-Chip, it's authors suggested to build the chips – the basis of cluster nodes – using a library of IP cores. The functionality of cores ranges from numerical to linear algebra to graph algorithms. Placing cores on a chip in necessary quantities delivers specific functionality and performance of a cluster node. Designing these custom chips (ASICs) is done using familiar EDA tools.

MIDAS does not address problems of building interconnection networks to connect cluster nodes, nor it concerns infrastructural components or equipment placement problems. However, it serves as an important link for implementing the Electronic System Level design workflow detailed in “Introduction”: from chips to servers to supercomputers.

3. Problem Statement

> Dr. Hoenikker used to say that any scientist who couldn't explain to an eight-year-old what he was doing was a charlatan.
>
> *Kurt Vonnegut*
> CAT'S CRADLE, CHAPTER 15

This chapter presents a concise formulation of the problem that we are trying to solve, and introduces the reader to the possible mathematical formalisms that could be employed for the task.

3.1. Problem Statement

We formulate the problem as follows: build a cluster supercomputer from identical compute nodes, connected together via a network, equipped with uninterruptible power supply system (and possibly other infrastructural systems), subject to constraints imposed on minimum performance and maximum acquisition cost, total cost of ownership (TCO), space, power consumption (and possibly other characteristics), and yielding minimality to the criterion function: $f = TCO/Performance$. The criterion function is therefore the simplest multiplicative function, but its non-linearity induces certain consequences. This choice is justified in Chapter 6.

We call for the use of total cost of ownership instead of using just capital expenditures (the procurement costs), because operating expenditures can comprise a substantial share of the TCO. For example, water-cooled equipment, seemingly expensive in terms of up-front costs, may have lower operating expenditures, whilst allowing easy recuperation of waste heat which leads to further savings.

The problem is a discrete optimisation problem. To perform a thorough search, we need to: (1) try every possible compute node configuration, (2) choose the number of compute nodes that meets constraints; if there are multiple variants, try all of them, (3) connect compute nodes with all possible networks, (4) design infrastructural systems, etc. Of course, this formulation makes the task intractable. We need to introduce substantial simplifications, and we start with a definition of a *configuration*.

3.2. Defining a Configuration

Following Mittal and Frayman [62], if we have a set of components, and every component has ports to connect it to other compatible components, then a configuration is defined by exact instances of components and structure of their connections with each other.

This "material" definition is easily transferred into the field of cluster supercomputers. Indeed, a compute node has a number of distinct ports, designed specifically to connect it to other components of a supercomputer. Certain network ports can be used to connect to a high-speed interconnection network, while others can serve to build a storage network. Power ports connect compute nodes to relevant power equipment, such as uninterruptible power supplies (UPS).

Some "ports" are less material but still stipulate compatibility between components. For example, mechanical compatibility between rack-mounted servers and racks is ensured by following industry standards. On the contrary, blade servers are normally compatible only with enclosures made by the same vendor. In a CAD tool, two models of blade servers made by different vendors will have to be represented as having incompatible "ports".

But a configuration is not described merely with material items. Let us consider an example. Multiprocessor compute nodes often have a system setting called *"Memory interleave"*, which alters the assignment of memory blocks to CPUs. By turning this setting on or off, hardware structure can be tuned to a particular memory access pattern of applications. For any given algorithm, both values of this setting can result in either performance increase or degradation. Choosing inappropriate value can result in a performance loss of this particular node. In parallel computing, where overall performance is often determined by the slowest node, this can lead to slowing down the entire cluster computer. For such a non-material configuration item, there are no "ports" to connect it to other components, still we need a way to represent this item in a CAD system. The key question is therefore knowledge representation.

3.3. Representing Configurations with Multipartite Graphs

For a start, we must be able to construct valid configurations of compute nodes. In layman terms, building a compute node from components boils down to filling provided "sockets" and "slots" with these components, according to certain rules.

We follow the approach of Bozhko and Tolparov [14] for representing configurations of arbitrary technical systems using multipartite graphs. However, instead of undirected graphs with cycles we propose to use directed acyclic graphs, and rationalise our choice in Chapter 10.

A relatively simple example of representing a compute node configuration with a multipartite graph is shown in Figure 3.1 (the graph is directed, but arrows are not shown to reduce clutter). Each partition of the graph contains components of the same type. Edges between components in different partitions represent compatibility of those components (components are considered compatible if there are no restrictions on their simultaneous use in the technical system). Every path in the graph represents a valid configuration.

This fictional compute node can have either one or two CPUs. Three CPU models are available, but if two CPUs are used, the models must be identical. Additionally, each CPU has a set of memory slots associated with it (a NUMA architecture). Note that we do not represent memory slots in this particular graph, as this would further complicate matters. Instead, we specify three available sizes of memory that can be attached to a single CPU.

There are two rules according to which this compute node must be configured: (1) at least one CPU must be installed, (2) memory slots belonging to a certain CPU socket can

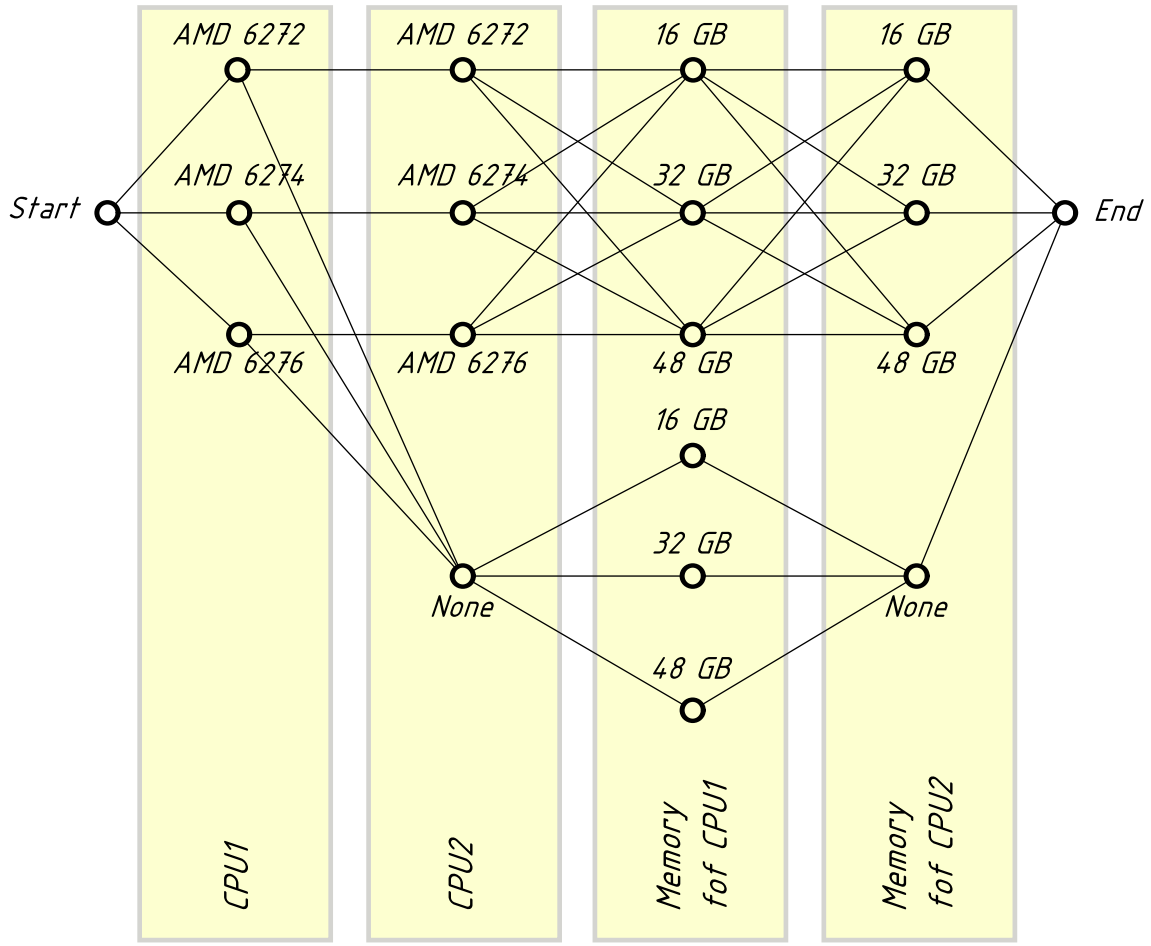


Figure 3.1.: Representing configuration of a fictional compute node with a multipartite graph.

only be filled if the corresponding CPU is installed.

Traversal of this graph models possible scenarios of configuring a compute node from components. The process begins with filling the first CPU socket with one of three possible CPU models. Then, if no second CPU is used, we only need to choose one of three possible memory sizes for CPU1. Hence, a uniprocessor configuration has 9 realisations. If we decide, however, to install a second CPU, then we can choose one of three possible memory sizes for CPU1 and one of three sizes for CPU2. Therefore, in a dual-processor node, for any given CPU model, there are nine possible memory configurations, and combined with three CPU models, this gives 27 realisations. Together, this fictional node can be configured in $9 + 27 = 36$ distinct ways.

Even though only some of these 36 configurations are close to optimal, we need to represent all of them in a graph, to avoid invalid configurations. Trying to "simplify" this graph may generate invalid solutions, such as having both CPU sockets empty, or having memory attached to both CPU sockets while only one is filled.

Figure 3.2 describes traversal of the graph. Vertices – components of the compute node –

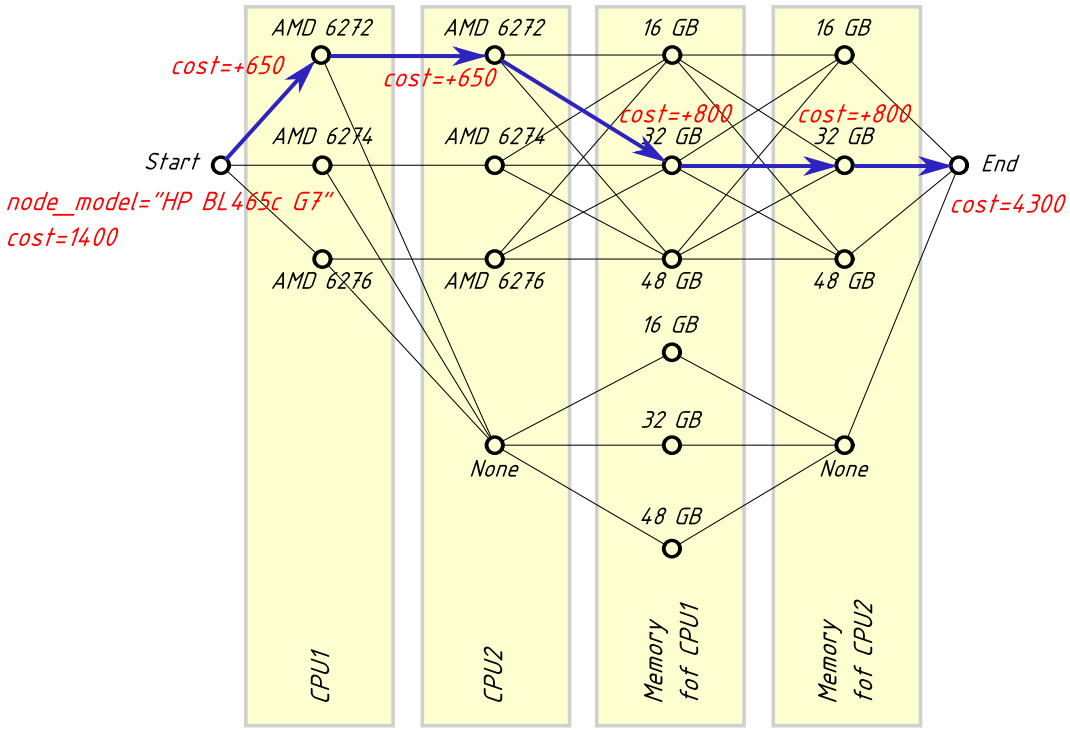


Figure 3.2.: Traversing the graph to calculate characteristics of a configuration.

are assigned with a set of characteristics, literal or numeric, such as `"cost=650"`. Traversing the graph involves finding all possible paths from "Start" to "End". One such path is highlighted with arrows. Traversing vertices along the path allows to accumulate characteristics assigned to vertices or to perform evaluation of expressions. For example, expression `"cost=+800"` signifies adding "800" to the current value of variable `"cost"`. (See complete syntax in Chapter 10). Thereby, after traversing the path, the value of cost is determined and stored for future use.

Real-life equipment can be configured in much more ways. For example, Hewlett-Packards's "SL390s G7" server (the 4U height version) has two CPU sockets, and 13 CPU models are supported. Twelve memory slots are available, providing for numerous memory configurations, with both ordinary and low-power modules. Internal x16 PCIe 2.0 slots allow installation of up to three GPU accelerators. Up to eight hard drives can be installed, combined into RAID arrays of three possible levels – 0, 1 and 5. A server can also have an optional InfiniBand adapter, and last but not least, it is shipped in either left or right variant, depending on its final position in the enclosure. As can be seen, the possibilities for configuration are endless, as is the combinatorial difficulty of the task.

3.4. Algorithm for Designing Computer Clusters

The algorithm is described in full in Chapter 11. The required input to the algorithm is the minimal performance that the cluster computer must be able to achieve. Optional inputs include design constraints (see Chapter 8). The output of the algorithm is the *configuration*

of the cluster computer, with all technical and economic characteristics necessary to (a) conduct procurement, and (b) perform further design procedures as necessary. For each configuration of a compute node obtained via traversing the graph, the following stages of the algorithm are performed:

1. Using inverse performance modelling (see Chapter 9), calculate the number of compute nodes that is required to attain the specified performance.
2. Design an interconnection network (see Chapters 13 and 14).
3. Design infrastructural systems, such as uninterruptible power supply system (see Chapter 15).
4. Place equipment into racks and place racks on the floor plan (see Chapter 16).
5. Compute the criterion function for the configuration (see Chapter 6).

On every stage of this algorithm various technical and economic characteristics of the configuration are evaluated, such as its cost, power consumption, amount of equipment, etc. Simultaneously, on every stage configurations are checked against specified constraints, which allows to detect and discard violating configurations as early as possible.

The characteristic feature of our approach is that we propose to perform separate stages of the design process by querying, via network, appropriate modules – performance modelling module, network design module, and so on. This modular approach ensures extensibility of the CAD tool. It also allows to utilise the most current versions of design algorithms by plugging in corresponding modules.

3.5. Combinatorial Explosion

Separate stages of the design process are subjected to local optimisation performed by the corresponding modules. This allows to put a limit on combinatorial explosion. For example, suppose that for each configuration a network can be designed in 5 ways, and a UPS system in 3 ways. Therefore, for each configuration it will be required to run the network design stage 5 times (5 units of work), and then for each of these semi-complete designs the UPS design stage must be run 3 times (15 units of work), resulting in 20 instances of design modules invocation.

In our approach, we run the network design stage 5 times and choose a locally optimal solution. Then, we run a UPS design stage for 3 times, and again choose a locally optimal solution. This approach may miss some globally optimal designs; however, it allows to keep design times reasonably low.

There are other occasions where combinatorial explosion may manifest itself. For example, turning the “Memory interleave” option in the BIOS of a compute node on or off results, in fact, in two separate configurations, with drastically different performance characteristics (see section 9.1.1 for this and other examples). The “Memory interleave” setting *can* be represented with a configuration graph (by adding a new partition to the graph with two vertices); however, it only makes sense to do so if there exist performance models that can adequately predict performance based on this BIOS setting.

Yet another case is choosing a software environment for the computer cluster. If there are several software packages to solve a problem, and each is characterised by its own performance curve and total cost of ownership (licensing costs as well as maintenance and support costs), then thorough search among all software alternatives is required. In the current implementation of our CAD tool, we ask the user to manually specify what software (and its performance model) should be used to drive design decisions.

3.6. Interdependencies

Real life computing equipment can present unexpected interdependencies of components. We review several examples of such situations where the configuration graph can adequately represent valid configurations.

1. Modern commodity CPUs have a built-in memory controller which only supports a specific amount of RAM. Therefore, a uniprocessor configuration can only be equipped with a limited amount of memory. If more memory is needed, it is necessary to install a second CPU, and use its designated memory slots. Here the requirement of adding more memory unexpectedly entails the need to install more CPUs. (See, for example, Fig. 3.2, where installing 64 GBytes or more memory inevitably requires installation of both CPUs).

2. Highest-capacity memory modules (such as 32 GByte DDR3 SDRAM modules) often work at lower speeds than modules of lower capacity (16 or 8 GBytes). In high-performance computing, the performance of memory subsystem is of utmost importance. To reflect the issue, the configuration graph can be reworked to specify exact types of memory modules and their performance characteristics.

   Traversing the graph will generate all possible configurations. Then, constraints can be imposed on the minimal total amount of memory and on the minimal memory speed. This will weed out unsuitable configurations with slow memory. (It is also desirable to pass the memory performance characteristics on to the performance modelling module for more accurate performance predictions).

3. Some servers have a large number of memory slots, but when using high-capacity memory modules, only part of the slots can be utilised, because the memory controller built into a lower-end CPU can only address a limited amount of memory. However, when using a higher-end CPU, all slots can be occupied.

4. Some servers can accommodate a wide variety of CPUs, ranging from low-end, low-power models to hi-end models with high power consumption. It is possible that due to thermal constraints either two low-power CPUs can be installed in a server, or only one high-power CPU. The configuration graph can take care of this situation.

In all of these cases, using graphs allows to adequately represent real life interdependencies of components. There are, of course, more than these few examples of interdependencies. Some of those listed above can be obsoleted with introduction of new hardware, while at the same time new interdependencies will emerge.

3.7. Proposed Design Framework as a Means of Collaboration

Procurement of an HPC system involves at least three parties: hardware vendors, application software vendors, and byers; the latter are research institutions and companies. (In certain settings, although not in general, buyers can write their own software to be run on a supercomputer, and therefore also play the role of application software vendors).

These three parties have little in common. Hardware vendors have knowledge of their hardware and peculiarities of its usage. Application software vendors know the properties of their software, and can predict how well it will run on the system being designed and built, and how the software performance will be affected by changing computer parameters.

For example, they can predict what impact a twice faster network could have on the overall performance of their software. At the same time, they might not know complex compatibility relations between hardware components. During the procurement process, both groups – hardware and software vendors – work together to determine an "optimal" configuration of a supercomputer for the task, and benchmarking is the main method employed.

Buyers, constrained by tight deadlines, necessarily face uncertainties. When large supercomputers are procured, it is generally impossible to run a full-scale benchmark, because hardware vendors are unwilling to commit resources and time to building large prototype machines. Similarly, if a decision to install a twice faster network is considered, buyers need cooperation from both hardware and software vendors to make a conclusion.

A faster network will increase procurement and possibly operating costs. It is difficult to quantify the increase unless the hardware vendor is willing to prepare several network designs for the client, or if buyers possess tools to automatically design networks and assess their technical and economic characteristics. On the other hand, a faster network may increase performance of the supercomputer, perhaps surpassing associated cost increase. Again, it is difficult to quantify performance increase unless software vendor cooperates, or if there are tools to predict performance depending on computer parameters.

The proposed framework gives in the hands of buyers, the most numerous group of players in the HPC market, a convenient set of tools which lessens their dependence on hardware and software vendors. Software vendors can turn informal knowledge of their software into performance models. Such models can be built and verified (similarly to how we built a performance model for ANSYS Fluent software suite in section 9.4), and then tweaked to be precise enough to make useful performance predictions.

Hardware vendors then can provide a configuration graph for their hardware, which will generate valid configurations, together with cost, power and other characteristics. Network, UPS and other vendors can provide modules that can design networks (see Chapters 13 and 14), UPS systems (see Chapter 15), etc. The CAD tool will bring all components together, allowing buyers to quickly search their chosen part of the design space, without unnecessary reliance on vendors, and arrive to provably good designs.

4. Outline of the Thesis

Part I: Introduction and Problem Statement

CHAPTER 1: INTRODUCTION

We start the thesis by explaining motivation behind the work, highlighting benefits of automated design space exploration, and clarifying how design of computer clusters complements the field of electronic design automation (EDA).

CHAPTER 2: RELATED WORK

We review related work in the filed of configuration of computer systems: from the general formulation of the task, dating back to 1970s, to multiprocessor systems (1998), to automated design of high-performance computers (2005 and 2009).

CHAPTER 3: PROBLEM STATEMENT

This chapter formulates the design task in terms of a discrete optimisation problem, explains the use of multipartite graphs to represent configurations, outlines the algorithm for automated design of cluster supercomputers, and proposes strategies to deal with combinatorial explosion.

We also provide the overview of complex interdependencies between components that can arise in real life configurations of supercomputers, and the role of configuration graphs in representing compatibility between components.

Finally, we explain how the automated design framework presented in the thesis, together with its software tools, lessens the dependence of supercomputer buyers on hardware and software vendors.

CHAPTER 5: SCIENTIFIC CONTRIBUTION

Here, we list scientific contributions of the thesis, i.e., what the reader can learn from the thesis compared to the state of the art.

Part II: Criterion Function and Design Constraints

CHAPTER 6: CHOICE OF THE CRITERION FUNCTION

In this chapter, we propose the criterion function to be used throughout the thesis: the ratio of total cost of ownership to performance. We explain why using one-criterion instead of muti-criterion optimisation is possible and provides good results. Additionally, a generalisation of the criterion function based on interval arithmetic is introduced.

CHAPTER 7: ECONOMICS OF CLUSTER COMPUTING

We explain the benefits of total cost of ownership compared to procurement cost as a metric guiding design decisions. We also compare capital and operating expenditures. Finally, questions of balance between high-speed, high-cost components and low-power, low operating cost components are addressed.

CHAPTER 8: DEALING WITH COMBINATORIAL EXPLOSION

Here, we detail strategies to deal with combinatorial explosion. The first approach is to impose constraints on characteristics of compute nodes or the whole machine, and the second one is to use heuristics that allow to quickly weed out unpromising solutions.

CHAPTER 9: PERFORMANCE MODELLING

In this chapter, we introduce the notion of performance and speed of computer components, and then review related work concerning factors that influence performance as well as approaches to performance modelling.

We then discuss the "optimal" number of computing blocks (cores, CPUs, compute nodes) to be used for parallel execution, and the corresponding "throughput mode" of operation of large supercomputers.

Further, we introduce inverse performance models and the algorithm to determine the number of compute blocks required to attain a given performance level.

Finally, a simple analytical performance model for "ANSYS Fluent" CAE software suite is proposed, and the process of querying performance models via Internet is explained.

Part III: Automated Design of Computer Clusters

CHAPTER 10: GRAPH REPRESENTATION OF CONFIGURATIONS

We explain the use of directed acyclic graphs for representing compatibility between components of arbitrary technical systems, comparing benefits and disadvantages of directed and undirected graphs.

We then propose assigning arithmetic expressions to vertices and edges of the configuration graphs, which results in evaluation of technical and economic characteristics of systems during graph traversal. Finally, XML syntax for representing graphs is introduced.

CHAPTER 11: ALGORITHM FOR AUTOMATED DESIGN OF CLUSTER SUPERCOMPUTERS

Here, we describe in detail stages of the main algorithm used in the thesis, as well as discuss the limits of the design framework and their influence on the optimality of solutions in the mathematical sense.

We then make the case for designing computer clusters based on their minimum performance rather than on maximum budget.

CHAPTER 12: CAD SYSTEM FOR COMPUTER CLUSTERS

This chapter describes the user interface of the prototype CAD tool for computer clusters. We also discuss currently implemented modules of the CAD system and their invo-

cation sequence, and propose a parallelisation strategy for the CAD tool.

Chapter 13: Fat-tree Network Design

We propose an algorithm for designing two-level fat-tree networks with arbitrary blocking factors, automatically choosing the best combination of monolithic or modular switches on both levels, subject to various constraints.

We then discuss how technical and economic characteristics of fat-tree networks can be easily derived from corresponding per-port metrics. Comparison of strategies for future expansion of fat-trees concludes the chapter.

Note: the contents from this chapter were deposited in Arxiv, an open-access pre-print repository [94].

Chapter 14: Torus Network Design

We propose an algorithm for designing torus networks, and a heuristic for automatically choosing the number of torus dimensions. We then compare cost of torus and fat-tree networks.

Note: the contents from this chapter were deposited in Arxiv, an open-access pre-print repository [93].

Chapter 15: Designing Other Subsystems of Computer Clusters

We describe strategies to design storage and cooling systems, investigating the possibility of using outside air for cooling purposes, depending on climate. We then compare costs of cooling solutions, including water-based cooling, and propose to reuse waste heat for agricultural purposes. Finally, we introduce an algorithm for designing UPS systems.

Chapter 16: Equipment Placement and Floor Planning

This chapter introduces heuristics for placing equipment into racks, and compares approaches of equipment consolidation and distribution. We then provide a simple analytical model for determining floorspace required to host a computer installation.

Chapter 17: Practical Evaluation of the Algorithm

In this chapter, we evaluate the main algorithm of the thesis on a set of real life hardware. We compare technical and economic characteristics of individual configurations of compute nodes as well as clusters built using that configurations, with full infrastructure, such as interconnection networks and UPS systems.

We further provide a detailed analysis of several cluster designs, and quantitatively investigate how changes in technical characteristics of interconnection networks and UPS systems impact economic characteristics of cluster computers.

Chapter 18: Summary and Future Directions

Here, we summarise the contribution of the thesis, and provide guidelines for future directions of work, such as turning the prototype CAD tool into a decision support system.

5. Scientific Contribution

> To be of use to the world is the only way to be happy.
>
> *Hans Christian Andersen*

In this chapter, we list scientific contributions made by the thesis, listed by knowledge areas.

5.1. CAD systems

We propose a method for representing compatibility between components of arbitrary technical systems using directed acyclic multipartite graphs for the purpose of structural synthesis. We make the case for a modular CAD system for cluster supercomputers, with modules supplied and maintained by hardware and software vendors, ensuring the use of the most current price data and most recent algorithms to design separate subsystems. We envision the use of interval arithmetic to capture uncertainty in both cost and performance. We provide a proof that the single-objective optimisation used in the thesis always results in a Pareto-optimal solution. We suggest a heuristic that allows to decrease design space by a factor of ten but does not accidentally reject optimal solutions. We also offer arguments against using local optimisations.

5.2. Performance modelling

We introduce the notion of inverse performance models and propose a two-phase iterative algorithm for inverse performance modelling.

5.3. Computer networks

We propose algorithms to design two-level fat-tree and torus networks, with arbitrary blocking factors. The algorithms operate with real life equipment characteristics such as cost, power consumption, occupied rack space, weight and others, and are able to arrive to cost-efficient network designs by utilising partially populated modular switches.

5.4. Data centre design

We propose strategies and heuristics for placing equipment into racks, for the general case of non-identical equipment blocks, taking into account space, weight and power budget

of individual racks. We devise an algorithm for calculating floor space size required for a given number of racks, with a constant run time. We also offer a greedy algorithm for sizing an uninterruptible power supply system.

5.5. Cooling systems

We provide a chart for choosing air preparation methods for cooling with outside air and suggest an algorithm for calculating cooling capacity for cooling with outside air. We then perform comparison of capital and operating costs of three types of cooling solutions.

5.6. Economics

We provide a comparison of factors that influence cost and performance of cluster supercomputers, together with a quantitative analysis of using low-power ("green") memory modules. We then investigate properties of the total cost of ownership (TCO) as a function of the number of compute nodes. Further, we offer an overview of TCO components for supercomputers. Finally, we propose reusing waste heat from data centres for large-scale greenhouses, outlining an implementation plan.

Part II.

Criterion Function and Design Constraints

6. Choice of the Criterion Function

> There are decision objectives other than maximizing expected return and minimizing maximum loss. That is, in many practical situations there are criteria of optimality that are more appropriate than these two mentioned.
>
> *Russell L. Ackoff*
> THE DEVELOPMENT OF OPERATIONS RESEARCH AS A SCIENCE

As we are solving a combinatorial optimisation problem with many alternative solutions, we need criteria to guide design decisions. The main approaches here are: (a) define a set of criteria and perform multi-objective optimisation, (b) convolve multiple criteria into a weighted additive or multiplicative criterion function and perform single-objective optimisation, and (c) impose constraints on all but one criteria, and optimise the remaining criterion.

Dieter and Dietz [22] used a weighted additive criterion function, while Hsiung *et al.* [43] proposed to produce multiple designs subject to constraints (performance, cost, power, reliability, etc.), and then choose one with the best performance. (Both works are reviewed in more detail in Chapter 2).

There is also a separate approach where alternative computer designs can be evaluated based not on their technical and economic characteristics, but rather on the value of results that the computer can deliver [59]. With this approach, if a computer cannot deliver results on time, or if they are not accurate enough, the "value of calculations" is lowered, down to zero in case of useless results (such as a late weather forecast).

We identified two main criteria for designing cluster supercomputers: total cost of ownership (TCO) and performance on a specific task. Total cost of ownership, reviewed in detail in Chapter 7, allows to capture in a natural and meaningful way many factors that affect procurement and operation of supercomputers. For example, power consumption can be most naturally accounted for by calculating electricity costs during the life span of a computer, instead of trying to artificially introduce it into the criterion function with an arbitrarily assigned weight.

In the following sections we (a) review multi-objective optimisation based on the two criteria, cost and performance, (b) propose a simplest multiplicative criterion function, and (c) explain possibility of single-objective optimisation.

6.1. Multi-Objective Optimisation

Multi-objective optimisation allows to identify *several* non-dominated alternative designs in the set of feasible designs, and then a human designer can choose one of them according to expertise or intuition. To demonstrate multi-objective optimisation with the two criteria outlined above, cost and performance, we performed analysis using the configuration graph depicted in Figure 10.4 from Chapter 10.

This graph allows to generate 264 valid configurations of a compute node (characteristics of hardware can be found in Appendix B). We then used the CAD system for computer clusters, described in Chapter 12, to generate all possible cluster designs based on these 264 node configurations, using a constraint on the minimal performance, which was set to be 240 tasks per day on the `"truck_111m"` benchmark (see section 9.4). We also requested to design a non-blocking fat-tree network and a UPS system with 10 minutes of backup time (more examples of this sort are given in Chapter 17).

It turns out there are 136 feasible cluster designs that adhere to the constraint on minimal performance. Figure 6.1a presents the general view of 136 design points, drawn in the coordinate system of performance and procurement cost. As can be seen, all of them have the requested performance of at least 240 tasks per day, but have a widely varying cost, ranging from roughly $128,000 to $556,000. To facilitate meaningful analysis, we zoom in into the region of interest – that is, range of performance from 240 to 254 tasks per day – in Figure 6.1b. To further simplify view, we disregard expensive solutions, zooming in into the cost range of $120,000 to $170,000 in Figure 6.1c.

The three solutions in the bottom are non-dominated Pareto-optimal solutions, that is, trying to improve one criterion of such a solution – for example, decrease its cost – will lead to another solution where the second criterion is worsened, in this case, performance is also decreased. All other solutions in the figure are dominated, that is, for each such solution there is (at least one) solution that dominates it: either cost is lower, or performance is higher, or both.

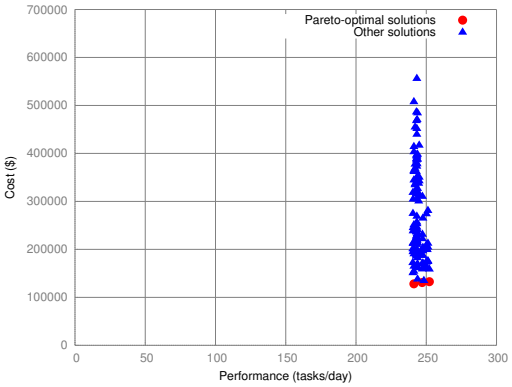
However, after obtaining three Pareto-optimal solutions, we cannot decide which of them to choose, because it requires expertise or intuition from a human designer. This contradicts to the goal of simplicity of the CAD system that we are trying to build (see section 3.7). It would be beneficial to automatically choose one of the non-dominated solutions according to a certain strategy and present it to a human designer. In the following sections we show how this can be done.

6.2. "TCO to Performance Ratio" As a Criterion Function

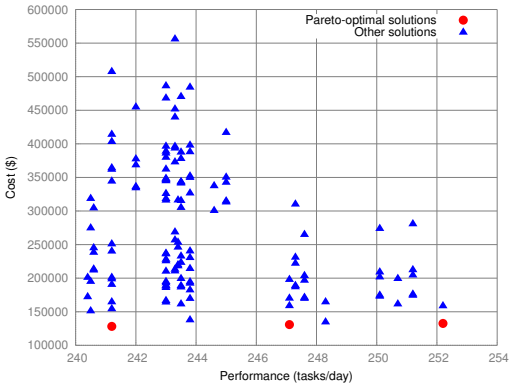
The simplest multiplicative criterion function that utilises the two criteria we identified above, TCO and performance, is just their ratio: $f = \frac{TCO}{Perf}$. The combinatorial optimisation procedure seeks to minimise this function on the set of feasible solutions.

Functions of this type are already used in the computing industry to evaluate quality of designs. For example, IBM's announcement of servers based on POWER7 CPU [46] mentions improvements in metrics such as "price/performance", "performance per watt" and "performance per-square-foot".

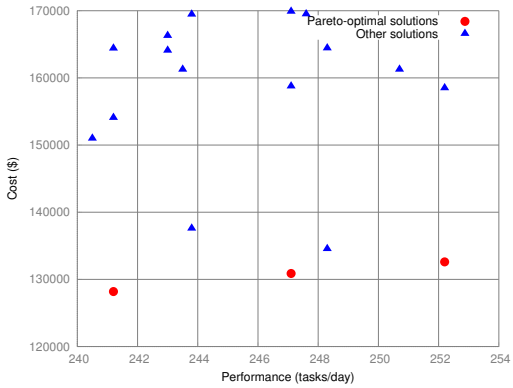
Similarly, HiPEAC proposes [26] to use a criterion function "Performance per Euro" for



(a) General view of 136 feasible solutions



(b) Zoom-in of a region of interest



(c) Zoom-in into less expensive solutions

Figure 6.1.: Graphical representation of 136 feasible solutions, with different levels of zoom.

ordinary computing devices, and "Performance per Watt per Euro" for mobile devices. We, however, believe that the latter function is indeed better suited for mobile devices, and should not be used for designing computer clusters, because it significantly changes ordering of optimal solutions in non-intuitive ways, whereas accounting for power consumption using TCO is more convenient and natural.

Let us consider yet another example where using a "TCO to performance ratio" allows for comparisons in the presence of counterpoising factors.

Example 6.1 *Certain IBM servers, such as the "Power 780" server, can run in so-called "TurboCore" mode [46], when four out of eight cores on the CPU are turned off, the clock frequency is slightly raised, and the chip's resources (L3 cache, memory bandwidth) are shared between the remaining cores. Here we have a complex interplay of factors.*

First, switching off half of the cores seemingly reduces performance of the server. On the other hand, the remaining cores can enjoy slightly higher clock frequency and, more importantly, twice more L3 cache memory and memory bandwidth per core, which can significantly improve performance of some latency-sensitive workloads, somewhat compensating for switched off cores.

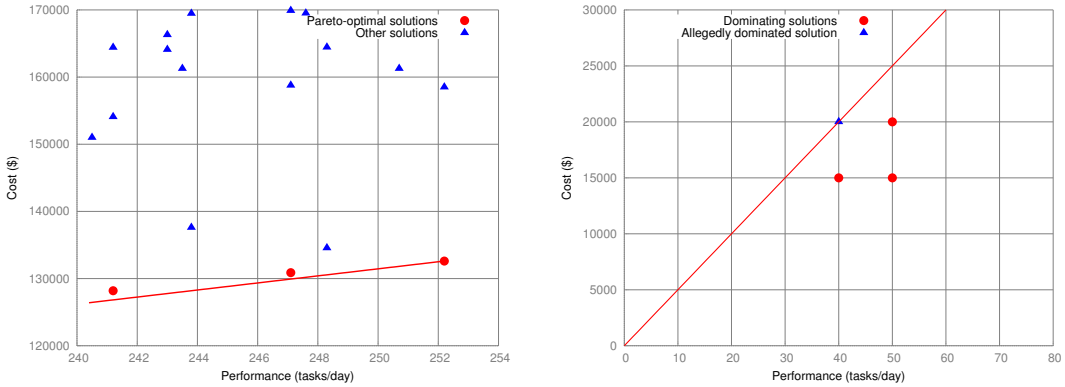
And even if using "TurboCore" mode results in the overall performance drop of a single server, it can still be justified due to the reduction in cost of per-core software licenses. Further, if more servers are necessary to handle the workload in "TurboCore" mode, then additional space in the machine room must be allocated, increasing operating expenses, and so forth. Obviously, there is only one universal way to account for all these factors: carefully calculate total cost of ownership and performance in each case.

We must also note that criterion functions in the form of "Cost to Technical Characteristic Ratio" are not always meaningful. For example, in case of cables, cost per metre is lower for longer cables. However, in practice shorter cables are beneficial whenever possible, despite their higher cost per metre which is just not relevant.

6.3. Making the Case for Single-Objective Optimisation

Let us use the proposed criterion function – TCO to performance ratio – for single-objective optimisation. Remember that we seek to minimise this function on the set of feasible solutions. We refer to the same graphical representation as seen in Figure 6.1c, but this time we supplement it with a straight line originating from the centre of coordinates. We gradually increase the slope of this line, dragging it upwards, until it crosses one of the points representing solutions (see Figure 6.2a). The slope of this line – the tangent of its angle of incline – is now equal to the ratio of cost to performance of the solution that the line crossed. It is the minimal value among all feasible solutions, because the slope was monotonically increased, and it is the first point crossed. Therefore, the solution corresponding to the crossed point is *optimal* in the sense of single-objective optimisation.

Interestingly, it is also one of the non-dominated solutions of multi-objective optimisation. Let us prove it by contradiction. Suppose the point that is first crossed by the line corresponds to a dominated solution. Therefore, there is at least one solution that dominates it: where either (a) cost is lower, or (b) performance is higher, or (c) both; three such possible solutions are shown in Figure 6.2b.



(a) Straight line, crossing the first solution

(b) Proof by contradiction

Figure 6.2.: Geometric representation of single-objective optimisation.

The line's slope is $\tan\theta = \frac{Cost}{Performance}$, therefore in case (a), when cost is lower, the slope is also lower, and this means that the dominating point corresponding to case (a) would be crossed first instead (it's evident from the figure). This contradicts our assumption. Same argument applies to cases (b) and (c). As a result, the assumption was incorrect, and we come to conclusion that the first point crossed is a non-dominated solution.

This is a remarkable result, because we previously found that the first point crossed is an optimal point in single-objective optimisation, and now it is also one of non-dominated points in multi-objective optimisation. This means that a simple, single-objective optimisation procedure automatically yields one of the solutions that a much more complex multi-objective optimisation would produce.

Therefore throughout this thesis we use single-objective optimisation. It is simple to perform, and geometric representation of its results is intuitive to the human designer. However, multi-objective optimisation is still helpful for experienced designers.

We also note that the proposed single-objective optimisation does *not* lead to the same results as the simple constrained optimisation, where a constraint on minimal performance is imposed first, and then the lowest-cost solution is chosen. This can be demonstrated with Figure 6.2a which depicts solutions that satisfy the constraint on minimal performance of 240 tasks per day. Of them, the point in the lower left corner corresponds to the lowest-cost solution (241,2 tasks per day at $128,188). (It is also necessarily one of Pareto-optimal solutions: any other solution that has a better performance also has a bigger cost).

In terms of the proposed single-objective optimisation, however, the optimal solution is the rightmost Pareto-optimal solution (252,2 tasks per day at $132,608), because of the minimal slope of the line passing through it. In simple terms, this solution is "better", because for a slightly bigger cost it offers substantially better performance than the lowest-cost solution. It could happen, of course, that under other circumstances any of the Pareto-optimal solutions could be optimal in the single-objective sense, including the lowest-cost solution.

6.4. Generalisation of the Criterion Function Based on Interval Arithmetic

> He is no wise man that will quit a certainty for an uncertainty.
>
> *Samuel Johnson*

Both components of the criterion function – total cost of ownership and performance – are never known with certainty. Therefore an ordinary number, the result of division of TCO by performance, does not hold information about uncertainty of the input values. There are two ways to deal with uncertainty: fuzzy numbers and interval arithmetic. For the former, we can ascribe membership functions to fuzzy numbers to specify uncertainty; however, we usually don't have enough information to specify a meaningful shape of a membership function. In contrast, interval arithmetic provides a compromise between ability to express uncertainty and intuitive understanding.

Total cost of ownership is a sum of all costs during the lifetime of the system, and if the components of this sum are specified as intervals, thereby incorporating uncertainty of their estimation, then the final result of TCO evaluation will also be an interval, calculated subject to simple rules of interval arithmetic. Performance, whenever possible, should also be represented as an interval.

In the criterion function $f = \frac{t}{p}$, where t is TCO and p is performance, both arguments are positive numbers. The function is increasing with respect to t and decreasing with respect to p. Suppose TCO is represented with the interval $\mathbf{t} = [\underline{t}, \overline{t}]$, and performance with the interval $\mathbf{p} = [\underline{p}, \overline{p}]$. The resulting interval of the ratio is then given by:

$$[\underline{y}, \overline{y}] = [\underline{t}/\overline{p}, \overline{t}/\underline{p}]$$

To illustrate the above, we calculated the cost to performance ratio for 136 designs in Figure 6.1, and then sorted them in ascending order. We then repeated the calculation, representing both cost and performance as intervals with small random fluctuations around the original values. This produced intervals for the cost to performance ratio. The intervals for the best 25 designs are shown in Figure 6.3. Only if the intervals of the criterion function of two designs do not overlap, we can say with certainty that one design is better than the other (if they do overlap, the decision is uncertain).

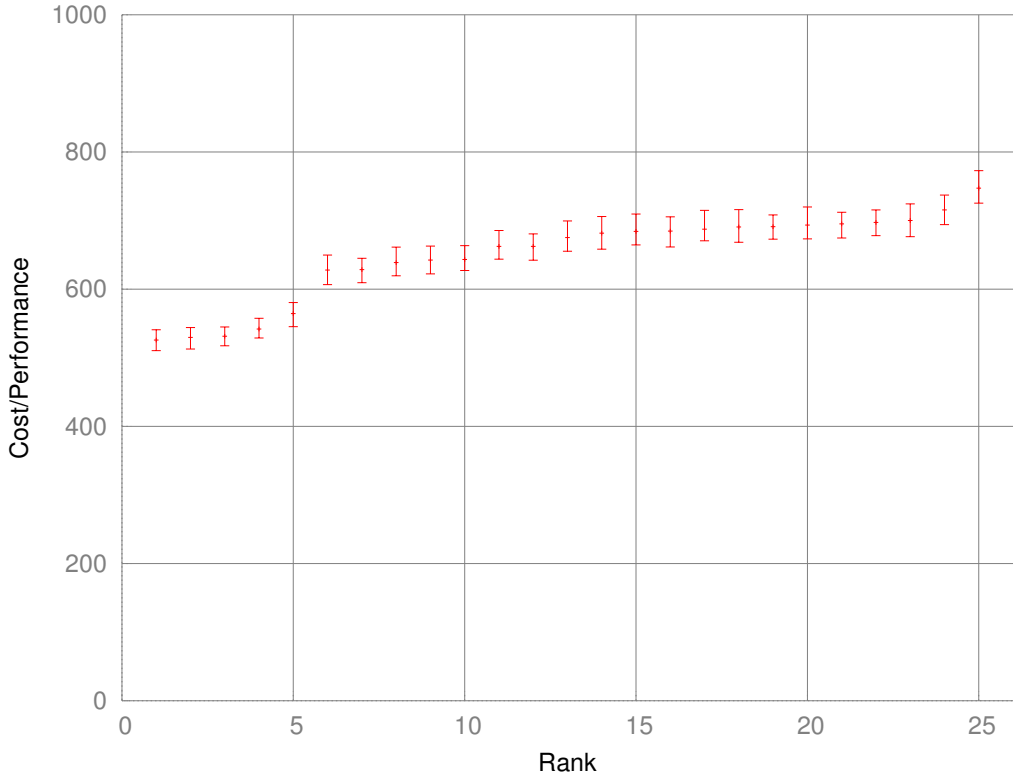


Figure 6.3.: Intervals for the criterion function of the best 25 designs.

7. Economics of Cluster Computing

> Textbooks often ignore the cost half of cost-performance because costs change, thereby dating books, and because the issues are subtle and differ across industry segments. Yet, an understanding of cost and its factors is essential for computer architects to make intelligent decisions about whether or not a new feature should be included in designs where cost is an issue. (Imagine architects designing skyscrapers without any information on costs of steel beams and concrete!)
>
> *John L. Hennessy, David A. Patterson*
> COMPUTER ARCHITECTURE:
> A QUANTITATIVE APPROACH,
> 5TH EDITION

In this chapter we review economics of cluster computing. We start with the discussion of balance: when the budget is fixed and there are multiple contradicting requirements, optimum is achieved by balancing these requirements. We then move to the total cost of ownership, or TCO – a universally accepted measure of costs incurred during the life-cycle of a system. We further explore economic characteristics of cluster computers, and conclude the chapter by highlighting the role of standardisation in reducing life-cycle costs.

7.1. From Innovative to Commodity Technologies

Cluster computing has traditionally relied on *commodity* technologies and mass-produced *off-the-shelf* components. The word "commodity" refers to the fact that a market for a product has little differentiation among possible suppliers: that is, any brand of a product is acceptable. Many components of cluster computers are therefore commodities: network hardware (adaptors, switches and cables), storage hardware (storage systems and hard disk drives), memory modules, and finally motherboards. A notable exception are CPUs, as there are only several vendors, and the choice of CPU dictates many other aspects of the system.

For off-the-shelf components, their characteristic feature is low price, which is due to the fact that mass-production allow to use more efficient technological processes, as well as helps to amortise non-recurring engineering costs over large batches of products. CPU

manufacturing is the example of mass production. Additionally, commodity technology usually also implies mass production.

Switching to a new technology is only reasonable when benefits of switching outweigh all costs associated with replacing a legacy technology with a new one, and overcome hindrances such as inertia in thinking.

When a new, innovative product first appears on the market, it is difficult to maintain low prices because mass production has not yet started. Mass production would allow for lower prices, thereby increasing chances of market acceptance. But mass production is only reasonable to start *after* the product finds market acceptance, and this creates a vicious circle.

This is exacerbated due to the argument similar to Amdahl's law but applicable to economics as well: significantly improving a small subsystem leads to only a limited overall improvement of the whole system (on the other hand, with a similar reasoning, significantly increasing the cost of a subsystem leads to a limited relative increase of the system cost). For example, memory manufacturers can come up with a new memory module, having 50% better performance, measured with a STREAM benchmark. However, using it in computer systems only boosts overall performance by 20%, while increasing overall computer costs by 30%.

In this situation, cluster designers have little economic incentive to try the new memory type, because simply using more compute nodes with older memory technology will result in the same 20% performance boost with roughly linear 20% cost increase. As a result, the fate of innovation depends on market acceptance. In cluster computing, attempts to decrease costs and reduce risk related to vendor lock-in are based on reliance on mass-produced off-the-shelf components available from multiple vendors, which allows to keep costs low. However, unlike the market segment of "traditional" supercomputers where unique components are the norm, in cluster computing reliance on mass-produced components may become a barrier to innovation.

7.2. Questions of Balance

With the criterion function we chose for this thesis – ratio of TCO to performance, see Chapter 6 – it is especially evident that changes to a candidate solution that increase its cost must be accompanied by *exceeding* increase in performance, otherwise the change is inexpedient.

In this section we review several cases of changes in the structure of cluster computers, accentuating that in each particular case a detailed analysis of counteracting factors is necessary to find a proper solution.

7.2.1. Choosing Proper Components

In their comprehensive exploration of the design space of cluster computers, Dieter and Dietz demonstrated that neither intuitive choices of components nor locally optimal choices yield global optimality of a candidate solution: "For example, the best choice of processor is not always the fastest processor, nor the one with the best peak performance per unit cost, nor the cheapest; the best processor is the one that is used in the design that

best meets the requirements determined by your application(s) and budget (e.g., money, space, power consumption)" [22]. To paraphrase, which component is the best can only be determined after considering the entire candidate solution in connection with design constraints and expected use of the system: there are not really any shortcuts, and exhaustive search can be the best alternative.

For example, in 2010, Douglas Eadline analysed technical and economic characteristics of two Intel processors, Atom D510 and Xeon 5570, using *"POV"*, a rendering benchmark [28]. According to the analysis, the Atom chip had a 7,7× lower performance, but at the same time a 22× lower cost, therefore the price/performance advantage of Atom over Xeon was on the order of 2,86×. The author noticed that considering other components as well – including motherboards, memory modules and power supplies – will significantly decrease the given advantage, but still hoped that low-power chips might be suitable for certain workloads. Indeed, in 2013, "Hewlett-Packard" launched so-called "Moonshot" line of servers, based on low-power Intel Atom S1260 CPUs and targeted at data centre workloads [41]. (At the same time, we note that the 22× difference in price for Intel Atom and Xeon chips is heavily influenced by purchasing power of market participants, and may not reflect real manufacturing costs).

7.2.2. Multiprocessor Servers

Symmetric multiprocessor (SMP) servers can be a viable alternative to uniprocessor servers. Several vendors produce dual-processor machines based on X86 architecture, therefore it has now become a commodity technology. Prices gradually went down since such servers first appeared on the market, influenced by mass production and competition. Currently the easiest way to build a uniprocessor machine is to use a de facto standard dual-processor motherboard, but install only one processor.

From system builder's perspective, dual-processor servers allow to double the packaging density by placing two CPUs in a standard 1U rack-mount server. Even more importantly, from programmer's perspective, dual-processor servers are shared-memory machines that are easier to program with OpenMP or similar frameworks than two separate uniprocessor machines.

However, multiprocessor designs must be well-balanced, because performance of subsystems may be divided among the processors. This was the case in the past with the memory subsystem, when CPUs in a dual-processor machine interfaced to memory via a *bus*, which created a bottleneck (now this limitation is overcome by having memory controllers built into CPUs). This is still the case in the present with the network subsystem, where the available network bandwidth is shared among processors. Therefore in case of imbalances uniprocessor servers may be a better choice. For example, in their 2005 analysis [22], Dieter and Dietz found that under a fixed budget for a cluster computer, using uniprocessor servers as compute nodes sometimes yielded better performance of a supercomputer.

Quad-processor servers apparently fell victim to a vicious circle described above: demand for such machines could emerge if their prices were lower due to mass production, but mass production could only start if demand was high enough. As a result, their prices did not fall low enough to make these servers attractive as cluster compute nodes. Additionally, performance of interconnection networks increased to the point that allowed

to program cluster computers using only MPI rather than with a hybrid MPI+OpenMP approach; this made quad-processor servers with fast shared memory redundant, because cheaper dual-processor servers could deliver the same performance. Quad-processor machines remain a useful niche solution in data centre workloads where large amounts of shared memory are important, such as database systems and big data analytics.

Having transparent access to huge amounts of shared memory simplifies programming of HPC software, therefore there have been attempts to combine up to tens of standard servers to form what from programmer's viewpoint is a shared-memory machine. For example, "Gordon" supercomputer [98] combines up to 32 dual-processor servers with proprietary vSMP Foundation software into a shared-memory machine with 2 TB of DRAM memory. Inter-processor communication is done via InfiniBand network, and is transparent for the programmer.

This software-based approach for building shared-memory machines has much higher flexibility and avoids long hardware design cycles. The resulting "virtual" multiprocessor machine uses only commodity dual-processor servers and mass-produced InfiniBand networking hardware, leading to low costs.

7.2.3. Local vs. Network Storage

Computer clusters are often bundled with storage systems that provide a coherent view of the parallel file system to all compute nodes in the cluster. However, such systems are expensive, especially when they are provisioned to support highly-parallel access. For many applications there is a need for *local* temporary storage, where each running thread performs I/O operations with its own set of files. In this case, having a RAID0 HDD-based storage array within each compute node significantly decreases performance requirements on the parallel file system, freeing it (and the interconnection network) for other tasks instead of moving temporary results back and forth.

Here, the choice is either to over-provision the storage subsystem or to install RAID controllers and HDD drives into compute nodes. The latter is a low-cost option to reduce time applications spend for I/O operations; however, there are two possible drawbacks: (a) for applications that don't use temporary files this option is useless and only increases cluster costs, and (b) it also creates additional points of failure in compute nodes.

Regarding the former, Shainer *et al.* [84] note that using a parallel file system, "Lustre", when running CAE software "LS-DYNA" led to performance increase compared to using local disk storage (the paper doesn't detail whether RAID arrays were used or not). Apparently, "LS-DYNA" was unable to efficiently utilise local storage, or this storage was inadequately slow, which led to the necessity of employing a parallel file system. Installing a parallel storage system raises TCO (including maintenance costs); if performance increase is comparable or higher, the installation is justified.

If a RAID controller or any drive in a RAID0 array fails, the job will need rescheduling, decreasing overall job throughput of a computer cluster. If failure rates of components are known, the impact can be analysed quantitatively for various storage configurations. For example, if a RAID10 array raises TCO by 5% compared to RAID0, but only increases job throughput by 3%, its use is not justifiable.

Installing more RAM in a server to be used for disk cache can further boost performance of the local storage subsystem. Therefore the question could be posed as for the balance

between adding more RAM, which is expensive and prone to cache thrashing, and adding more HDD drives to a RAID array, which improves I/O throughput but also has its limits (such as throughput of the RAID controller itself). Yet another approach consists of increasing the amount of RAM or flash memory based caches on RAID controllers. This multitude of choices necessitates careful analysis in search of balance.

7.2.4. Regular vs. "Green" Components

There has been ongoing debate as to whether low-power, or "green" components are suitable for HPC purposes. Generally, within the same semiconductor manufacturing process, specified in nanometres, power savings are attained by using lower clock frequencies and possibly lower voltage; the chips run slower but use less power. Switching to a more advanced manufacturing process (called "die shrink") allows either to (a) use less power at the same frequency, or (b) raise frequency while staying within the same power envelope.

Blumstengel and Arenz [13] analysed power consumption of servers using DRAM modules made with memory chips with differing manufacturing process (30, 40 and 50 nanometres), transfer rates (1600 and 1333 MT/s) and voltages (1,35 V and 1,5 V). For example, in a dual-processor Intel Sandybridge EP server with 64 GBytes of memory running SPECPower benchmark, memory modules ran in two configurations: 1600 MT/s at 1,5 V and 1333 MT/s at 1,35 V. Power consumption of the server was 217 W and 208 W, respectively. Power savings therefore totalled to 9 W, or 4,3%, attributed to both lower memory clock frequency and lower voltage. Authors explain that comparatively low per-server savings translate into large figures for big installations. At the same time, reducing memory speed led to a 2,3% decreased benchmark performance: 497 144 op/s vs. 485 952 op/s, respectively.

The results were then analysed according to the performance/power ratio, that is, how many benchmark operations could be performed per watt of consumed power (higher is better). The metric for a configuration with low-power memory is better, 2336 op/W vs. 2291 op/W (a difference of roughly 2%), thereby seemingly justifying the use of memory modules in low-power mode.

Let us now compare the two configurations according to the TCO/performance ratio we adopted in Chapter 6. We will assume server cost of $5,000 and electricity price of $0,15 per kW·h [29], and will consider TCO to be server cost plus energy costs in the lifetime of four years. For 217 W and 208 W servers, lifetime energy costs are $1,140 and $1,093, respectively. TCO figures are therefore $6,140 and $6,093, and TCO/performance ratios are 0,01235 and 0,01254 (lower is better). With our metric, the high-speed memory configuration is in fact better by roughly 1,5%. This highlights that comparing design alternatives immensely depends on the choice of criterion function. The difference of 1,5% might appear modest, but with large machines, where 1 PFLOPS of peak performance has capital costs on the order of 25 million US dollars (see Chapter 17 for more estimates), 1,5% translates into $375,000.

Further analysis reveals that the high-speed memory configuration remains better until electricity price rises to $0,80 per kW·h (that is, five-fold), which again disputes expedience of using low-power memory configuration.

Overall, we note that operating computer components in low-power mode should be balanced against possible performance degradation. Alternatively, switching to compo-

nents manufactured with a more advanced semiconductor process will provide power savings without performance impact, but incurs higher hardware procurement costs.

7.2.5. Custom vs. Ready-Made Software

In engineering, supercomputers allow to decrease time-to-market for new products using simulation and virtual product prototyping. Project budget can be spent for developing efficient custom software, however, this path is risky and time-consuming. Alternatively, less efficient but ready-made software can be used. This path requires investment into software licenses and more hardware than might be necessary with custom software, but it reduces risks of overdue software projects and overspent budgets.

When creating custom software is inevitable – e.g., because no ready-made software exists – there can be numerous choices as to which hardware architecture to target. It might be easier to write a program for a shared-memory SMP machine than for a distributed-memory computer cluster, thereby conserving expensive programmer time. On the other hand, computer clusters are cheaper to rent than SMP machines, especially with the widespread emergence of cloud providers, so investing time into writing an efficient and scalable MPI program can eventually pay off.

7.2.6. Regular vs. Blade Servers

There are three advantages of using blade servers instead of regular rack-mount servers: first, blade servers are installed into an enclosure and therefore can share certain infrastructural components such as power supplies. For example, an enclosure with 16 blade servers can have just four large power supplies with an added benefit of redundancy, instead of 16 small power supplies, one per each server. Four larger power supplies are also cheaper to manufacture than 16 smaller ones due to economy of scale.

Second, blade servers can connect to network hardware via enclosure backplane, eliminating the need for cables. The backplane connection is also very reliable and not as prone to vibration as cable connections.

The third advantage of blade servers is their increased density. However, according to the law of diminishing returns, further attempts to minimise server volume will be less successful. Indeed, a standard 42U rack can house 42 regular servers. Blade enclosure from "Hewlett-Packard" can house 16 blade servers in 10U space, thereby increasing the number of servers per rack to 64, or by 52%. "IBM iDataPlex" rack further increases the number of servers to 84, or by 31% compared to previous step. Subsequent density increase would be even more difficult to achieve with air-cooled servers, and a massive change to water cooling will be required. Constrained cooling in high-density servers stipulates the use of low-power components such as CPUs which results in performance loss.

The major downside of blade servers is their price, explained by custom development. Blade servers from different manufacturers are not interchangeable, and there are no standards, so each manufacturer has to "reinvent the wheel" and commit an unnecessary R&D spending.

Additionally, decreasing the volume of computing equipment has limited effect on occupied floor space due to reasons similar to Amdahl's law: parts of computing infrastructure,

such as power supply and cooling hardware, are not affected by this decrease. For example, suppose there were four racks of servers, two racks of cooling systems, and one rack of power supply equipment, for a total of seven racks. If server density is increased twice, there will be two server racks instead of four, but the overall number of racks would be five instead of seven, which translates into a modest 29% floor space decrease. Inter-rack aisles are also not affected by server density increase, and therefore further diminish the overall effect. As a result, decreased floor space and savings associated with it may not compensate for increased cost of blade servers.

7.2.7. Fat-Trees vs. "Thin" Trees

Multi-level fat-tree networks are studied in depth in Chapter 13. Navaridas *et al.* [68] demonstrated that blocking fat-trees ("thin" trees) can cause performance degradation for applications with insufficient locality of communications. As a result, there is a balance in choosing (1) a more expensive non-blocking network, or (2) a cheaper blocking network that incurs performance impact.

7.2.8. "Brainware"

Bischof *et al.* [12] observed that a small amount of user projects in a supercomputer centre (15 projects) corresponded to a large amount of computer usage (50%). As a result, even slight performance tuning of several top projects can save energy due to decreased execution times, thereby freeing up money. According to their results, improving performance of the top 30 projects in their environment by 20% would save enough money to pay three HPC experts involved in tuning ("brainware"), plus €0,5M per year more. The balance here is in either (1) using more hardware to run inefficiently programmed software, or (2) using "brainware" – expert knowledge – to make software more efficient without changing hardware resources.

7.3. Total Cost of Ownership and Its Components

Total cost of ownership, or TCO, is a universal measure of expenditures incurred by the system throughout its life cycle. Expenditures are broadly categorised into *capital* and *operating*. TCO arguably provides the most natural way to account for various factors of system operation in a consistent fashion. For example, to account for power consumption of a system there is no need to introduce power into an existing criterion function that contains cost, forming non-intuitive constructs such as "performance per cost per watt". Instead, cost of consumed electricity, which easily reflects power consumption, can be added to other costs. Additionally, this approach allows to naturally take into account variation of electricity prices around the world.

TCO also allows to correctly analyse situations where a product seemingly inexpensive in terms of capital expenditures requires high operating expenditures. If not the TCO, products would be judged based on their capital costs, giving unfair advantage to products that have lower initial price but then require maintenance contracts, unforeseen licensing costs, etc.

TCO calculations can be as complex or as simple as required for each particular case (or as available data allows). Even if no detailed data is available, quoting figures for operating expenditures helps to put numbers into context. For example, Hoefler estimates [42] that electricity costs for a typical server over four years comprise 45% of the cost of the server itself. Another example is a press release by the US Department of Defence that breaks the $105 million acquisition into "$80 million for multiple systems along with an additional $25 million in hardware and software maintenance services" [110] – that is, capital and operating expenditures (note that in this case electricity costs were not included into operating expenditures).

7.3.1. Properties of TCO as a Function

Let us consider TCO of a cluster computer as a function of the number of compute nodes: $TCO = f(N)$. It is defined for integer positive values of N. For practical purposes, this function can be considered monotonically increasing. Additionally, it is characterised by stepping behaviour: a small change in N sometimes incurs a relatively large increase in TCO value. This happens because adding just one more compute node sometimes entails adding a new network switch, a new rack, etc. In other situations, infrastructural components – networks, racks, etc. – have enough reserves to accommodate more compute nodes.

This also explains why the function is not additive, that is, $TCO(A + B) \neq TCO(A) + TCO(B)$. When the cluster of A nodes is considered, adding B nodes might be able to make use of available reserves in infrastructural components, therefore building a single large cluster of $A + B$ nodes could be less expensive than building two clusters with A and B nodes separately. In other situations, however, the result can be different. Let us illustrate this by example, ignoring operating expenditures.

Example 7.1 *The cluster for $A = 20$ nodes requires 20 nodes, one 36-port switch and one rack. The cluster for $B = 16$ nodes requires 16 nodes, one 36-port switch and one rack. The cluster for $A+B = 36$ nodes requires 36 nodes, one 36-port switch and one rack. In this case, $TCO(A+B) < TCO(A) + TCO(B)$, because adding $B = 16$ nodes to existing $A = 20$ nodes uses available resources – switch ports and rack space.*

Let us now consider the case of $A = B = 35$ nodes. Apart from nodes, both small clusters will need one 36-port switch and a rack. At the same time, a cluster of $A + B = 70$ nodes will need two racks and, more importantly, a sophisticated switch with 70 or more ports. The cost of this switch will be higher than the cost of two standard, mass-produced 36-port switches. In this case, $TCO(A + B) > TCO(A) + TCO(B)$.

Calculating $TCO(N)$ by different methods can lead to different results. The major contributing factors are the following:

1. Different composition of costs included in TCO calculation – that is, whether certain components are included or omitted (for example, due to the lack of data). Another situation is attributing costs to either capital or operating: for example, housing a computer can be attributed to capital expenditures (building a machine room) or operating expenditures (renting data centre space).

2. Different prices used in calculation. This includes price difference depending on brand (different makers of similar equipment), market price fluctuations over time, difference in prices across the regions of the world, difference in electricity prices depending on its origin ("regular" vs. "renewable" energy), etc.

3. Different structure of components in the machine, in the way they are connected and placed. One example would be using a two-level fat-tree network instead of a modular switch (Chapter 13). Another example is placing compute nodes into racks in a different manner (Chapter 16); dense placement may result in freeing up some racks, which produces savings in rack cost (capital expenditures) as well as space rental (operating expenditures).

7.3.2. Software

In this and the following sections we give examples of TCO components that are often overlooked. We start with software costs.

Wolfgang Burke, explaining the use of HPC technologies at "BMW Group", the automotive company, notes [18] that the current per-core licensing model needs revisiting: scalability of CAE software is limited, therefore with increasing the number of processor cores speed-up soon flattens out, while licensing costs continue to rise linearly. In other words, CAE software doesn't scale well, and the party that pays for this inefficiency is the customer, not the software vendor.

7.3.3. Installation and Deployment

Installation and integration may require vendor's expertise. Additionally, installation of large systems cannot be done by in-house workforce in reasonable time. As a result, installation costs should be accounted for. Certain site preparation activities, such as installing raised floors for cables and water pipes, can require a large amount of construction work.

Seemingly simple tasks such as pulling network cables can also result in significant spendings: for example, in 2006 "Hewlett-Packard" and "IDC" estimated cost for pulling a cable at $100 [82], although in our opinion this figure is inflated because it is based on the assumption that the pulling operation requires two man-hours.

7.3.4. Power

Holistic approach to power consumption allows to more accurately manage power costs. Brehm *et al.* note [15] that there is an optimal CPU clock frequency for a computer cluster that yields the lowest "energy-to-solution" for a particular job. Setting frequency too low has little effect on overall *power consumption* of the computer in watts, but together with impaired performance it leads to longer time-to-solution, and therefore to the overall higher *energy consumption* in watt-hours. On the other hand, setting too high a frequency, although slightly increasing performance, disproportionately increases energy consumption due to semiconductor leakage currents in CPUs.

European project "FIT4Green" explored possibilities to reduce power consumption in data centres by consolidating virtual machines via live migration and switching off reclaimed servers. The approach was verified in supercomputing data centres (Jülich Super-

computing Centre and other sites) and was estimated to provide energy savings from 4% to 27% [34].

Switching to new technologies allows to achieve lower power consumption per unit of performance. For example, in 2009 the "Jaguar" supercomputer had 2,98 MW/PFLOPS (megawatts per one PFLOPS of peak performance) [105]. In 2012, the "Yellowstone" supercomputer achieved a 0,96 MW/PFLOPS [107] – that is, a 3× better metric value. However, power consumption considerations alone are not enough as a justification to upgrade to new technology.

Indeed, the 2012 electricity price for US commercial customers is $0,10 per kW·h [111], which translates into $0,87M per MW·year. Therefore upgrading from 2009 to 2012 technology would lead to energy savings of roughly $1,75M per year per each PFLOPS of peak performance. However, a 1,5 PFLOPS "Yellowstone" computer is estimated to have a capital cost of $25M to $35M [96], which is an order of magnitude higher per PFLOPS than anticipated $2M per year savings from reduced energy consumption. As a result, decommissioning a supercomputer based on its high energy consumption alone is not justified.

Other factors may come into play, however, such as space savings, because when no more data centre floor space is available for expansion, the alternatives are to either (1) upgrade to a new computer technology, providing more performance within the same floor space, or (2) build a new data centre, which could be even more expensive. Under these circumstances, upgrading becomes economically viable.

Interestingly, power consumption of supercomputers closely relates to data centre floor space: for example, Brehm *et al.* [15] note that in a new data centre built for housing the "SuperMUC" computer [55], the space for computing equipment is 3 160 m^2, while areas for cooling and electrical equipment take up *twice* more space combined – 4 643 m^2 and 1 750 m^2, respectively.

7.3.5. Repair and Maintenance

Computer clusters contain a lot of parts that are likely to fail during operation. This situation is alleviated by the fact that some parts are commodity components that can be sourced from multiple vendors. In any case, during the lifetime of a cluster computer break downs are inevitable, and there must be a strategy to handle them.

One strategy involves repair and maintenance. Downtimes should be minimised, and this could be done by either (1) using more reliable parts (with lower failure rates) – that is, maximising time to failure, or (2) repairing faster – that is, minimising recovery time (the idea of focusing on MTTR rather than on MTTF is due to "Recovery oriented computing" project by Patterson *et al.* [78]).

The former approach requires more expensive parts that have lower failure rates or have built-in fault-tolerance mechanisms. The latter approach requires keeping a reserve of spare parts to facilitate faster replacement, avoiding the need to order them from the manufacturer.

There is, however, an unconventional view that failed compute nodes should be simply disregarded. Based on the concept of recovery-oriented computing, Douglas Eadline supposed [27] that for low-cost compute nodes repair effort may be inexpedient. In this case, he proposes, nodes should be turned off and ignored, possibly until collecting and shipping them to a centralised repair shop at a later time.

We note that the economic viability of this approach generally depends on repair costs that can vary across the world, but overall the approach appears interesting. The main principle here is that staff time is not dispersed for tasks that can be done later and in a more suitable environment such as a specialised repair shop.

Nodes that fail easily have their own implications. The job that utilised the failed node will need rescheduling, and the previous optimal topology-aware task placement can be impossible. The best workaround is for the job to flexibly adapt to the number of compute nodes that the scheduler can allocate.

7.4. Aggregate and Specific Characteristics

Technical and economic characteristics of supercomputers can be divided into two broad categories: aggregate and specific. Aggregate characteristics are those that can be used with the word "total": total performance, power consumption (W), occupied floor space (m^2), volume of equipment (m^3), etc. Specific characteristics are ratios of one aggregate characteristic to another. Some commonly used specific characteristics are listed below:

1. Cost/performance, or TCO/Performance – the metric that we use throughout this thesis. Applicable both across technology generations and across different architectures.

2. Performance/Watt – characterises energy efficiency. Variation of this metric is MW/PFLOPS that we used in section 7.3.4. It is a meaningful metric for comparison between different computer architectures, but across technology generations it mostly reflects advances in semiconductor technology.

3. Performance/m^2 and Performance/m^3. As with Performance/Watt, across generations these metrics mostly reflect advances in semiconductor technology. However, within one technology generation, they clearly characterise manufacturer's ability to densely pack components: water-cooled servers, for instance, have higher values of these metrics.

(There can also be component-level rather than system-level characteristics; for example, for CPUs these can be Performance per MHz of clock frequency, or Performance per mm^2 of die size).

Trends in some technical characteristics over time – i.e., from one generation of technology to another – have been studied. For characteristics such as Performance/Watt or Performance/m^2 growth is slowing down, but it is still continuing, and as new results arrive, marketing specialists still report them as "breakthroughs".

What is less studied, however, are trends in economic characteristics. What were the trends, for example, for system-level Cost/Performance, or better yet TCO/Performance of computer clusters across many generations of semiconductor manufacturing processes? Knowing the trend would help in predicting economic characteristics of future exascale machines. Hindering factors are: (a) cost data for new installations is unavailable to general public, (b) TCO data only becomes available after several years of operation, and again

rarely becomes public, (c) there are different approaches of structuring TCO into components which skews reported results, and (d) economic analysis is required as costs must be adjusted with regards to inflation.

7.5. Public Spending Issues

Intuitively, costs of obtaining one unit of performance (Cost/Performance metric) should decrease with time; however, it is unclear how fast is this decrease, whether it has limits, and whether it has slowed down recently. It is also obscure whether Cost/Performance ratio within one technology generation is approximately equal or drastically different for computers based on different architectures (e.g., between general-purpose CPUs and specialised accelerators such as GPUs). These questions have real-world implications, because in the situation of limited resources investment decisions must be based on facts rather than assumptions. For example, what should be the balance for investing the public funds between (a) funding research in new and more efficient architectures, and (b) funding mass production of less-efficient but cheap processors? Problems of this sort are often tackled by *operations research*, but we need to gather a lot of data even to formulate questions.

There are yet other factors that complicate cost analysis. Cluster computers are built from mass-produced components, but any attempt to introduce new types of components requires R&D effort. This was the case, for example, with the "Gordon" supercomputer [98] which utilised flash drives to create a new level of storage hierarchy. Although flash drives were mass-produced, a considerable R&D effort was necessary to find the ways to use them efficiently. R&D costs cannot be incorporated into cost of the first prototype, because doing so will make the prototype less attractive than same-performing computers based on traditional architectures. Nor can R&D costs be amortised over a fixed number of systems, because it is not known beforehand how many systems using the new design would be produced.

Sometimes an innovation is embodied in a product, and mass production starts. In this case R&D and other non-recurring engineering costs are amortised over large batches. However, public spending doesn't stop at just buying mass-produced items, because significant effort is still needed to make efficient use of technology. A prominent example are GPU accelerators: when this technology became available, institutions that embraced it had to invest time and other resources before they could reap performance benefits – at least by rewriting their software to make use of GPUs. Therefore private investment by GPU vendors was followed by significant public spendings, required for tailoring the product to users' needs.

The story now repeats with Intel Xeon Phi accelerators: there is a learning curve to pass before enough knowledge and skills are accumulated and research laboratories can make full use of the accelerator. Here, public spending also accompanies private investment in the product. But in this case it is somewhat lower than in the case of GPUs, because the Intel Xeon Phi accelerator has a familiar X86-based architecture and software development tools.

The goal for emerging technologies is therefore not only to be cheap in production but also to minimise consequent public spending for tailoring the product to users' needs. Complex functionality should preferably be implemented on manufacturer's side so that

costs could be amortised over many users. This way users won't need to "reinvent the wheel" and duplicate each other's efforts by implementing functionality that could be done by manufacturer once and for all (there is, of course, a balance between simplicity and "over-engineering").

Another way to curb public spending is to use open-source technologies wherever possible, both for software and hardware. This way, if functionality was not provided by the manufacturer but was implemented by a user group, it would become available to other users, lowering costs. A good example is "Rocks", an open-source Linux distribution designed specifically for easy deployment of cluster computers [74]. "Rocks" was developed by University of California, San Diego and contributors, using funding from the US National Science Foundation. During the course of ten years, it was installed on almost 2 000 clusters around the world, thereby relieving the institutions from inventing their own ad hoc solutions, decreasing custom development effort, and lowering costs.

8. Dealing with Combinatorial Explosion

> Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.
>
> *Alan Jay Perlis*
> FROM COMPLEXITY TO SIMPLICITY: UNLEASH YOUR ORGANISATION'S POTENTIAL

8.1. Sources of Combinatorial Explosion

We noted in Chapter 3 that configurations of a technical system (in our case, a compute node of a cluster) to be examined are generated by traversing a multipartite configuration graph. The graph describes compatibility relations between components of a technical system. Trying to oversimplify the graph leads to inability to capture crucial aspects of the system under study. However, expanding the level of detail beyond reasonable limits (and therefore complicating the graph) leads to exponential increase in the number of generated configurations.

Thereby a balance must be maintained: the graph should be detailed enough to capture aspects of the system important for further analysis, but not more detailed than that. For example, as described in section 3.5, the BIOS setting called "Memory interleave" can have impact on application performance, and its boolean value *can* be represented in a configuration graph. However, if performance models employed in subsequent design stages don't make use of the value of this setting, there is no reason to include it in the graph, as this will needlessly double the number of generated configurations.

Even if the configuration graph is carefully optimised by leaving out unnecessary features, it still can generate a considerable number of configurations. For example, in Chapter 17 we find that the sample graph used throughout this thesis generates 264 configurations of Hewlett-Packard's "BL465c G7" servers.

It is always beneficial to decrease the number of configurations to be analysed, because it consequently decreases the number of invocations of design modules (the modules that design interconnection networks, UPS systems, etc.) We employ two approaches to minimise the number of configurations to be examined and shorten the design time: heuristics and design constraints.

8.2. Heuristics

We define a *heuristic* as a technique that is faster than exhaustive search and provides a satisfactory, although not necessarily optimal solution. In this thesis, we advocate for the use of "TCO/Performance" criterion function, and in Chapter 17 we discover that the 264 configurations of a compute node lead to cluster designs with vastly different values of the criterion function; differences as large as five times have been observed.

This suggests that the majority of compute node configurations don't lead to good designs, and consequently we can try to isolate such unproductive configurations. The goal is to find some metric for a configuration that (a) reflects the quality of this configuration as a building block for cluster computers, and (b) can be derived from the configuration's technical and economic characteristic. Then, based on the value of this metric, we can label some configurations as unpromising and weed them out from further consideration. The result is that the size of the design space is significantly decreased, and only a small share of promising configurations is examined.

Two conditions must be fulfilled for the heuristic to be efficient: (1) the metric must be easy to calculate – that is, finding out whether a configuration is promising or not must be easier than subjecting it to the usual design stages, and (2) the metric must be a good predictor of the quality of the cluster computer – in particular, "false negative" errors, when a good or even optimal compute node configuration is erroneously marked as unpromising and rejected, are strongly undesirable.

8.2.1. The Case of Peak Floating-point Performance

Not surprisingly, we found that the value of "Cost/Peak floating-point performance" ratio of a single compute node fulfils both conditions: it is easy to calculate, and it also serves as a good predictor of the respective ratio of the cluster built using such nodes. We conducted an experiment using the same sample configuration graph that we use in Chapter 17. We selected the top 10% of 264 generated configurations (that is, 26 configurations) with the lowest "Cost/Peak FP performance" ratio and designed computer clusters based on these compute node configurations using the following parameters: minimal peak floating-point performance of 100 TFLOPS, a non-blocking fat-tree network, and a UPS system with at least 10 minutes of backup time.

These 26 configurations resulted in the equal number of cluster designs. The heuristic indeed didn't leave out good configurations: in fact, 19 cluster designs out of 26 were the best designs obtained during exhaustive search. In other words, applying this heuristic does not accidentally reject configurations that lead to optimal designs.

We can also formulate this finding more quantitatively: if we assume the quality of the optimal design to be 100%, then the top 10% of designs obtained using exhaustive search had the quality ranging from 100% to 67%, while the top 10% of configurations selected by the heuristic led to designs with the quality ranging from 100% to 57%. We thereby conclude that, when designing computer clusters based on the requirement of minimal peak floating-point performance, the proposed heuristic can reduce design time ten-fold without compromising quality of obtained solutions.

8.2.2. The Case of Application Performance of ANSYS Fluent Software

"ANSYS Fluent" is a CFD software suite which heavily relies on floating-point capabilities of the computer, but its performance also depends on memory and network performance. We therefore were interested in finding whether the previously proposed heuristic, based on peak floating-point performance of a single compute node, can be used to shrink the design space of clusters designed for "ANSYS Fluent" workloads. We repeated the experiment above with the top 10% of configurations selected by the heuristic, this time designing computer clusters with the minimal performance of 240 tasks per day on the `"truck_111m"` benchmark of "ANSYS Fluent" software.

This time, only 6 configurations out of 26 led to actual cluster designs, because the remaining 20 configurations selected by the heuristic could not provide the required performance of 240 tasks per day: although these configurations had an appealing "Cost/Peak FP performance" ratio, they all featured a cheap Gigabit Ethernet network which severely limits performance of "ANSYS Fluent". (This is not surprising, because similar configurations equipped with an expensive InfiniBand network adaptor had a worse value of the metric, and were therefore rejected by the heuristic).

Of 6 configurations that could provide the required performance, 5 used InfiniBand network, and only one used Gigabit Ethernet, as its CPUs were powerful enough (running at the clock frequency of 3 GHz) to limit the required level of parallelism and therefore compensate for poor network scalability. All 6 configurations coincided with the 6 top configurations detected by exhaustive search.

Quantitatively, 10% of best configurations examined by exhaustive search led to cluster designs with the quality ranging from 100% to 70%, while 10% of configurations selected by the proposed heuristic led to 6 cluster designs with the quality ranging from 100% to 84%. Again, the heuristic did not accidentally reject the optimal configuration, and we conclude it can be used for designing computer clusters based on the requirement of minimal performance of "ANSYS Fluent" software.

8.2.3. Conclusion

The above examples demonstrate that we can safely consider a huge amount (e.g., 90%) of generated configurations as potentially unproductive and disregard them. They do not participate in any of the subsequent design stages, which significantly decreases overall design time. For even greater safety, the prototype CAD tool weeds out 80% of unpromising configurations when applying a heuristic, while still leaving the remaining 20% for further analysis.

The proposed heuristic is based on floating-point capabilities of a compute node, and therefore its applicability is limited by floating-point intensive codes. It cannot be used to design computer clusters for workloads such as data mining which don't rely on floating-point operations, because the value of the metric "Cost/Peak FP performance" ratio is a bad predictor of cluster performance on data mining tasks.

8.3. Design Constraints

Establishing constraints on technical and economic characteristics of the candidate solution or its components is an efficient measure that allows to reject unsuitable solutions on as early stages as possible.

8.3.1. Node-level Constraints

A configuration of a compute node has many technical and economic characteristics, and these can be used to filter out unsuitable configurations right after they have been generated via graph traversal. Rejected configurations don't participate in subsequent design stages, which shortens overall design time.

It should be noted, however, that the entire framework for automated design of computer clusters was created for the purpose of *automation*, on the assumption that the complexity of the task is beyond human limits. In other words, guidance from the engineer can sometimes be misleading. With this in mind, constraints should only be established on characteristics when their meaning is evident. For example, we can filter out configurations that have too few main memory per CPU core when we know that application software to be run on the machine will require a particular amount of memory.

On the other hand, it makes little sense to establish constraints on the number of cores in a CPU, apart from trying to exclude outdated CPU models from consideration, in which case they should not be present in the graph database in the first place. It may be tempting to specify that 16-core CPUs should be used in place of 12-core ones, but the engineer may not realise the full complexity of interrelations between components as well as economic aspects involved.

For instance, the compute node configuration which is optimal for the example in section 8.2.1 is equipped with the "AMD Opteron 6274" CPU. This CPU has 16 cores, 16 MBytes of L3 cache memory, runs at the clock frequency of 2,2 GHz, and costs $779. Now consider another CPU, the "AMD Opteron 6238": it has 12 cores, the same 16 MBytes of L3 cache memory (that is, 33% more cache memory per core), has a clock frequency of 2,6 GHz, and costs $569.

For certain applications whose working set fits well in cache memory, the higher per-core cache size of the latter CPU, together with its higher clock frequency and lower price, can make it the optimal choice. To avoid making such decisions on a case-by-case basis, the design task is best left to automated tools. To summarise, constraints should not be established if they are based purely on engineer's intuition.

8.3.2. System-level Constraints

System-level constraints are those that are established on characteristics of the cluster computer as a whole, and not on individual components such as compute nodes. The reader is already familiar with the essential constraint of this sort, the minimal performance of the cluster computer – without specifying it, the design process cannot even start.

There are other system-level constraints as well: economic (capital and operating costs of the computer) and technical (power consumption, volume of equipment, occupied floor space, etc.)

The design process passes each candidate compute node configuration through a sequence of stages (see section 3.4). On each stage, the final configuration of the cluster computer gradually builds up, based on the configuration of the compute node. This allows to reject violating configurations on early stages.

For example, if we determine the required number of compute nodes and find that power consumption limits or equipment volume limits are exceeded, then there is no need to proceed with the next stage that designs an interconnection network; we can reject the current configuration and examine the next one. But if the network was designed, both constraints should be checked again, because after adding network equipment one or both limits could be exceeded. (Constraint checking is a simple and fast operation, much faster than invocation of an external design module).

As in the previous case of node-level constraints, system-level constraints are only beneficial when they stem from real life limitations. For instance, trying to specify a constraint on the volume of equipment, in the attempt to design a "smaller" computer when, in fact, there is a lot of available space, can force the automated design system to use expensive blade servers instead of the ordinary rack-mounted servers, resulting in unnecessarily expensive solution.

It cannot be known in advance whether a particular compute node configuration will be rejected due to system-level constraints – and if yes, at which design stage. Different configurations can be rejected at different stages, but every rejection shortens design time and reduces the number of final cluster designs presented to the engineer.

8.4. The Case Against Local Optimisations

Instead of meticulously searching through many configurations, it can be tempting to perform "local" optimisations – that is, to find an ostensibly "optimal" model of a component, and reject configurations that feature non-optimal models of that component. This will immediately weed out many configurations, seemingly reducing the search space.

For example, one can try to determine an "optimal" CPU model. Hewlett-Packard's "BL465c G7" server can be configured with 18 different CPU models, so choosing one model and rejecting all other models will tremendously decrease the number of configurations to be examined.

In fact, such local optimisations can miss globally optimal solutions. As explained in section 7.2.1, locally optimal choices of components do not lead to global optimality. There is no CPU model which is equally good for all workloads, so examining only one CPU model is likely to miss other, better suited models. Size and complexity of design space clearly exceeds human capabilities, and well-intentioned attempts of optimising locally can lead to very uneconomical solutions. Let us illustrate this by a very simplified example.

Example 8.1 *Suppose we are designing a cluster computer with a minimum performance of 100 tasks per day, and the constraint of 7 units imposed on the equipment volume. There are two servers that can be used as compute nodes: rack-mounted server R costs $2,000 and occupies one unit of space, while blade server B costs $4,000 and occupies 0,5 units.*

Both servers can be equipped with one of the two CPUs. CPU X costs $1,000 and delivers performance of 10 tasks per day, while CPU Y costs $2,000 and delivers performance of 15 tasks

Configuration index	Node type	CPU type	Number of nodes	Performance	Equipment volume	Cost	Cost to performance ratio
1	R	X	10	100	10	$30,000	300
2	R	Y	7	105	7	$28,000	267
3	B	X	10	100	5	$50,000	500
4	B	Y	7	105	3,5	$42,000	400

Table 8.1.: Four configurations of the sample cluster

per day. This way, there are four different server configurations that lead to cluster designs listed in Table 8.1.

One attempt of local optimisation may claim that CPU X has a higher performance per unit cost, therefore it is locally optimal and should be chosen (this decision would favour configurations 1 and 3 that both feature CPU X).

Another line of reasoning may be based on space constraints; one can claim that blade servers B are more likely to meet strict space constraints, and therefore they should be chosen (this would favour configurations 3 and 4, both of which are based on server type B).

However, none of these local optimisations produce the globally optimal solution. It is configuration number 2 that meets both performance and space constraints, has the minimal cost, and is optimal by its "Cost/Performance" ratio. In contrast to this, configuration number 1 doesn't meet space constraints, while configurations 3 and 4 are significantly more expensive than the optimal configuration 2.

As can be seen, local optimisations can lead nowhere. Similar argument applies not only to components of compute nodes but to other cluster subsystems as well. For example, monolithic InfiniBand switches have a lower per-port cost than modular switches (see, e.g., Table 14.3). However, this fact doesn't make such switches universally optimal, because using them in the upper layers of fat-tree networks results in fabrics that are complex to build and maintain and also less reliable.

9. Performance Modelling

> Anyone can build a fast CPU. The trick is to build a fast system.
>
> *Seymour Cray*

In our framework, where the criterion function is the ratio of TCO to performance, the ability to quickly estimate the performance of each configuration of the computer cluster is crucial. We therefore refer to performance modelling. Such models are mathematical objects of various nature that receive the configuration of the cluster computer and output its performance at a particular task.

In this chapter we review related work in the field of performance modelling, identifying factors that affect performance, ranging from obvious to unexpected. We also reveal that the accuracy of performance models is not very high but still sufficient to compare different configurations of cluster supercomputers.

We then introduce *inverse performance models* which, given the required performance, return the number of compute nodes needed to attain it. Finally, we describe a simple performance model for "ANSYS Fluent", the CAE software.

We need to distinguish between *performance* and *speed*. We define performance as an integral characteristic of the computing system or its constituent functional units. Performance is measured on a particular task, using a benchmark. Every functional unit, in turn, can have a number of speed characteristics, determined by the parameters of this unit.

For example, computer's main memory is a functional unit, and its *performance* can be measured using a benchmark – for example, STREAM [58] or another benchmark. Different benchmarks measure performance differently, so even if the units of measurement are the same (such as MBytes/sec in STREAM), the figures will be different. As a functional unit, main memory has two distinct *speed* characteristics: bandwidth and latency. Bandwidth characterises how fast the memory can transfer large amounts of data, and is measured in bytes per second, while latency characterises how long it takes the memory to deliver a single word, and is measured in seconds (or smaller units of time).

There are a number of *parameters* of main memory that influence both bandwidth and latency, and therefore also influence its performance as a whole. Such parameters include: memory bus width, clock frequency, access timings of DRAM chips, whether memory is registered, whether interleave mode is turned on in the BIOS, etc. (See Figure 9.1). The same applies to other functional units of a computer system.

Relations between unit's parameters and its speed characteristics, and consequently its performance, are often complex and unclear. For example, there is no universal answer as to what effect on memory bandwidth (or latency) will produce a 10% increase in memory clock frequency. Even if this effect was known, it still would be difficult to estimate how

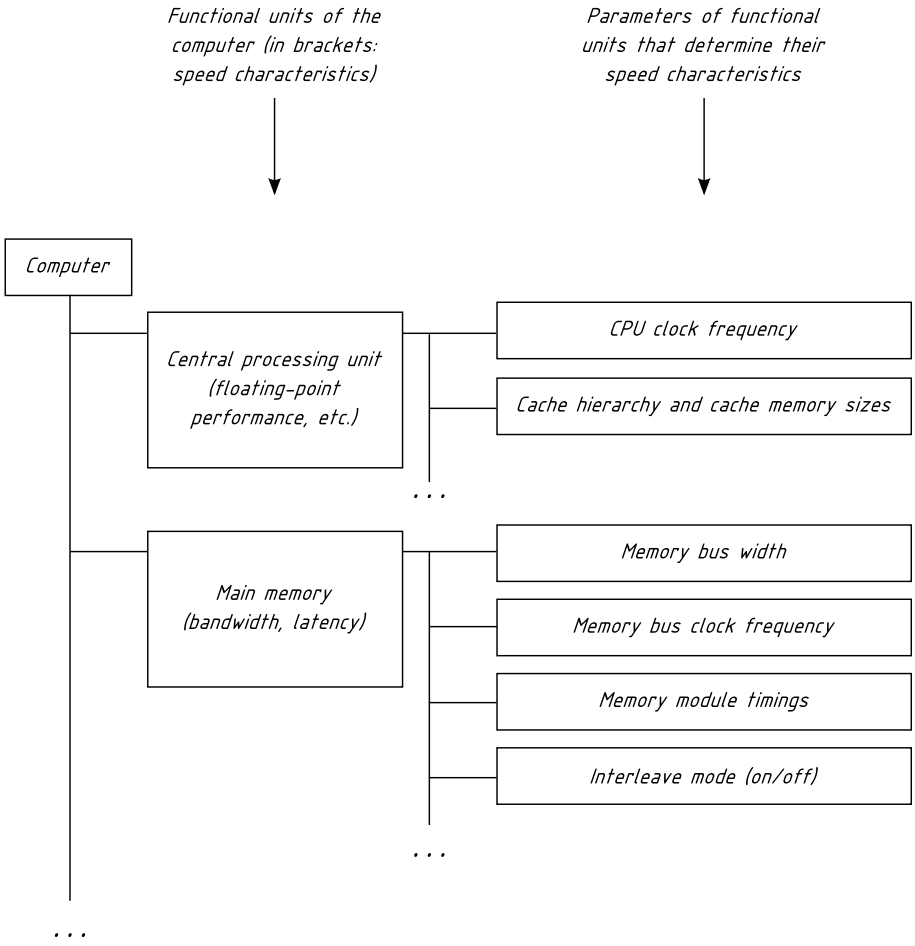


Figure 9.1.: Functional units of the computer system, their speed characteristics and parameters.

it would affect performance of real life applications, especially in the presence of modern complex cache memory hierarchies.

Performance of the entire computer is dependent on *many* of its functional units and subsystems. As a result, cluster designers often have to rely on rough estimates of performance, because more detailed data is not available. Additionally, communities that design subsystems – memory, interconnection networks, etc. – are isolated from each other and from system-level designers, because there are no reliable ways to predict the effect that enhancements in separate subsystems can have on the overall system performance.

9.1. Related Work

In this section we describe existing work in various fields related to performance modelling, ranging from microarchitectural to system-level aspects.

9.1.1. Factors That Influence Performance

Performance problems may come from heterogeneity of modern hardware (such as NUMA architecture), and inability of runtime environments to automatically take care of this. For example, multi-threaded applications should have their threads "pinned" to specific CPUs, so that memory primarily accessed by the thread is "local" to the CPU. Treibig *et al.* [108] analysed performance of memory subsystem of a dual-socket server based on Intel Nehalem CPU, using STREAM benchmark, with and without "pinning" of threads.

The results indicate that pinning, enforced at runtime, resulted in considerably higher observed memory bandwidth (e.g., 46% higher on average for 24 threads), although the source and binary codes of the application under study (STREAM) remained the same.

Balaji *et al.* [10] note that inopportune mapping of MPI processes to compute nodes in torus networks leads to performance degradation, proposing a methodology for topology-aware mapping. We additionally note two factors. First, such inopportune mapping can occur in other network topologies as well, for example, in fat-trees. Indeed, if the majority of MPI processes are mapped onto compute nodes within one sub-tree, while some processes are mapped onto nodes in a different sub-tree, accessible only via the root-level switches of a fat-tree, then the MPI communication performance will suffer from the high-latency connection between the two groups of processes.

The second factor worth noting is that this performance degradation is, in a sense, unavoidable: a new large compute job may find a supercomputer in a "fragmented" state, when there is no a single contiguous block of compute nodes available within one sub-tree to guarantee locality of communications. In this case, the job will be assigned to blocks of compute nodes residing in different parts of the interconnection network, and performance degradation will be observed.

In theory, smaller compute jobs, already running, could be migrated to "defragment" the cluster computer. Another approach could be to hold the job in a queue waiting for a contiguous block of compute nodes becoming available, however, this leads to a lower utilisation of available resources, and although the run time of the job could be decreased by achieving preferential process mapping, the total time to solution – from job submittal to termination – may, in fact, increase.

A study by Subramoni *et al.* [100] describes a method of placing MPI processes onto compute nodes, automatically taking network topology into account. This allows to place heavily communicating MPI processes in proximity, potentially even within the same compute node, demonstrating performance gains of up to 15%. Again, this increase in performance is achieved at runtime, without any changes to source or binary code of the application.

From the examples described above follows the difficulty for accurate performance modelling. There are also quite obvious cases when performance variations will be observed, for example, when using different MPI implementations at runtime, or linking against different mathematical libraries: community-provided open-source libraries versus vendor-specific optimised libraries such as "AMD Core Math Library" or "Intel Math Kernel Library". Trying to account for these effects would lead to increased complexity of performance modelling.

We present one more example concerning factors that cause variation in observed performance and therefore complicate construction of simple and efficient performance mod-

els. Shainer *et al.* [84] note that using a parallel file system, "Lustre", for I/O when running CAE software "LS-DYNA" led to roughly 15% performance increase when using 64 cores in a cluster, and a 20% increase when using 128 cores. (They don't note, however, what increase in cost is incurred by using a parallel storage system based on "Lustre").

We have reviewed situations where the observed performance variation could be expected. Let us now refer to a number of cases where performance degradation was unexpected or counter-intuitive.

Mytkowicz *et al.* [64] describe how performance is affected by (a) the size of memory used to store UNIX environment variables, and (b) the order of object files in the command line of the linker, when linking the executable file of the application under study. Chan *et al.* [20] observed a detrimental effect of cooling fan vibrations on hard disk bandwidth, which led to performance degradation for I/O intensive jobs. In both of the described cases, a performance model can hardly account for such runtime effects.

Pase and Reddy [77] studied performance of "ANSYS Fluent" CAE software on servers made by IBM. One group of tests involved running the software using MPI, but within a single server. Choosing MPI library optimised for InfiniBand hardware, compared to the standard MPI library for Gigabit Ethernet network, generally resulted in a small performance boost (on the order of 10% in most cases), although the network hardware of the server was never used, and all MPI communications were in the memory.

Their study also found that using only one CPU in a dual-socket server reduces pressure on the memory subsystem, therefore using twice as many uniprocessor servers, compared to dual-processor servers, yielded performance gains on the order of 40..50% for large compute jobs, despite increases in network traffic. It means that the application under study, "ANSYS Fluent", was more sensitive to memory performance than to network performance.

As a result, using four uniprocessor servers led to 41% higher performance than using two dual-processor servers. We note, however, that acquisition cost and the total cost of ownership for the former configuration are likely to exceed those of the latter by more than 50% or 60%, making the former configuration economically disadvantageous.

Another interesting effect that makes performance modelling difficult is observed when using Intel Turbo Boost (or similar) technology: with it, the clock frequency of a CPU can be significantly increased when power and thermal constraints allow, e.g., when only a few cores are active. For example, on a 4-core "Intel i7-920XM" CPU, when only one core is active, the clock frequency can increase from 2,0 GHz up to 3,2 GHz – that is, by 60%. This means that the performance model must take into account not only the base frequency of a CPU, but also its possible increase.

However, frequency scaling depends on runtime conditions: in some installations, there is no cooling reserve, and scaling will never occur, while in water-cooled machines, such as "SuperMUC" [55], Turbo Boost can operate continuously [15] (although leading to a higher power consumption of the machine).

This could be dealt with by specifying as an input to the performance model whether Turbo Boost will be available, and to which extent. However, such additions make the model more complex and less elegant.

There is an open question regarding functionality of a performance model: given the total number of cores in a computer cluster, shall it simply predict the performance of a given code on this machine, or shall it advice optimal placement of processes on compute

nodes? For example, if the performance model "knows" that a certain software has a speed-up of 50% when running on 4 cores, should it recommend running the software in a single-thread (sequential) mode, using only one core, possibly utilising a Turbo Boost frequency scaling of 60%, with associated overall performance gain, or should the model simply report the performance estimate on 4 cores, as requested by the user, and exit? This extra piece of information is useful, but we need to find the way to communicate it to the higher levels of the cluster design framework.

Yet another side effect results from differently clocking CPUs in NUMA systems. Consider the following scenario: a compute job utilises only the first CPU on a dual-processor server. The remaining CPU is not participating in calculations and is therefore clocked at a reduced frequency to save power. However, if it happens that memory used by the compute job utilises memory modules connected to the second CPU, which is not running at full speed, then accesses to that memory will be characterised by bigger latencies, degrading overall performance of applications running on the first CPU.

We have not experimentally verified whether such a problem really exists in modern NUMA systems. In HPC environments all available CPU sockets are usually utilised, so the chances for this problem to manifest itself are low. However, it highlights how complex interrelations between components of modern computing systems can lead to obscure effects which are difficult to capture in performance models.

9.1.2. Approaches to Performance Modelling

We argue that a single universally accepted measure of performance of a certain code on a certain computer is the *time* to solution. For example, measuring performance in GFLOPS makes no sense for codes that do not perform floating-point operations, such as graph-traversal codes. For this class of workload, performance can be measured in edges traversed per second (TEPS), but this, in turn, makes no sense for other types of workloads, such as Map-Reduce, etc.

Therefore, time remains a universal measure. In practice, a reciprocal measure is more convenient: the number of times this code can be run in a unit of time – for example, in a day: it's more natural for humans, because higher is better. CAE software "ANSYS Fluent" uses "performance rating": the number of times that a certain benchmark can be run in 24 hours. For example, a performance rating of 240 indicates that a computer can solve 240 tasks per day, or 10 tasks per hour. Therefore, time to solution for a single benchmark run is 6 minutes. There are no other reasons for the use of performance rating instead of time except convenience; in other respects they are equivalent and reciprocal.

Performance model is an object of arbitrary internal structure that can predict performance. As a result, both parameters of the computer used to run the code, as well as parameters of the code, must be supplied to this model. It is evident that adding more inputs to the model leads to its exponential complexity.

Benchmarks provide a convenient way to reduce the number of code parameters that must be supplied to the performance model. A familiar example is the "Linpack" benchmark that is used as a "proxy" for applications involving linear algebra computations. However, it is only relevant as a proxy if the application behaves in a similar way to "Linpack": for example, if it uses sparse, rather than dense matrices, and so on.

Therefore there is a multitude of benchmarks for different classes of applications, made

by scientists working in the field of workload characterisation. There have been efforts to create parametrised performance benchmarks that would be able to serve as performance proxies for *any* application – see, for example, Apex-Map benchmark by Strohmaier and Shan [99], which has three parameters: memory size used by the code which behaviour is to be modelled, as well as temporal and spatial locality of memory accesses made by this code.

Now that the problem of specifying parameters of the code for the purpose of performance modelling has been more or less solved by using representative benchmarks, it is necessary to understand which parameters of the computer must be supplied to a performance model to get a performance prediction within a certain error.

In the simplest case, a performance model can take into account a very limited number of factors, such as the clock frequency of the CPU, the total number of cores in the compute cluster, and type of the interconnection network. We introduce such a model below. Its simplicity leads to its limited applicability: for example, it doesn't take cache memory size on the CPU into account, therefore it cannot be used to quantitatively compare clusters built with CPUs with different cache memory sizes. Qualitatively, more cache memory can lead to better performance, but this was obvious without the model, and the model can't capture this dependency.

To capture more aspects of the hardware, simulators can be used, such as PTLsim [115], gem5 [11], Multi2Sim [109] or Simsys [71]. Simulators have a benefit in that they can simulate execution of arbitrary code, not just an existing benchmark or performance proxy. Simulators allow to explore a much larger hardware design space (down to accurate timing of memory access operations, for example), albeit at expense of long run time.

However, the necessity to explore the design space – the reason to use simulation in the first place – is exacerbated by the fact that simulators run considerably slower than real hardware. Therefore, using simulation is usually beneficial only when hardware being simulated is not existent yet – for example, when trying to understand how different designs of cache memory hierarchy would influence performance of an application.

Because simulation times are so long, there have been attempts to reduce them. The first approach is simulating only part of the binary code of a big program, and then the challenge is to find the smallest part of the code which is still representative with regard to behaviour of the whole program – see, for example, paper [86] by Sherwood *et al.*

The second approach doesn't change the code to be simulated, but instead tries to explore only a tiny part of the total hardware design space, and then employs different approximation techniques to build mathematical models that would use interpolation to predict performance with simulation accuracy but in less time. Joseph *et al.* [50] used nonlinear approximation model based on artificial neural network with radial basis function (RBF) activation functions, while Ipek *et al.* [49] used sigmoid activation functions. In both of these studies, microarchitectural CPU parameters were varied, while the applications under study were fixed (SPEC CPU2000 benchmark suite).

Later, Lee *et al.* [54] compared approximation approaches based on artificial neural networks and on piecewise polynomial regression using cubic splines, and found that both approaches have roughly the same accuracy. In this case the methodologies were used to study performance of two applications (Semicoarsening Multigrid and High-Performance Linpack) on three existing parallel computers (BlueGene/L with 512 compute nodes and two Intel Xeon-based clusters with 64 compute nodes), and therefore involved not simu-

lation but measurement. In contrast with the previous work, machine parameters were fixed, and application parameters were varied. Both methodologies could predict performance with a median error of 10% and less.

9.2. Throughput Mode

Scale of parallel computers continues to increase, but older codes are not able to efficiently utilise available hardware parallelism. As a result, while degree of parallelism (for example, the number of cores) increases, application speedup deviates from the straight line ("linear speedup") and then plateaus, indicating that employing more hardware to do the work leads to diminishing returns.

Determining the degree of parallelism N_{max}, after which performance increase is considered negligible, is purely subjective. In certain applications, such as urgent computing, even a modest increase in performance can justify putting more hardware to the task. In general, we can arbitrarily define "efficient" execution. For example, suppose we define *efficiency* as the ratio of speedup to the degree of parallelism N, presented as a percentage. As N is increased, speedup plateaus, and therefore efficiency decreases. We can define execution as "efficient" when efficiency stays above, say, 70%. The N_{max} corresponding to this efficiency is then considered the maximal reasonable degree of parallelism.

When an application is to be run on a parallel computer with the number N_f of parallel computing units (cores, CPUs, compute nodes, etc.) bigger than N_{max}, it runs in what is called "throughput mode". In this mode, several copies of the application are run, each in its own partition of the parallel computer with the degree of parallelism not exceeding N_{max}. Aggregate performance in throughput mode is the sum of performance figures of individual partitions. One simple strategy is to create as many partitions of size N_{max} as possible, and the remaining computing units will then form the last, smaller partition. This strategy leads to lowest time-to-solution of individual tasks, but may have a lower overall throughput.

Example 9.1 *Consider an application which has a maximal reasonable degree of parallelism* $N_{max} = 200$ *(say, cores), where it achieves performance rating of* $P_{max} = 4$ *(tasks per day). It is also possible to run this application on* $N_1 = 100$ *cores with performance of* $P_1 = 2{,}5$*, and on* $N_2 = 50$ *cores with performance* $P_2 = 1{,}4$*. Let us consider two possible scenarios for partitioning the parallel computer of* $N_f = 500$ *cores for this application.*

Scenario A. Create two partitions of 200 cores each, and the third partition of 100 cores. In one day the throughput of the system (the total number of tasks solved) will be 4+4+2,5=10,5. The minimal time to solution is obtained on the first and second partitions and equals 6 hours (because 4 tasks are solved in 24 hours).

*Scenario B. If the goal is to maximise the throughput, and obtaining the result of each individual task as soon as possible is not important, then one can assign 10 tasks to 10 partitions, each consisting of 50 cores. In a day, 10 partitions will solve 1,4*10=14 tasks (which is 33% more than in Scenario A). The time to solution of each individual task is, however, considerably longer: 24/1,4=17,1 hours.*

9.3. Direct And Inverse Performance Models

9.3.1. Definition

As we defined it earlier, a performance model is an object of an arbitrary internal structure that can predict performance of a certain code when executed on a certain computer. To facilitate the prediction, parameters of the code P_{code} and of the computer P_{comp} must be supplied to the model D:

$$D : (P_{code}, P_{comp}) \rightarrow Performance$$

In the context of parallel computing, we can operate with parameters of the elementary building block of a parallel computer P_{comp_block} and the number of such blocks N. (Building blocks can be defined differently depending on the level of abstraction: floating-point units within a core, cores, CPUs, compute nodes, etc.) With this in mind, we can reformulate the general definition above as follows:

$$D : (P_{code}, P_{comp\_block}, N) \rightarrow Performance$$

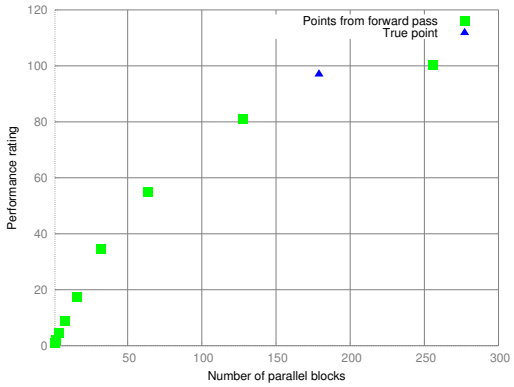
We now define an *inverse* performance model, specific to parallel computing: it is an object that, when supplied with parameters of the code, parameters of the elementary building block of a parallel computer and the performance that must be attained, returns the number of blocks in a parallel computer:

$$I : (P_{code}, P_{comp\_block}, Performance) \rightarrow N$$

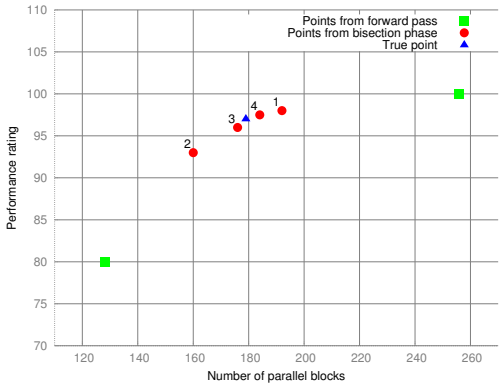
The notion of inverse performance models is applicable from chip to server and up to system level. On a chip level, consider Intel Xeon Phi as an example: if microarchitectural details are specified, and clock frequency is known, how many cores must be on the chip to attain a specific level of floating-point performance? If each core has a peak rate of 16 double-precision floating-point operation, and clock frequency is 1.053 GHz, then 60 cores are required to achieve a "tempting" value of 1 TFLOPS of peak floating-point performance.

On a system level, if parameters of compute nodes (type of CPUs and their clock frequency) and the type of interconnection network (e.g., Gigabit Ethernet or InfiniBand) are specified, then the inverse performance model can answer the question: how many cores in a computer cluster are required to attain a time to solution of 5 minutes on a specific benchmark of "ANSYS Fluent", `"truck_111m"` (equivalent to performance rating of 288 tasks per day). For example, a performance model that we construct below predicts that for a clock frequency of 3,47 GHz the number of cores required to reach that performance with Ten Gigabit Ethernet network is 374, while with InfiniBand it is only 312. This conclusion couldn't be easily derived from assorted benchmark results, and inverse performance modelling made it possible.

We deliberately mentioned that a direct performance model is an object with an arbitrary internal structure. Inside the model there can be (a) an analytical model, such as a formula, (b) an approximation model, such as one built using artificial neural networks, (c) a simulator, (d) an FPGA prototype, etc. Therefore querying a direct performance model can be a potentially lengthy task.



(a) First phase, with nine forward iterations



(b) Second phase, with four iterations of the bisection method

Figure 9.2.: Inverse performance modelling

9.3.2. Algorithm

Inverse performance modelling can be performed quickly only in one case: when an explicit formula for N can be derived from a direct performance model, which is itself specified via a formula. Such cases are unlikely to occur in practise. For all other occasions, we introduce a simple iterative algorithm for inverse performance modelling that involves querying a direct performance model several times.

The algorithm relies on monotonic increase of performance as a function of N. The main input of the algorithm is the performance rating P that must be obtained, and the output is the number of parallel blocks N. It works in two phases: in the first phase, the algorithm queries the direct performance model, gradually increasing the number of blocks N by a certain factor, until reported performance rating is greater than or equal to P. For example, if the requested performance rating is $P = 97$, then the first phase will stop at $N = 256$, as it is in this point where the requested rating is exceeded (see Figure 9.2a).

In the second phase, the algorithm refines the value of N using bisection method, searching between the last and the penultimate values of N. In our case, it searches the interval of $N = 128..256$ until the value is found that corresponds to performance P, accurate to the required precision. Four iterations of the bisection method use the following values of N: 192, 160, 176 and 184 (see Figure 9.2b).

Let us describe the algorithm by stages. An important input parameter is N_{max}, the maximum reasonable number of parallel computing blocks, where application performance starts to plateau. It depends on parameters of the application (P_{code}) and parameters of machine architecture (P_{comp_block}). Another application (or the same application with different input data), or a different structure of the parallel computer will yield a different value to N_{max}. Performance observed at N_{max} is denoted by P_{max}; performance higher than this can be achieved only in throughput mode.

1. The algorithm starts with checking whether the requested (projected) performance P_{proj} was greater than the reasonable maximum P_{max} (line 1). If yes, it informs about throughput mode, returns N and exits. N is calculated as N_{max} scaled by the ratio of

Algorithm 1 Inverse performance modelling

Input:

P_{proj}: Performance rating to achieve
N_{max}: Maximum reasonable number of parallel computing blocks
P_{max}: Performance rating observed at N_{max}
$Performance\ (N) = f(N, P_{code}, P_{comp\_block})$: Direct performance model

Goal: :

N: Number of parallel computing blocks that yields performance rating P

1: **if** $P_{proj} \geq P_{max}$ **then**
2: { Requested performance P_{proj} greater than the reasonable maximum P_{max} }
3: **print** Throughput mode
4: $N \leftarrow N_{max} \cdot P_{proj}/P_{max}$
5: Exit
6: **end if**
7: $N \leftarrow 1$; $P \leftarrow Performance\ (N)$
8: **if** $P \geq P_{proj}$ **then**
9: Exit { One block was enough to deliver required performance }
10: **end if**
11: { First phase: forward pass }
12: **while** $P \leq P_{proj}$ **do**
13: $N \leftarrow N \cdot MulFactor$ { Increase N until P_{proj} or higher rating is reached }
14: **if** $N \geq N_{max}$ **then**
15: $N \leftarrow N_{max}$ { Do not exceed N_{max} }
16: **end if**
17: $P \leftarrow Performance\ (N)$
18: **end while**
19: { Second phase: bisection method }
20: $N_r \leftarrow N$ { Right interval boundary }
21: $N_l \leftarrow N_r/MulFactor$ { Left interval boundary }
22: $P_{prev} \leftarrow 0$
23: **while** $(N_r - N_l > \varepsilon_N)$ and $(|P_{prev} - P| > \varepsilon_P)$ **do**
24: $N_m \leftarrow (N_l + N_r) \div 2$ { Interval centre }
25: $P_{prev} \leftarrow P$
26: $P \leftarrow Performance\ (N_m)$
27: **if** $P > P_{proj}$ **then**
28: $N_r \leftarrow N_m$ { Adjust right boundary }
29: **else**
30: $N_l \leftarrow N_m$ { Adjust left boundary }
31: **end if**
32: **end while**
33: $N \leftarrow N_r$

P_{proj} to P_{max}. This corresponds to partitioning strategy leading to minimal time to solution, as described in Scenario A in Example 9.1.

2. The algorithm checks if $N = 1$ parallel computing block is enough to deliver requested performance (line 7). This can be the case, for example, when the block is a powerful compute node with several multicore CPUs.

3. If one block was not enough, the algorithm performs the forward pass: it increases N by $MulFactor$ until projected performance P_{proj} is reached (line 13). Additional check makes sure that N doesn't exceed N_{max}. By the end of the "while" cycle, N receives such a value that the corresponding performance $P(N)$ (the last point in the sequence of forward pass points in Fig. 9.2a) "overshoots" the projected performance (the "True point" in the same figure).

   $MulFactor$ controls the rate of increase of the forward pass, and consequently how many queries of the direct performance model will be required; we found the value of 2 to be suitable in practice.

4. As a result of the forward pass, the point that we are looking for is now located between the last (N) and penultimate ($N/MulFactor$) points (lines 20 and 21). The algorithm employs bisection method to refine the value of N. The interval is bisected until it becomes shorter than ε_N, or when performance $P(N)$ doesn't change much (performance obtained on the previous step, P_{prev} (line 25), is different from current value of performance by no more than ε_P).

   As shown in Fig. 9.2b, the initial interval [128, 256] was bisected four times, each time followed by adjusting one of its boundaries. Performance was evaluated in four points: $N = 192$, leading to interval [128, 192], $N = 160$, leading to interval [160, 192], $N = 176$, leading to interval [176, 192], and finally at $N = 184$, leading to interval [176, 184], after which the process was considered to have converged, as the interval length $184 - 176 = 8$ has become less than $\varepsilon_N = 10$.

   The larger the values of ε_N and ε_P, the larger the uncertainty of N, but at the same time the lower the number of queries of the direct performance model.

5. Finally, the algorithm chooses the right boundary of the interval as a value of N, thereby ensuring that resulting performance $P(N)$ returned by the inverse model is not lower than requested (line 33). In this case, $N = 184$ will be returned.

The algorithm relies on monotonic increase of performance P as a function of N, but this holds in practice. Additionally, results of performance evaluation can be cached for later reuse, which vastly decreases the number of queries to the direct performance model. This is important, because, as we noted earlier, the direct performance model can have an arbitrarily complex structure, ranging from an artificial neural network to an FPGA prototype.

9.4. Simple Performance Model for ANSYS Fluent

We now describe a simple analytical direct performance model for computational fluid dynamics (CFD) software suite, "ANSYS Fluent", that approximates measured benchmark

results with a quadratic polynomial. The model serves to demonstrate capabilities of our automated design framework, with results of practical evaluation presented in Chapter 17.

To create the performance model, we used measured benchmark results for "ANSYS Fluent" version 13, obtained on a benchmark task of modelling air flow over a truck body, `"truck_111m"` [6]. The corresponding CFD model contains 111 million cells and accounts for turbulence in the flow. Measured results were available for Intel Xeon 5600 series CPUs, and for two types of interconnection network hardware: Ten Gigabit Ethernet and InfiniBand.

We didn't set the goal of building a very precise performance model: first, as shown earlier, in real life situations significant performance variations are observed even in similar software and hardware settings, and it makes no sense to build a model to be more accurate than the object it tries to simulate. Second, this performance model is mostly intended to serve demonstration purposes as part of the design framework. However, even in its current state the model proved useful for "sizing" cluster computers tailored to running "ANSYS Fluent" jobs.

We used an entirely ad hoc approach to construct the performance model. First, after analysing benchmark data, we determined boundaries of efficient execution. We defined efficiency as the ratio of speedup to the degree of parallelism – that is, the number of cores involved in the computation:

$$Eff = \frac{S}{N}$$

With Ten Gigabit Ethernet network, efficiency decreased unacceptably after $N_{max} = 384$ cores, while with InfiniBand network efficient execution was observed up to $N_{max} = 3072$ cores. We noted that in the region of efficient performance, efficiency can be approximated with a linear fit:

$$Eff(N) = k_e \cdot N + b_e$$

For InfiniBand, linear fit yielded the following parameters: $k_e = -0{,}00979, b_e = 88{,}42$. For Ten Gigabit Ethernet we didn't have enough measurement data available, so we assumed that in the region of $N < 192$ cores, efficiency would be approximated by the same formula that was used for InfiniBand, and in the region $192 \leq N \leq 384$ the following parameters would be used: $k_e = -0{,}08270, b_e = 101{,}98$. Graphs for efficiency are shown in Figure 9.3a; note that efficiency for Ten Gigabit Ethernet is piecewise linear.

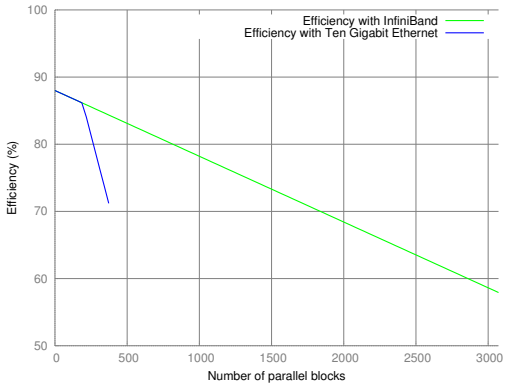
By definition, speedup is the ratio of parallel performance to serial performance:

$$S(N) = \frac{P_{par}(N)}{P_{ser}}$$

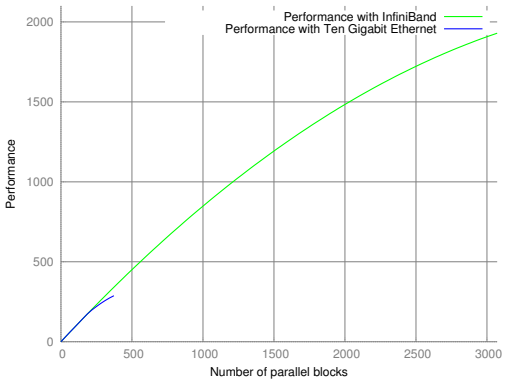
We assumed that we can approximate serial performance P_{ser} as a linear dependence on CPU clock frequency f:

$$P_{ser} = k_{CPU} \cdot f$$

We found a good fit to experimental data with $k_{CPU} = 0{,}3125$. Bringing all of the above formulae together, we express parallel performance as a function of the number of cores N and clock frequency f:

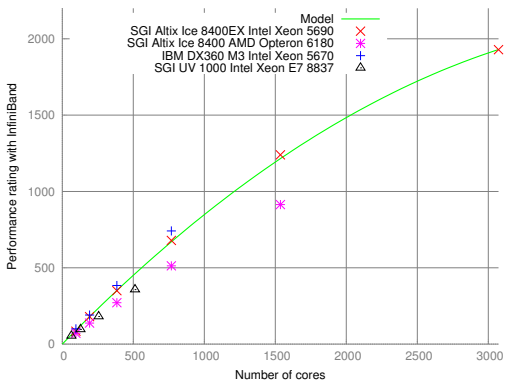


(a) Linear approximation of efficiency

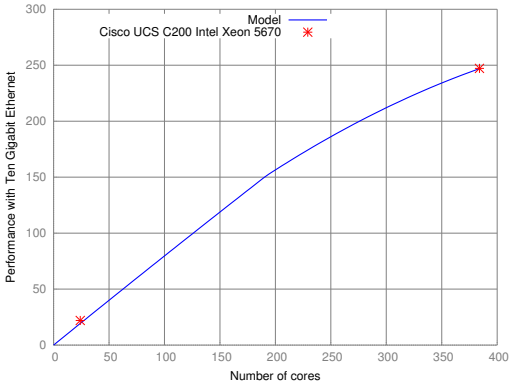


(b) Performance, calculated at $f = 3{,}47$ GHz

Figure 9.3.: Performance model for "ANSYS Fluent"



(a) Performance with InfiniBand (Experimental data normalised to $f = 3,47$ GHz)



(b) Performance with Ten Gigabit Ethernet

Figure 9.4.: Performance model for "ANSYS Fluent" with experimental data

$$P_{par}(N, f) = P_{ser} \cdot S(N) = k_{CPU} \cdot f \cdot S(N) = k_{CPU} \cdot f \cdot N \cdot (k_e \cdot N + b_e)$$

As a result, when clock frequency f is fixed, parallel performance is approximated as a quadratic polynomial of N. Corresponding graphs of $P(N)$, for a fixed value of $f = 3{,}47$ GHz, are depicted on Fig. 9.3b.

Figure 9.4 plots these performance graphs separately and in more detail, overlapped with experimental data. In case of InfiniBand (Fig. 9.4a), measurements were performed on computers with different CPU clock frequencies. Therefore, to plot all data on a single graph, we normalised measured data to clock frequency of $f = 3,47$ GHz that was used as a base point. The graph contains data points for three computers with InfiniBand network, and one computer ("SGI UV 1000") with a proprietary NumaLink interconnection network, which is shown here for reference.

In case of Ten Gigabit Ethernet (Fig. 9.4b), there was only one machine (and only two

measurements), so no scaling was required. We consider model's agreement with experimental data not ideal but satisfactory for our purposes. Raw benchmark data can be found in Appendix A.

9.5. Accessing Models via Internet

In our automated design framework, performance models are just one type of "pluggable" modules used to provide required functionality for the CAD system. Modules are queried over network. This approach allows software vendors to publish performance models for their software on their own websites as "black boxes", maintaining control over the models and updating them as new data becomes available.

We implemented the aforementioned performance model for "ANSYS Fluent" in a CGI application [88]. The application allows easily predicting performance if the number of cores in a compute cluster is known. The application also incorporates the inverse performance modelling algorithm (Algorithm 1), thereby allowing users to determine the number of cores if they know the performance that the cluster must be able to achieve. In both cases, CPU clock frequency and type of interconnection network (InfiniBand or Ten Gigabit Ethernet, or both) must be specified.

Queries to the application can be performed via web interface, resulting in human-readable output, or in a completely automated fashion, supplying input parameters in key-value pairs and receiving output in a similar fashion. An example query for direct performance modelling is presented below:

```
cores=1024
benchmark=truck_111m
network_tech=10gbe,infiniband-4x-qdr
cpu_frequency=3,47
```

The application returns the following reply:

```
cores=1024
benchmark=truck_111m
network_tech=Infiniband-4X-QDR
software=ANSYS FLUENT 13.0.0
perf_model_id=Demo model with linear approximation of
  efficiency, March 2012
performance_throughput_mode=False
performance=870,5
time_to_solution=99,3
max_rating=1943,7
max_rating_at_cores=3072
```

Let us inspect the output.

1. The value of `network_tech` is parsed, and the application automatically chooses the best type of network – in this case, InfiniBand – which is returned;

2. The identification string `software` specifies the software whose performance is being modelled, while `perf_model_id` specifies the model ID, as there can be several models of differing complexity and accuracy for the same software;

3. The application returns a value of "False" for `performance_throughput_mode`, which indicates that throughput mode (see section 9.2) is not required, because the number of parallel computing blocks specified by the user, in this case, $N = 1024$ cores, is less than the maximal reasonable number of blocks, N_{max}. If the user requested to calculate performance for the number of cores higher than N_{max}, the application would calculate it according to the principles of lowest time to solution (see Scenario A in Example 9.1);

4. The next string, `performance`, is the most important output, as it returns the value of performance rating, in tasks per day;

5. `time_to_solution` returns time required to run one task. It is reciprocal to `performance`, but for convenience it is expressed in seconds, not in days. In this case, the computer will complete a new task each 99,3 seconds. As N increases, performance also increases, and time to solution decreases. When N approaches N_{max}, performance plateaus, and time to solution doesn't decrease any more;

6. The last two strings, `max_rating` and `max_rating_at_cores`, are for informational purposes. They specify the highest possible performance rating and the number of parallel blocks N_{max} at which it is attained. This gives insight as to whether the original query was close to performance scalability limit, or still far from it. In our case, there is a reserve to increase the number of cores three-fold, up to $N_{max} = 3072$, and performance will increase by a factor of 2,23.

Part III.

Automated Design of Computer Clusters

10. Graph Representation of Configurations

10.1. Undirected vs. Directed Graphs

Throughout the years, several schemes were proposed to describe configurations of technical systems. Approaches include morphological analysis by Fritz Zwicky and using And-or trees to represent alternatives in design decisions.

In 2004, Bozhko and Tolparov [14] suggested to represent configurations of arbitrary technical systems using multipartite graphs. Their approach is able to account for compatibility between components (or functions) of the system and allows for a clear visual representation.

The first step is to define a tree of components or functions of the system which can be implemented in multiple, mutually-exclusive ways. Figure 10.1 presents an example. Electric torch has two physical components (light and power sources) and one function (portability), each of which that can be implemented in several different ways.

The next step is to construct a multipartite graph, where each partition corresponds to a component or function, and vertices in that partition correspond to possible implementations. Finally, the edges are drawn to indicate compatibility of components (see Fig. 10.2).

Components or functions not connected with an edge are incompatible. In this example, we represented than incandescent lamp (1) is only compatible with lead battery (4), but not with other power sources. Conversely, LEDs (2) are not compatible with lead batteries (4), hence there is no edge between the two. All power sources are compatible with hand-held implementation (6), while only alkaline battery (3) is compatible with head-wearable implementation (7). Finally, all portability implementations (6 and 7) are compatible with all light sources (1 and 2).

If there are more components or functions in the technical system, more partitions are added, and vertices in them are connected to all existing partitions according to compatibility. Configurations of the technical system are given by complete subgraphs (cliques)

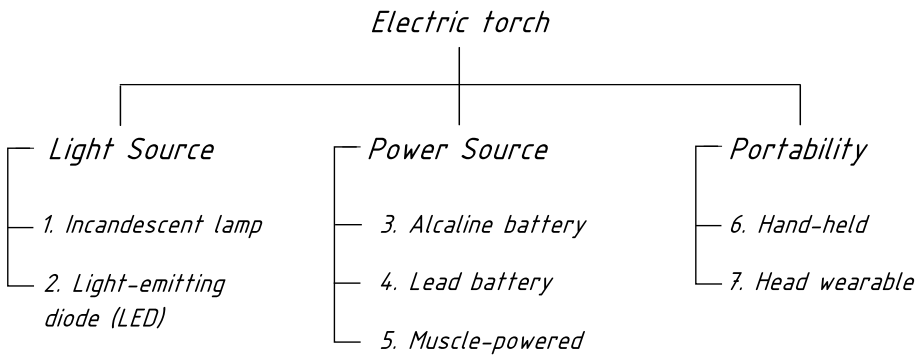


Figure 10.1.: Functional structure for electric torch.

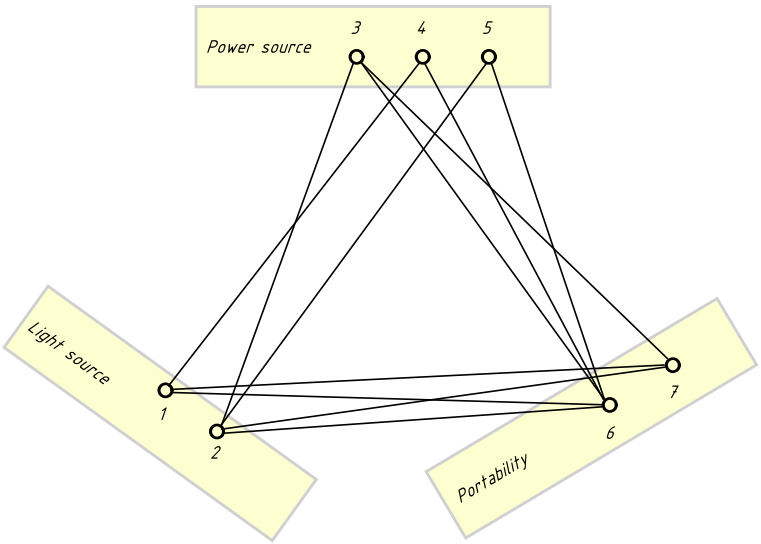


Figure 10.2.: Graph representation for electric torch, using undirected graphs.

of the graph with the size of the clique equal to the number of partitions. For example, in Fig. 10.2, vertices {1, 4, 6} form a clique (with three partitions, it is a triangle), therefore this set of components represents a valid configuration. Other configurations are {2, 3, 6}, {2, 3, 7} and {2, 5, 6}. However, Bozhko and Tolparov note that finding all such complete subgraphs in a given graph has exponential complexity.

They further note that in real life situations most components are compatible, and partitions often form bicliques – fully-connected bipartite subgraphs (see, for example, a biclique formed by partitions “Light source” and “Portability” on Fig. 10.2). Therefore real life graphs are very dense, which allows to save machine memory by storing complementary graphs.

Based on this idea of undirected graphs with cycles, and relying on the fact that in real life systems most components are compatible, we propose to use directed acyclic graphs. They lack a similar expressive power because, due to the absence of cycles, partitions are laid out in a linear way, with a clear “start” and “end”, and hence representing incompatibilities between components in two partitions is only possible when these two partitions are adjacent. Incompatibility is denoted in the same way as with undirected graphs: that is, by removing edges between incompatible components from a biclique formed by two partitions.

However, we find that such directed acyclic graphs are suitable for our purpose. They also have a benefit of clear visual representation that makes them easier to construct and understand than undirected graphs with cycles. The traversal procedure is based on depth-first search. Each path in the graph, from “Start” to “End”, represents a single valid configuration of a technical system.

If an undirected graph contains only two partitions, or if it contains three partitions, with two of them forming a biclique, then such a graph can be converted to a directed graph, without loosing relationships of compatibility. With a tripartite graph, a biclique between two partitions is broken, and one of the partitions is arbitrarily connected to the

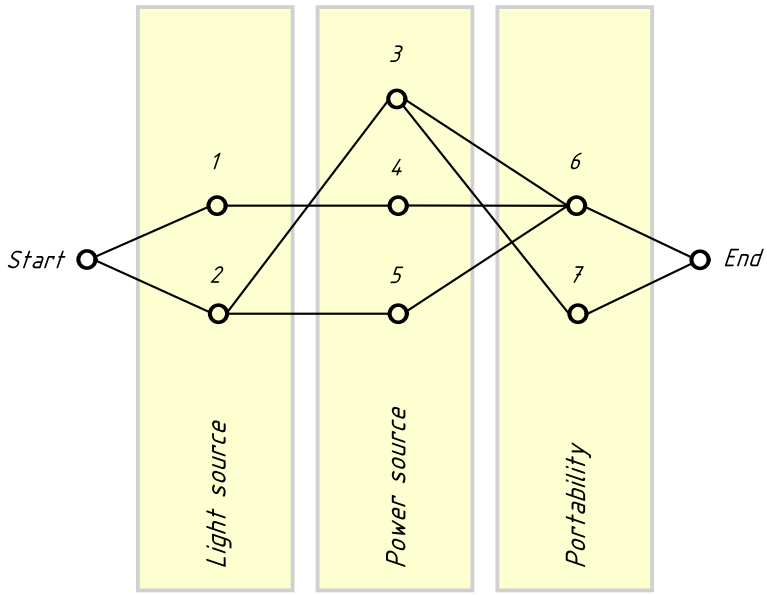


Figure 10.3.: Graph representation for electric torch, using directed acyclic graphs (arrows not shown).

"Start" vertex, and the other one to the "End" vertex.

Figure 10.3 shows the above undirected graph converted into the directed form. Arrows on edges are not shown as they uniformly point from "Start" to "End". Traversing all possible paths in this graph generates the same four configurations as the undirected version, thereby conserving compatibility relationships.

10.2. Expression Evaluation During Graph Traversal

A major advancement comes from assigning *expressions* to graph's vertices and, optionally, edges. These expressions can be evaluated in a very natural way when traversing the graph. The following context-free grammar for expressions in extended Backus-Naur form is proposed:

```
expression = string, '=', '"', ( complex_expression |
      | update_expression ), '"' ;

complex_expression = ( arithmetic_expression |
      parenthesised_expression ) { sign,
      ( arithmetic_expression | parenthesised_expression ) } ;

update_expression =
      sign, complex_expression ;

arithmetic_expression =
      simple_expression { sign, simple_expression } ;
```

```
simple_expression = string | number ;

parenthesised_expression = '(' arithmetic_expression ')' ;

number = digit {digit} ;
digit = '0' | '1' | '2' | '3' | '4' | '5' |
      '6' | '7' | '8' | '9' ;
string = char {char} ;
char = 'a' | 'b' | ... 'z' ;
sign = '+' | '-' | '*' | '/' ;
```

Below are examples of valid expressions that can be assigned to graph vertices:

```
node_cost="6000"
node_cost="+200"
ups_cost_per_kw="ups_cost / (ups_power_rating / 1000)"
```

If the right-hand side of the expression starts with an arithmetic sign, it is considered an *update expression*, and is evaluated as if the left-hand side was specified before the sign. That is, the expression `node_cost="+200"` is evaluated as `node_cost="node_cost+200"`, thereby updating the value of `node_cost`.

A non-trivial example of representing a configuration of a compute node with a graph is given in Figure 10.4. Rectangular blocks with vertices represent graph partitions, and labels on the left are partition names. Arrows highlight one of the paths in the graph. Expressions on the right of the graph are those assigned to the vertices along the path.

The compute node is based on "Hewlett-Packard" BL465c G7 server. The first partition sets some basic characteristics of a compute node, including its cost that will later be updated. Then, the graph diverges into two branches, because this server can use CPUs of two different series, AMD 6100 and 6200, each compatible with its own memory modules. In the figure, CPU partitions are shown to contain only three vertices for the reason of simplicity, whereas the actual server supports ten AMD 6100 series CPU models and eight AMD 6200 series CPU models.

Expressions assigned to CPU vertices specify important CPU characteristics such as the number of cores, CPU clock frequency and the peak number of floating-point operations performed per cycle; all of these are later used to calculate peak performance of the compute node. Also present here are expressions that update compute node cost and power consumption.

After the "CPU1" partition that defines the first CPU in the server, graph traversal can proceed in two ways. If a second CPU is to be installed in the server, edges proceed to partition "CPU2", and in this case the expressions are evaluated one more time, which leads to further update of node's cost and power consumption. It is essential that the second CPU, if installed, must match the first one, therefore partitions "CPU1" and "CPU2" do not form a biclique.

Alternatively, to represent the configuration where the second CPU is not installed, edges from partition "CPU1" follow to two auxiliary vertices that don't have any expressions associated with them.

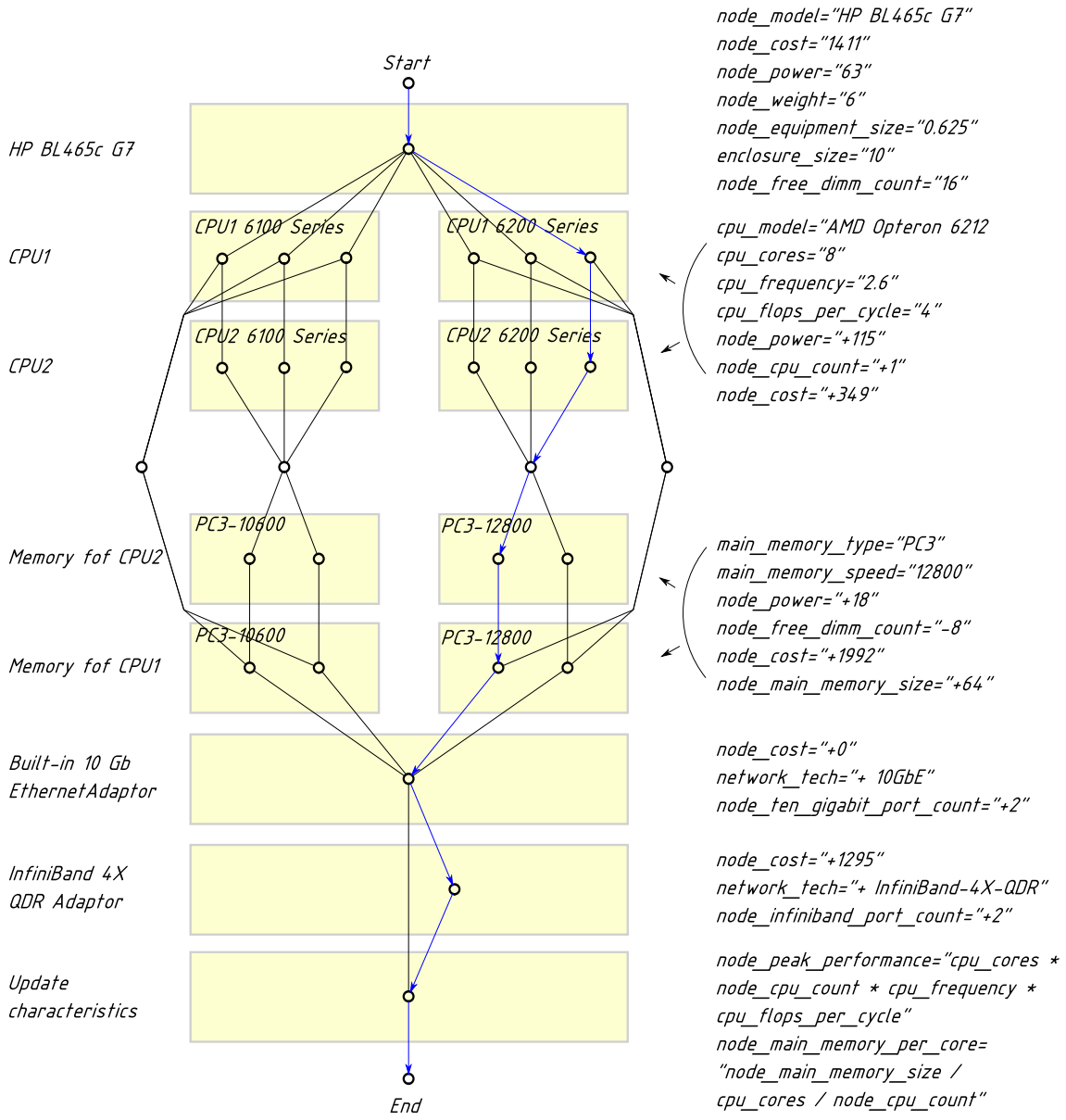


Figure 10.4.: Graph representation for the compute node configuration using a directed acyclic graph.

From partition "CPU2", edges go to partition "Memory for CPU2". In this case, all components in both partitions are compatible (for example, all AMD 6200 series CPUs are compatible with all PC3-12800 memory modules), therefore a biclique could be established, but we chose to introduce an auxiliary vertex between the two partitions: it reduces visual clutter, at the same time having no effect on graph traversal.

We earlier noted the limitation of describing configurations with directed acyclic graphs, in that compatibility relations can be represented only between adjacent partitions (otherwise cycles would appear in the graph). In this case, "CPU1" is adjacent to "CPU2", which is further adjacent to "Memory for CPU2". This way, partitions "CPU1" and "Memory for CPU1" are *not* adjacent, and we cannot directly represent compatibility between the two. This limitation can be circumvented by further branching the graph, just as it was done with branching into two different CPU series, AMD 6100 and 6200.

In this particular case there is no need to branch the graph; we simply require that memory configuration of CPU2 matches that of CPU1 (see edges between partitions "Memory for CPU2" and "Memory for CPU1"). Expressions assigned to these partitions set values for memory characteristics, and also update node's cost and power consumption.

Of particular interest is the expression `node_free_dimm_count="-8"`, which updates the value set earlier in the first partition of the graph. It is used to track the number of available DIMM memory slots on the motherboard, preventing invalid configurations where more memory modules are configured than a motherboard can hold.

The next partition in the graph sets characteristics pertaining to the built-in 10Gbit Ethernet adaptor. As the adaptor is built-in, it doesn't influence the cost of the node (the expression `node_cost="+0"` is given for clarity and can be omitted). Another network adaptor, InfiniBand 4X QDR, is optional, which is indicated by an edge that bypasses it, on the left of the corresponding vertex. However, the highlighted path in the figure does include this vertex, which leads to the update of the value of node cost.

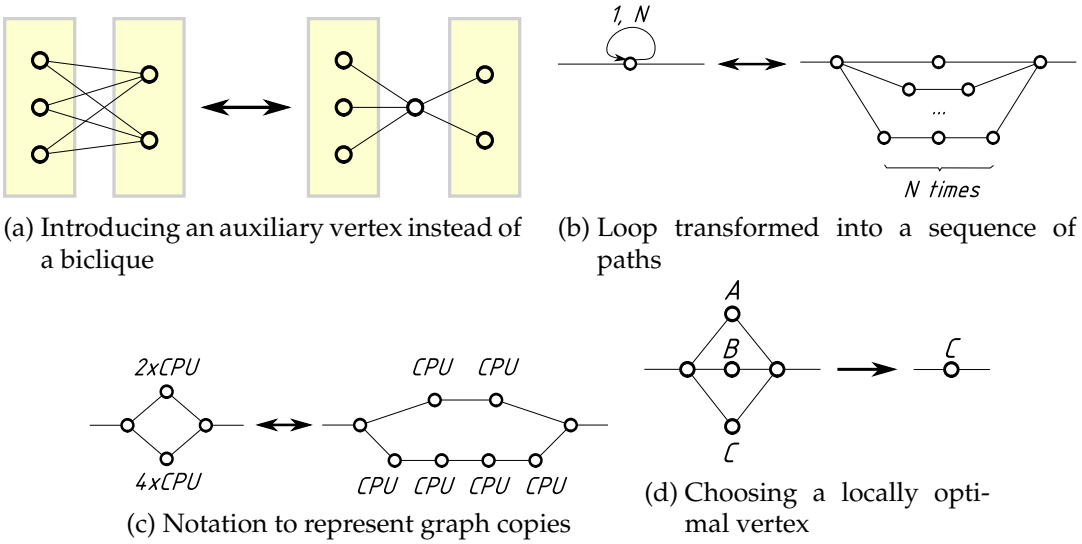
The last partition serves the purpose of updating characteristics of the compute node that can later be used as design constraints, or as a part of the objective function. Here, the node's peak floating-point performance is calculated by multiplying four characteristics whose values have already been set earlier. Another characteristic is the amount of operating memory per CPU core. This can be later used as a constraint to quickly filter out configurations that don't have enough memory to run applications. The constraint can be specified as: `min_node_main_memory_per_core=2`, where "2" is the minimal amount of RAM, in GBytes, that a compute node should have.

10.3. Graph Transformations

We propose a number of graph transformations that simplify visual representation of the graph. One of them was already mentioned: when two partitions form a biclique, an auxiliary vertex can be introduced to replace the multitude of edges. It doesn't affect graph traversal, but can reduce clutter for a better understanding (see Figure 10.5a).

If there is a loop associated with a vertex, and the corresponding expression is of the form "M, N", this is supposed to denote a sequence of paths, containing $M, M+1, \ldots, N$ vertices (see Figure 10.5b).

Similarly, when a vertex's label is preceded by an expression of the form $N\times$, where N



(a) Introducing an auxiliary vertex instead of a biclique

(b) Loop transformed into a sequence of paths

(c) Notation to represent graph copies

(d) Choosing a locally optimal vertex

Figure 10.5.: Graph operations

is a natural number, this is supposed to be transformed into a path of N identical vertices. For example, if a server can be configured with two or four CPUs, there are two identical ways to represent this with a graph: see Figure 10.5c.

The last operation concerns not a visual representation but rather a modification of a graph. If, out of some assumption, one of the vertices in a group is known to represent a locally optimal component, the whole group can be substituted with this vertex (see Figure 10.5d). This can be used to substantially reduce the number of paths, and therefore the number of configurations that the graph generates. However, there are two notes regarding this operation. First, it is obviously irreversible, which is indicated by a one-way arrow. Second, as usual with local optimisations, globally optimal solutions can be discarded.

A useful example of such local optimisation is the selection of an optimal memory configuration of a compute node. For example, 32 GBytes of memory can be configured in three ways: 8x4 GB, 4x8 GB and 2x16GB. One of these configurations can be chosen based, for instance, on cost, effectively lowering the total number of configurations of a compute node by a factor of three. In this particular case, using two 16 GB memory modules might be ruled out due to their high price, and one of the remaining options will be preferred. However, it can later be found that low-density memory modules occupy all DIMM slots on the motherboard, making memory extension impossible – a consequence of local optimisation.

10.4. Defining Software Configurations with Graphs

Configurations of supercomputer software can be represented using the same graph-based approach. We can specify dependencies between operating systems, libraries and application software packages. For example, one application software can rely on an open-source MPI library, while the other may require a third-party non-free library, and yet another application software package may already include all required libraries.

If all hardware configurations are compatible with all software configurations, then graphs for both can be simply joined in a chain. In more complex cases, such as software requiring a certain type of hardware, the hardware graph will be branched as necessary, and corresponding software graphs will be connected to the branches.

During the traversal of this augmented graph, characteristics such as cost will be determined for each software configuration. Moreover, performance modelling can then be used to calculate performance of each "hardware plus software" combination. For example, the use of free and non-free libraries can result in different performance and cost figures, and the analysis will allow to make informed decisions.

10.5. XML Syntax for Graph Representation

We propose to store graph information in XML files. This includes (1) description of individual hardware items using vertices, (2) commands to manipulate vertices, group them into partitions and manage connections between them. We define the following context-free grammar for commands in XML graph definition files (`string` and `expression` are defined as above in section 10.2). The notation is relatively simple, and does not allow to automatically represent loops (Fig. 10.5b) or sequences of identical vertices (Fig. 10.5c).

```
command = item | place | edge | connect | include ;

item = 'item', { string, '=', expression }, ItemName ;

place = 'place', 'to-partition=' PartitionName ;

edge = 'edge', 'from=' ItemName,
      [ 'from-partition=' PartitionName ],
      [ 'to=' ItemName ],
      [ 'to-partition=' PartitionName ] ;

connect = 'connect',
      ( 'from=' ItemName | 'from-partition=' PartitionName ),
      ( 'to=' ItemName | 'to-partition=' PartitionName) ;

include = 'include', FileName ;

ItemName = string ;
PartitionName = string ;
FileName = string;
```

There are five commands; the fundamental one is `item`, which defines a hardware item represented by a vertex in a graph. An arbitrary number of item characteristics can be supplied via `key=value` pairs, where `key` is a string, and `value` is an expression in the sense defined in section 10.2 – an arithmetic expression evaluated during graph traversal, possibly involving characteristics of other items.

Once a vertex has been defined, it can be reused multiple times by placing it into the graph, using the `place` command. Copies of the same vertex are differentiated by the partition to which they belong, specified as a required argument to `to-partition=`. Partitions are abstract entities, acting simply as vertex labels, and are helpful for visual representation of the graph. Therefore, alongside with the original vertex that doesn't belong to any partition, there can be multiple copies of it, each belonging to different partitions.

`edge` is a command used to create connections from one vertex to another. It's required argument is the name of the source vertex, specified in `from=`. `from-partition=` parameter is optional; if specified, it means that a *copy* of the source vertex residing in the specified partition should be connected rather than the source vertex itself (the copy must have been already created using `place`).

To define edge's destination, either `to=` or `to-partition=` is used, or both. `to=` specifies the name of the target vertex. If `to-partition=` is specified, it first creates a copy of the target vertex specified in `to=` and places it into the required destination partition, then creating an edge.

If `to=` is omitted, it is considered to be equal to the source vertex, and then `to-partition=` should be specified (otherwise a cycle in the graph would appear).

The `connect` command connects two partitions in all-to-all manner (forming a biclique between them), or connects a vertex to a partition, or a partition to a vertex. It replaces the need to call `edge` multiple times. If `from=` and `to=` are both specified, it could be interpreted as a request to create an edge between two vertices, but we rather consider it an invalid combination and advice the user to call `edge` instead.

Finally, the `include` command includes another XML file in place. To be successfully traversed, the graph requires two auxiliary vertices, "Start" and "End". Below is the example of an XML graph definition file.

```
<?xml version="1.0" encoding="UTF-8"?>
<itemslist>

  <item>Start</item>
  <item>End</item>

  <item cpu_model="AMD Opteron 6272" cpu_cores="16"
     cpu_frequency="2.1" node_cpu_count="+1">CPU</item>

  <item main_memory_type="PC3" node_main_memory_size="+32">
      Memory</item>

  <edge from="Start" to="CPU" to-partition="CPU1"></edge>

  <edge from="CPU" from-partition="CPU1"
      to-partition="CPU2"></edge>

  <edge from="CPU" from-partition="CPU1" to="Memory"></edge>
  <edge from="CPU" from-partition="CPU2" to="Memory"></edge>
```

```
  <edge from="Memory" to="End"></edge>

</itemslist>
```

The graph built according to this definition is presented in Figure 10.6. According to the commands in the XML file, first, two vertices are created, "Start" and "End". Then, vertices "CPU" and "Memory" are created, with corresponding characteristics. The first `edge` command draws an edge from "Start" to the *copy* of vertex "CPU", which is automatically created in partition "CPU1". The next `edge` statement draws an edge from this copy to another copy, now in partition "CPU2".

The following two `edge` statements draw edges from both copies to the "Memory" vertex, and the final edge goes from "Memory" to "End". The original "CPU" vertex is shown in the figure, but it is not connected to any other vertex. It is shown here for the sake of explanation; on other figures in this chapter we do not display vertices that are not part of the graph.

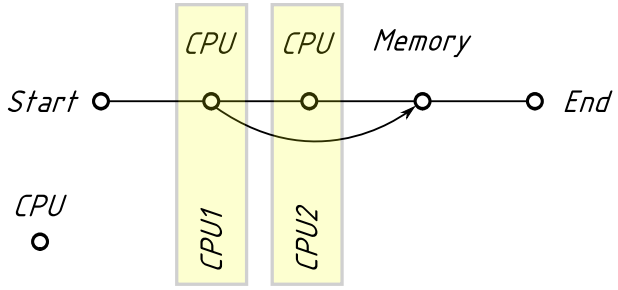


Figure 10.6.: Graph built from XML definition.

Traversing this graph generates two configurations, the first having a single CPU, and the second with two CPUs. Both configurations feature one memory block.

11. Algorithm for Automated Design of Cluster Supercomputers

In this chapter we present the main algorithm of our automated design framework. We use single-criterion optimisation (see Chapter 6). As with any combinatorial optimisation problem, the challenges are: (a) generating valid (feasible) candidate solutions, (b) quickly calculating the value of criterion function for each candidate solution, and (c) pruning the design space to discard inefficient solutions.

11.1. Generating Candidate Solutions

The ultimate *solution* to the problem of automated design of cluster supercomputers is a *configuration* of a supercomputer – that is, the exact specification of types and quantity of blocks within the system, and connections between them. However, attempting to represent numerous subsystems of a cluster supercomputer within candidate solutions (network, power, storage and other subsystems) would lead to combinatorial explosion. Instead, we use a configuration of an elementary building block, a compute node, as a candidate solution. During design stages, each solution is augmented by details corresponding to the specific design stage.

We first considered applicability of commonly used simulated annealing and evolutionary algorithms, such as genetic algorithms. However, these classes of algorithms rely on ability to generate new candidate solutions out of existing ones: simulated annealing needs to generate neighbouring states from the current state, while the crossover procedure found in genetic algorithms uses two existing candidate solutions to generate two new solutions.

Ideally, for these types of algorithms, candidate solutions (configurations of a compute node) should be represented with a string or a vector. However, we could not find an efficient encoding scheme; the main challenges were interdependencies and compatibility relations between components.

Example 11.1 *Suppose we want to represent a configuration of a supercomputer with a vector, and part of that vector is used to represent configuration of compute nodes. There are two types of servers that can be used as compute nodes, A and B, and five types of memory modules, numbered from 1 to 5. Server A has six memory slots, and can accept memory modules of types 1, 2 or 3, whereas server B has eight memory slots, accepting modules of types 3, 4 or 5.*

To encode memory configuration of a compute node, we use a vector with three elements. The first is the type of server (A or B), the second is the number of slots occupied by memory modules, and the third is the type of memory module (a number from 1 to 5). Two sample valid vectors are $(A, 4, 1)$ *and* $(B, 8, 5)$.

Considering these vectors as chromosomes and performing a crossover procedure (with a crossover point after the first gene) would lead to two new vectors: $(A, 8, 5)$*, which is invalid, because server* A *has only six memory slots, and doesn't support memory type* 5*, and* $(B, 4, 1)$*, also invalid, because server* B *doesn't support memory type* 1*.*

As a result, representing candidate solutions in a naïve way leads to generation of invalid solutions. In addition to simulated annealing and genetic algorithms, we also considered evolutionary programming which allows more freedom in representing candidate solutions (such representation doesn't have to be a string or a vector), but concluded that mutation operation inherent to this method would again lead to generation of invalid candidate solutions.

Instead of generating invalid configurations and backtracking, we chose to apply rules to directly generate valid configurations. We use directed acyclic graphs to represent compatibility between components, and traversing these graphs immediately yields a list of valid configurations (see Chapter 10).

11.2. Applying Constraints and Using Heuristics

With the graph-based representation of configurations, each candidate solution is a set of `key=value` pairs. Traversing the configuration graph evaluates expressions assigned to its vertices (see Section 10.2). This allows to apply user-specified constraints to the list of candidate solutions.

Additionally, expressions can be used to evaluate arbitrary metrics of each candidate solution. For example, in our framework we use the criterion function which is the ratio of total cost of ownership of a supercomputer to its performance on a given task (see Chapter 6). When a candidate solution – a configuration of a compute node – has been generated, we can calculate the value of the following metric: the ratio of cost of the compute node to its peak floating-point performance. This metric is in a certain sense an approximation of the criterion function obtained with less information, a "predictor" of its future value; it is then used as a heuristic to filter out candidate solutions which are unlikely to be optimal.

Both techniques – constraints and heuristics – allow to quickly prune large sections of the design space; see Chapter 8 for more details.

11.3. Algorithm

The proposed algorithm for automated design of cluster supercomputers operates on the pool of candidate solutions (configurations of a compute node) that remain after filtering out inviable configurations using constraints and heuristics. Each of the numerous design stages augments the candidate solutions with details corresponding to the stage. The flowchart of the proposed algorithm is outlined in Figure 11.1 (some dashed arrows that update metrics of candidate solutions are not shown to reduce clutter).

The design process starts with turning informal requirements into a formal specification, which requires human intervention. Knowledge of existing infrastructure can also be incorporated at this stage in the form of compatibility requirements. The outcome of the process is the list of global requirements and constraints that all future solutions must

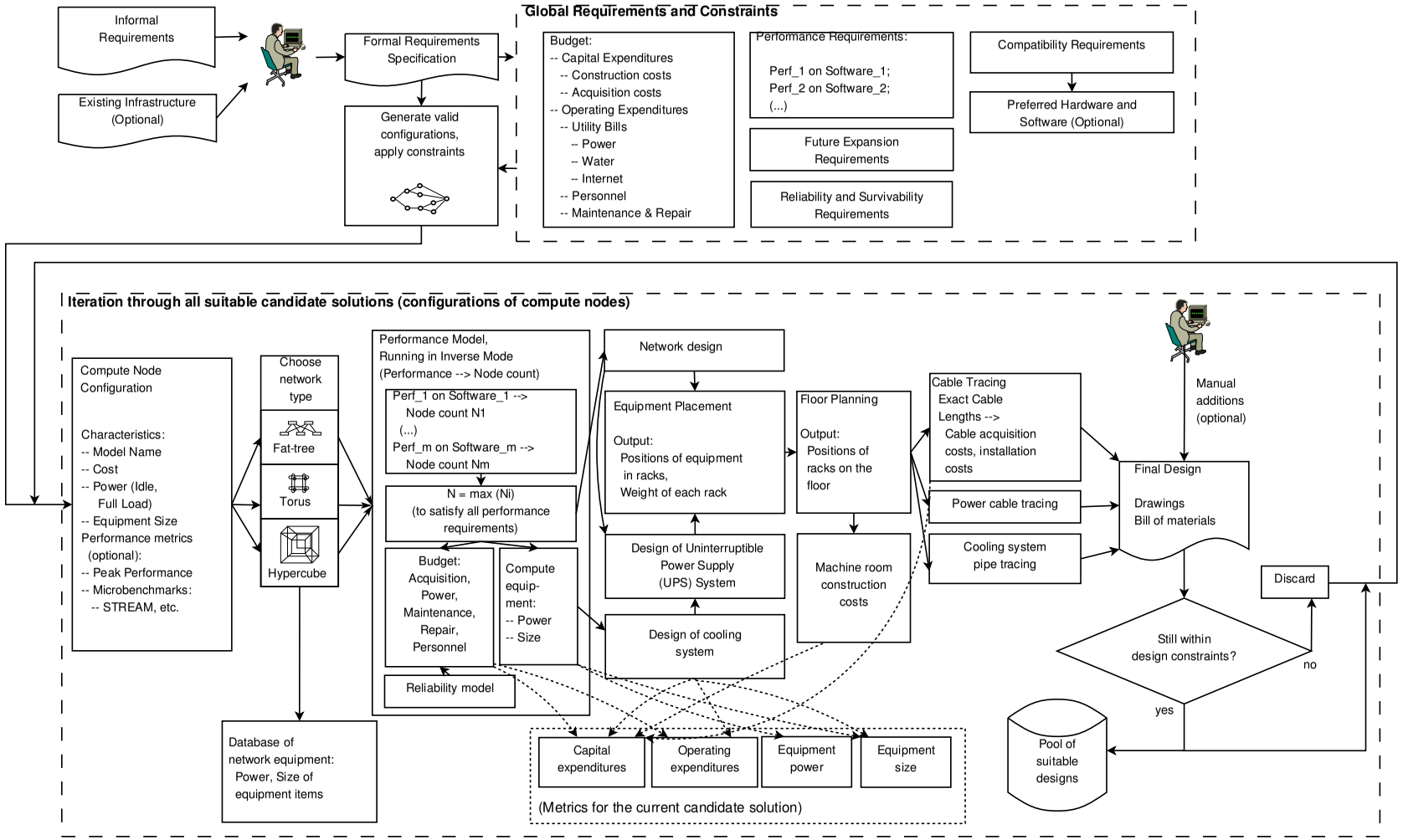


Figure 11.1.: Flowchart of the algorithm for automated design of cluster supercomputers.

obey. Such requirements and constraints can be of various nature and have a necessary level of detail. Performance constraints (lower limits on performance on a set of tasks) are the only required. Additionally, there could be budgetary constraints (upper limits on allowed capital and operating expenditures of the supercomputer), requirements on future expansion of the supercomputer, requirements on reliability (important for all types of installations) and survivability (useful mainly for emergency and military applications), and so on.

After the list of constraints has been built, a list of compute node configurations is generated from the configuration graph, and constraints are applied to filter out unsuitable configurations. At this stage, candidate solutions contain information on technical and economic characteristics of compute nodes only, rather than characteristics of subsystems of the cluster computer or the machine as a whole. Therefore constraints that can be applied at this stage are essentially "node-level" constraints: number of CPUs or cores in a node, power consumption of a node, its peak floating-point performance, etc. It is also at this stage that a heuristic can be used to filter out unpromising configurations.

When decimation of the configuration list is complete, the main part of design procedure starts. Compute node configurations are now considered candidate solutions. The loop iterates through every candidate solution (this can be implemented in an "embarrassingly parallel" way, see section 12.7), subjecting each of them to several design stages.

A configuration is stored as a set of `key=value` pairs. At this point, a candidate solution is still just a configuration of a compute node, therefore it holds only node-level characteristics: its model name, cost, power consumption, size, and others, as well as performance metrics: peak floating-point rate, memory bandwidth of a node, etc.

On the next stage, the user is presented with an opportunity to select the type of cluster interconnection network that will be used in further design procedures. The choice includes network topology, equipment type and additional parameters such as the maximum allowed blocking factor for the network. The choice of network and its parameters is important, because it influences both parts of the criterion function, cost and performance of the final system.

There is, in fact, a multitude of possible network choices: for example, a torus network, although cheaper than a fat-tree, can lead to unacceptable performance degradation for some algorithms. What factor is bigger – cost decrease or performance degradation – and what will be the decisive effect on the criterion function, cannot be known in advance, before performance modelling is done. Therefore, if several network choices must be inspected, each candidate solution must be cloned the required number of times, and the following design stages should operate on each copy independently (again, this can be done in parallel).

In principle, the same "cloning" approach applies to other subsystems of computer clusters that, when designed differently, influence both cost and performance. For example, design of the storage subsystem can influence both cost and performance. On the other hand, design of cooling and power supply systems has effect only on cost but not on performance (at least, in a first approximation).

Hence, if there are two types of cooling systems, and three types of power supply systems, there is no need to try six different combinations. Locally optimal decisions can be made: first, one of two cooling systems can be chosen, based on the minimal total cost of ownership, and then one of three power supply systems is selected, with the same optimal-

ity criterion. The resulting selection of subsystems is guaranteed to have the minimal total cost of ownership, and when combined with computing hardware of the cluster, yields global optimality to the criterion function, $TCO/Performance$.

The next stage of the algorithm uses an inverse performance model (see section 9.3) to determine the number of compute blocks (cores, CPUs, compute nodes, etc.) required to attain performance specified in performance requirements. If several such requirements were specified, the maximal number of blocks returned by all inverse performance models is used to satisfy all requirements: $N = \max_{i=1,m} N_i$. Performance modelling takes into account the choice of interconnection network done earlier.

As the number of compute blocks becomes known, their procurement (capital) costs can be determined and compared against global budgetary constraints. If constraints are violated, the candidate solution can be discarded immediately, because further design procedures would only raise costs further. The same applies to power consumption and equipment size of the computing hardware; they can also be checked against global constraints as early as at this stage.

Reliability model can be plugged in here to estimate maintenance and repair costs for computing equipment, updating operating expenditures of the candidate solution.

Based on the number of compute blocks, an interconnection network is designed – that is, exact hardware items and connections between them are determined. This allows to calculate capital and operating costs for network equipment, its power consumption and size, and update corresponding metrics of the candidate solution. Again, the updated metrics can be compared to global constraints, possibly discarding the candidate solution before engaging with further design stages.

Based on power consumption of computing and network equipment, a cooling system is then designed. Yet again, corresponding metrics of the candidate solution are updated and checked against global constraints. Then follows a power supply system that must be appropriately sized to back up the cooling system in case of power failure (see Chapter 15). The next stage concerns placing equipment into racks and positioning racks on the floor (see Chapter 16). This results in the size of the machine room, which influences capital costs for constructing a machine room, or operating costs of renting this space. Metrics of the candidate solution are updated once again.

The next stage is to trace network and power cables, and possibly pipes for the cooling system; these are all done by similar algorithms. The result is the final design, comprising of (a) drawings specifying blocks used to build the supercomputer and connections between them, floor plans, etc., and (b) the bill of items and materials that must be procured. Manual additions can be performed upon the design. If the resulting design is still within the constraints, it is saved to the pool of suitable designs for future inspection; candidate solutions in the pool are then sorted according to the value of criterion function. Otherwise, if constraints were violated, the current candidate solution is discarded, and the loop starts its new iteration with the next candidate solution.

11.4. Optimality of Solutions

In this section, we discuss whether the synthesis procedure outlined in Chapter 1 indeed leads to optimal solutions in the mathematical interpretation of the term. We arrive to

the conclusion that optimality is guaranteed within the limits of the framework, and also discuss what exactly are those limits. In the narrow sense, optimality is guaranteed by using exhaustive search in the design space. The question is, however, how closely we model the reality.

To model structure of a compute node, we use the approach based on directed acyclic graphs (see Chapter 10) which allows to reach any required level of detail in determining procurement cost and total cost of ownership (TCO) of the machine. However, we deliberately limit accuracy of description to reduce the size of design space. For example, we found that describing a compute node based on Hewlett-Packard's "BL465c G7" server with a sufficient level of detail gives us 264 different configurations. Using a more detailed description doesn't lead to more precision in determining the total cost of ownership.

We then use performance models which also have a certain precision. Using more precise performance models would increase modelling time, but would hardly make sense, because we have seen in Chapter 9 that real life cluster computers demonstrate performance variations, sometimes on the order of 10% or more, even if they are assembled from identical hardware components and use similar software stacks. Therefore precision of performance models is limited, but this is not a problem.

We showed earlier that certain design stages, such as interconnection network design, influence both performance and cost of the solution, therefore to account for the multitude of possible network designs we must examine each of them. This is a manifestation of combinatorial explosion.

For other design stages, such as design of power supply and cooling systems, that only affect TCO but not performance, we showed that using locally optimal results of such stages is enough. Then, after completion of each design stage, we can compare metrics of the candidate solution to global constraints, and possibly discard the solution, which limits the number of candidate solutions that reach the final stage.

Within the limitations outlined above, we use exhaustive search – that is, we analyse all 264 configurations of a compute node – so we are guaranteed to find one of 264 that brings optimality to the criterion function, $TCO/Performance$. That's why the synthesis is formally optimal.

For large problems, we propose to use constraints and heuristics to reduce the design space. Using either of these approaches cuts off parts of the design space, and therefore the optimal candidate solution can accidentally be lost; searching the reduced design space then leads to sub-optimal designs. In Chapter 17 we explore in more detail the quality of sub-optimal solutions.

In a wider sense of the term "optimal", the question is whether supercomputer designs derived with a proposed algorithm are indeed competitive with existing machines, designed manually or with partial automation. In other words, could another computer, designed according to different principles, be faster, or cheaper, or both? The answer is positive.

For example, our performance models could be incomplete or imprecise, or the competing machine could be hand-tuned to provide more performance for that particular task, or the task's algorithm could be adapted to make better use of the machine. All these factors could make a competing machine faster. Emergence of new hardware components on the market could additionally make the competing machine cheaper.

In this common meaning of the word "optimal", our framework provides *good* designs.

It still has a range of useful features compared to manual designs or ad hoc practices: it automates most parts of the process, lacks bias inherent to humans, and is capable of exploring large design spaces.

11.5. Duality of Design Problems

In real life, cluster buyers usually possess a certain fixed budget, and want to maximise the performance of a computer they can buy for that budget. In this formulation, the total cost of ownership is fixed, and performance must be maximised. This formulation is dual to ours where performance is fixed, and the total cost of ownership must be minimised.

Remember that with our algorithm for each candidate solution we perform the following actions: (1) determine the number of compute blocks necessary to attain required performance; (2) perform numerous design stages. The first of these actions is done by the inverse performance model (section 9.3) and has an iterative structure: we adjust the number of compute blocks N until performance $P(N)$ is greater than or equal to the required level. At each iteration we calculate performance $P(N)$. After N is found, we perform design stages. Thereby handling each candidate solution results in several invocations of a direct performance model that calculate $P(N)$, and then a *single* set of design stages for the last iteration.

Suppose now that we use a dual formulation of the problem, where the budget is fixed and performance must be maximised. In this case we would need to iteratively increase the number of compute blocks N until the total cost of ownership $TCO(N)$ is slightly less than or almost equal to the fixed budget. The challenge here is that at each iteration, in order to calculate $TCO(N)$, we need to perform all design stages. We also need to calculate performance $P(N)$ at every iteration, to ensure that increasing N does indeed increase performance. Here, handling each candidate solution requires several invocations of a direct performance model, and *several* sets of design stages – compared to the single set of stages in the previous case.

Design stages are lengthy and require network calls to modules that are external to the CAD system. As a result, the dual formulation takes considerably longer to solve, even though it would finally arrive to the same result as our method.

We also note that an attempt to "spend all available money" (that is, trying to design based on the fixed budget) may not lead to good solutions. Indeed, a certain number of compute nodes can fulfil all performance requirements, at the same time staying within power, space and other constraints. If some money remains unspent, there could be an incentive to buy more compute nodes. However, this can lead to worse solutions: when adding more nodes, constraints can suddenly become violated, because technical and economic characteristics tend to increase in steps rather than continuously even when adding a single node.

For example, the need to buy one more rack to house new nodes leads to a stepwise increase in capital and operating expenses; similarly, buying more nodes may require another network topology with more switches, again with a stepwise increase in cost.

Even if there are no hard constraints on these metrics, cost increase resulting from buying more nodes can prevail over performance increase, and the system that spends all available money will, in fact, be less optimal in terms of $TCO/Performance$ than the

original design that just attains the requested performance.

Due to all these circumstances, we stick to our formulation that tries to minimise the TCO rather than maximise the performance.

12. CAD System for Computer Clusters

In this chapter we describe the design and functioning of the software tool – the prototype CAD system – that implements the main algorithm of our framework described in Chapter 11. The CAD system is modular software, with the main application written in Object Pascal and therefore working both in GNU/Linux and MS Windows operating systems due to its portability.

The modules are used to implement specific design tasks; they are essentially CGI scripts that are queried over the network using Hypertext Transfer Protocol (HTTP). Thereby, in principle, they can be implemented in any programming language and run on any platform. In this prototype implementation we chose to place all modules on the same computer with the main CAD application, so all communication takes place inside a single computer. The modules for "ANSYS Fluent" performance model (Chapter 9) and interconnection network design (Chapters 13 and 14) are implemented in Object Pascal, while the module to design UPS systems (section 15.9) is implemented in the Python language.

In the prototype implementation, modules are run by the Python-based web server which is available in the standard Python installation. As a result, the entire software suite can be downloaded from the Internet and run on any GNU/Linux or MS Windows computer without installation, with the only requirement that the Python environment version 3.3 or higher is already installed.

The architecture of the CAD system was presented three times at the International Supercomputing Conference (ISC). In particular, in 2011 and 2012, research posters [90, 91] were presented that formulated, respectively, the goals for the CAD system and the modules comprising it. In the follow-up work, a research paper [95] was presented in 2014 that proposed a Python-based domain-specific language (DSL) called "SADDLE", used to create scripts that automate design procedures.

Since the CAD system was put online in November of 2012, it has been downloaded over 100 times. The latest release features source code for all components, facilitating adaptation of the tools by third parties to their own needs.

12.1. Overall Structure of the Main CAD Application

The application has the means to load a configuration graph (Chapter 10) stored in XML format and generate configurations from it. Internally, each configuration is represented as a list of `key=value` pairs that store characteristics of the configuration. During the course of design procedures, the list is augmented by more characteristics as they get evaluated. To reduce the size of design space, constraints can be imposed on the list of configurations. Node-level constraints only affect characteristics of separate compute nodes, and are usually imposed immediately after loading the configuration graph. Global constraints put restrictions on the machine as a whole; they are checked after each design stage and pre-

vent offending configurations from participating in further design stages.

Both types of constraints mark configurations as "disabled", effectively removing them from consideration by the CAD tool. Such configurations bear a tag explaining the reason for disabling. They can be automatically or manually removed from the list, or can be left for further inspection. Another technique to quickly weed out potentially unsuitable configurations is to apply a heuristic. See Chapter 8 for an overview of methods to deal with combinatorial explosion.

The main window of the prototype CAD tool is presented in Figure 12.1. The toolbar allows to load a configuration graph database in XML format (see toolbar button `"Load XML DB"`). After the graph is loaded, configurations are generated and displayed in the lower pane in a comma-separated values (CSV) form. They can be saved to a file (see toolbar button `"Export to CSV"`), which can later be read back (see toolbar button `"Import from CSV"`).

A pane `"Node-level constraints"` allows to specify constraints that act only on technical and economic characteristics of compute nodes, rather than an entire machine. In the figure, currently specified constraints are: `min_cpu_cores=12` (requires that compute nodes have at least 12 CPU cores) and `min_node_main_memory_per_core=4` (requires that a compute node has at least 4 GBytes of main memory per each CPU core). The first constraint, for example, will allow a compute node with two 6-core CPUs, or two 8-core CPUs, or one or two 12-core CPUs, because all of these configurations provide 12 cores or more. However, it will disallow a compute node with one 6-core or 8-core CPU.

A comment symbol # prepends a constraint that is to be ignored by the CAD tool. To apply constraints, the `"Impose node-level constraints"` button is used. If some configurations are disabled during this procedure, it is reflected in the status bar in the

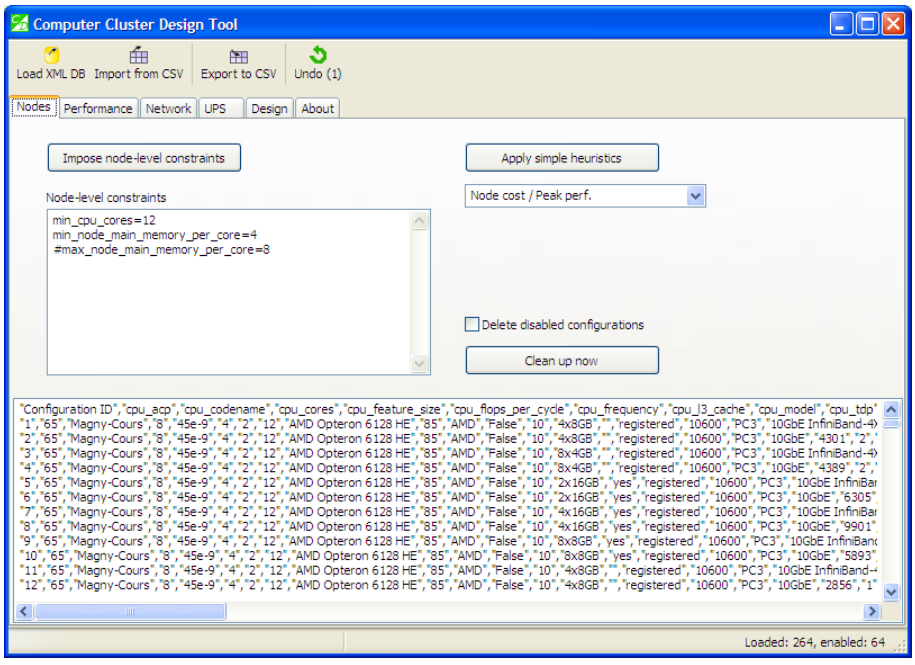


Figure 12.1.: Main window of the prototype CAD tool.

lower right corner of the window. In our example, only 64 configurations remain enabled out of 264 originally loaded.

Another way to weed out potentially unpromising configurations is to apply a heuristic. Currently implemented heuristic is "`Node cost / Peak performance`" (see Chapter 8 for more details). Both actions – node-level constraints and heuristics – can severely reduce the number of enabled configurations. If the user feels no need to examine which configurations were disabled and why, then disabled configurations can be removed from the list by pressing the "`Clean up now`" button. Alternatively, a check box "`Delete disabled configurations`" can be ticked to automatically clean up the list whenever some configurations become disabled during application of node-level constraints or heuristics.

The "`Undo`" button on the toolbar allows to revert the results of any action that resulted in modification of the configuration list: application of constraints or a heuristic, or a clean-up action.

12.2. "Performance" Tab

The "`Performance`" tab of the prototype CAD tool allows to select the performance model to be used during the design procedure (see Figure 12.2). The button "`Load`" is used to load the list of models by querying the specified URL. Upon pressing the button, the list of available models appears in the lower pane of the window.

Currently two performance models are implemented. The first, "`Peak performance`", refers to the peak floating-point performance of the cluster computer, and is "built-in" – that is, peak performance can be calculated directly from characteristics of compute nodes,

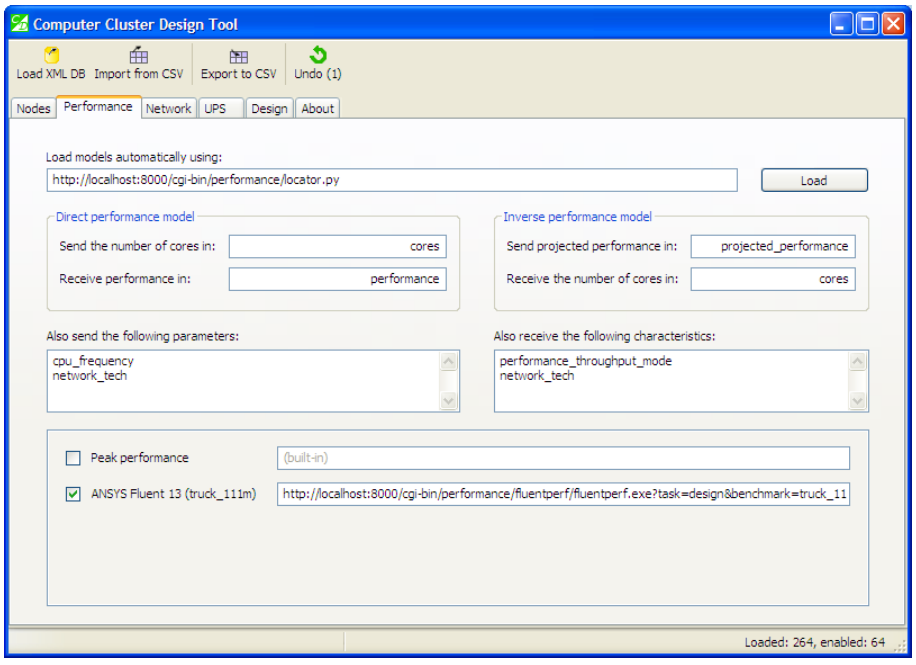


Figure 12.2.: "Performance" tab of the prototype CAD tool.

without querying any external module. The peak performance is given by the following expression: number of floating-point operations per cycle, times CPU clock frequency, times number of cores in a CPU, times number of CPUs in a compute node, times number of compute nodes in a cluster.

The second performance model is for the "ANSYS Fluent" computational fluid dynamics (CFD) software suite (see section 9.4), and in particular it's `"truck_111m"` benchmark. The URL to query the performance model is provided. Remember from Chapter 9 that there are two types of performance models, direct and inverse. A direct performance model receives the number of compute blocks and returns a performance estimate. In our case, the value of a characteristic `"cores"` from each configuration is sent, and the direct model's output is saved in a characteristic called `"performance"`.

Conversely, querying an inverse performance model sends the value of a characteristic `"projected_performance"` and receives the number of compute blocks required to attain this performance into the characteristic `"cores"`.

To correctly estimate performance, two more characteristics must be sent, `"cpu_frequency"` (CPU clock frequency in GHz) and `"network_tech"` (network technology of a compute cluster, specified as a list of technologies available for each compute node configuration, for example: `"10GbE InfiniBand-4X-QDR"`). If more than one network technology was sent in `"network_tech"`, then the performance model chooses the best one (producing the best performance) – in this case, `"InfiniBand-4X-QDR"` – and returns it back in `"network_tech"`.

Another characteristic that is received is `"performance_throughput_mode"`. Here, the performance model can specify a Boolean value indicating whether the number of compute blocks requested by the user was higher than the optimal, and "throughput mode" was required (see section 9.2 on the explanation of "throughput mode"). For a sample query to a performance model, see section 9.5.

12.3. "Network" Tab

The "Network" tab of the prototype CAD tool (see Figure 12.3) serves to load the list of network topologies and select one of them that will be used to design an interconnection network of a cluster supercomputer. The network design module developed in this thesis can design fat-tree and torus networks (see Chapters 13 and 14, respectively).

The essential characteristic that must be sent to the network design module is the number of compute nodes (`"nodes"`), while network cost is received and saved in `"network_cost"`. A number of additional characteristics is received and stored for future use, including important technical characteristics such as rack size occupied by the network equipment (`"network_equipment_size"`), power consumption (`"network_power"`) and weight (`"network_weight"`) of equipment.

In Figure 12.3, a non-blocking fat-tree topology is selected. Other topologies (blocking fat-trees or torus) differ in parameters passed when querying the network design module via Internet.

12.4. “UPS” Tab

The “UPS” tab of the prototype CAD tool (Figure 12.4) allows to load the list of uninterruptible power supply (UPS) design modules and select one of them to be used for design procedures. The UPS design module works according to the algorithm outlined in section 15.9. Currently, one model of UPS is available, a 45 kW “Liebert APM”.

The essential characteristic that is sent to the UPS design module is a combined power consumption of compute nodes and network equipment (`"power"`). Additionally, the minimum backup time is specified in `"min_ups_backup_time"`. In response, the module returns cost of the UPS system (`"ups_cost"`) as well as a number of other characteristics of the UPS, such as its backup time (`"ups_backup_time"`), size in racks, and weight.

12.5. “Design” Tab

“Design” tab of the prototype CAD tool is used for launching the design process and inspecting the results (see Figure 12.5). Global design constraints are specified in the designated pane. In contrast to node-level constraints from the “Nodes” tab that put restrictions on characteristics of compute nodes, global constraints pertain to characteristics of the entire machine.

The only required constraint is the lower bound on performance that must be obtained: `"min_performance=240"`. Other constraints are optional. In our example, constraints on capital expenditures (`"max_capex=200000"`) and power consumption of the cluster (`"max_power=10000"`) were specified; this allows to further reduce the number of con-

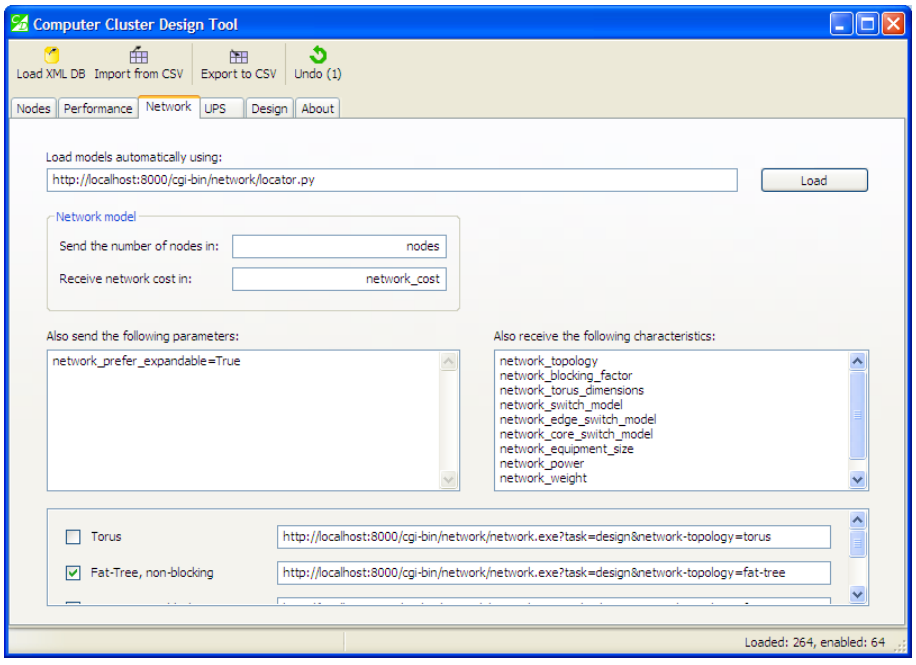


Figure 12.3.: “Network” tab of the prototype CAD tool.

figurations to be manually inspected by the user. In principle, any types of constraints can be specified here, as long as corresponding metrics are available in the configuration graph or can be calculated by design modules.

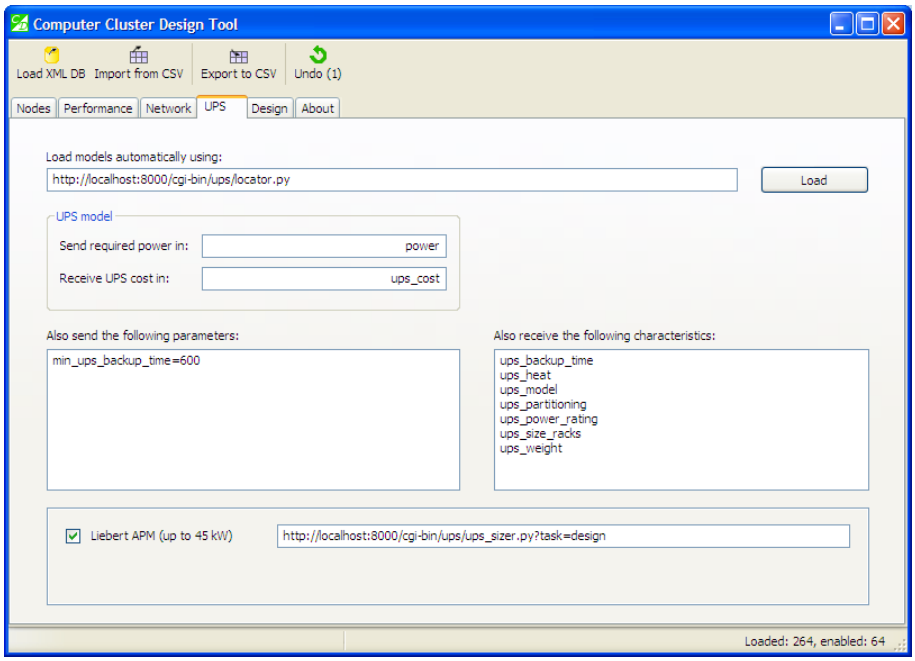


Figure 12.4.: "UPS" tab of the prototype CAD tool.

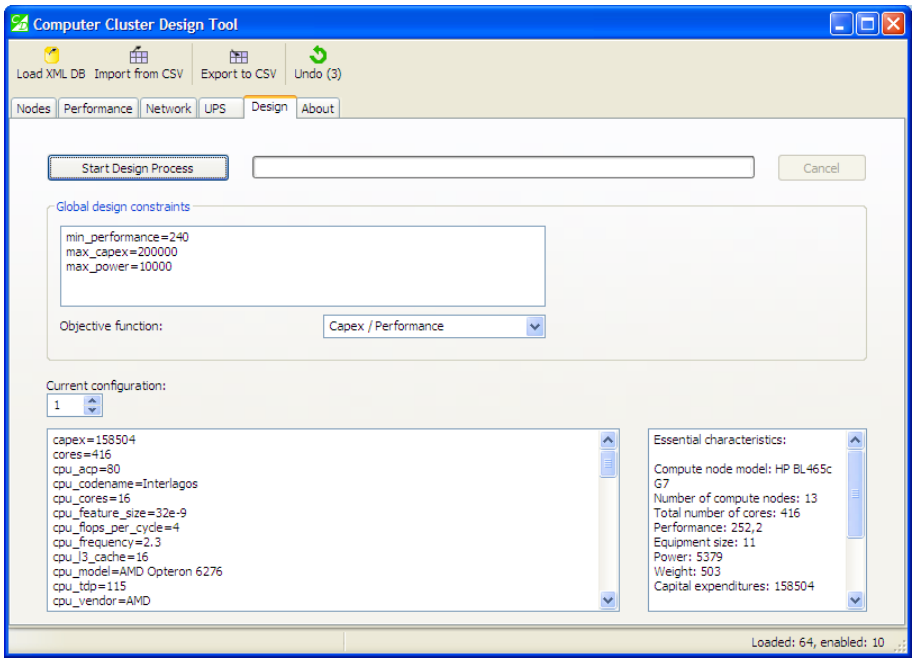


Figure 12.5.: Design results tab of the prototype CAD tool, with the best configuration.

Characteristic	Best design	Worst design
Number of compute nodes	13	28
CPUs per compute node	2	1
CPU model	AMD Opteron 6276	AMD Opteron 6272
Cores per CPU	16	16
CPU frequency, GHz	2,3	2.1
Total number of cores	416	448
Performance	252,2	247,1
Equipment size, rack mount units	11	21
Power, W	5,379	6,590
Weight, kg	503	593
Capital expenditures, $	158,504	197,776
Objective function, CapEx / Performance	628,49	800,39

Table 12.1.: The best and the worst of ten designs

After the "`Start Design Process`" button is pressed, the multi-stage process is launched; every configuration is separately inspected and processed (this allows for a trivial parallelisation strategy, see section 12.7 below). Upon termination of each design stage, the configuration is checked against the list of global constraints; if a violation is detected, the configuration is immediately disabled and doesn't participate in the subsequent design stages. In this case, out of 64 configurations only 10 meet design constraints and are still enabled after the design process finishes, as indicated in the status bar.

The list of configurations is further sorted according to the value of the objective function, and the best configuration is displayed. The lower left pane lists the `key=value` pairs that store technical and economic characteristics of the configuration. The lower right pane briefly represents essential characteristics in a more easily readable form. In this case, the number of compute nodes in a cluster is 13, performance is 252,2 tasks per day, power consumption is 5,379 W, and capital expenditures are $158,504. As can be seen, all design constraints are satisfied.

In fact, design constraints are satisfied for the first 10 configurations in the list, but characteristics of the best (1st) and the worst (10th) designs differ considerably, as indicated in Table 12.1. The 10th design is 27% worse by the value of objective function than the 1st. A thorough analysis is provided in Chapter 17.

54 configurations out of 64 do not meet design constraints and were marked as disabled during various stages of the design process. In this case, the CAD tool provides the explanation for disabling when hovering the mouse over the red circle (see Figure 12.6). For example, for the 11th configuration a constraint on capital expenditures was violated. As the configuration list is sorted according to the value of the objective function, and for disabled configurations this value cannot be calculated, those configurations appear in the list in no particular order.

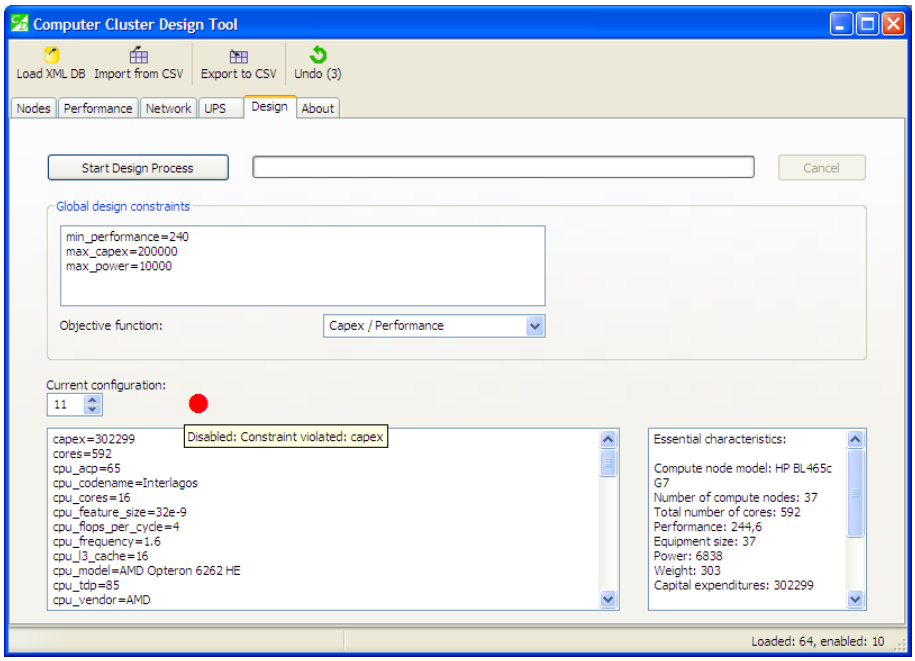


Figure 12.6.: Design results tab of the prototype CAD tool, with one of the disabled configurations.

12.6. Module Invocation Sequence

When the multi-stage design process is launched, each configuration is examined, and modules are invoked in the sequence stipulated by the algorithm outlined in Chapter 11, alternating with constraints checks:

1. Inverse performance model is invoked to determine the number of cores required to attain specified minimum performance.
2. The number of compute nodes is calculated, based on the number of cores returned by the inverse performance model. The number of cores is then recalculated by rounding it up (for example, 10 compute nodes contain 240 cores, not 237).
3. Technical and economic characteristics (cost, power, space, weight) of the computing equipment are evaluated. If there are violations of global design constraints, the configuration is marked as disabled, and the process ends.
4. Direct performance model is invoked with the newly calculated number of cores.
5. Network design module is invoked to design a network with a specified topology. Network equipment is added to computing equipment, and constraints are checked again.
6. UPS design module is invoked. UPS equipment is added to the existing hardware, and constraints are checked yet again.
7. Objective function is computed.

12.7. Parallelisation Strategy for the CAD Tool

As mentioned earlier in section 12.5, during the design process all configurations are examined separately. This suggests a simple parallelisation strategy: divide the list of configurations into equally-sized partitions, and spawn the corresponding number of threads to handle each partition. There are two important considerations here.

First, as shown in section 12.6, configurations can become disabled in the middle of the design process. Therefore creating equally-sized partitions doesn't guarantee synchronous completion of all threads, as some partitions can become empty earlier than the others.

Second, the biggest limiting factor in spawning threads is the amount of computing resources available on the servers that run modules (performance models, network and UPS design modules, etc.) If too many threads are spawned, servers will temporarily stop accepting connections, and corresponding threads will temporarily block until servers handle the load.

Alternatively, when massive queries are anticipated, design modules can be published as web services in a Platform-as-a-Service (PaaS) environment, such as "Google App Engine". This cloud environment automatically scales the number of web service copies listening to requests according to user demands.

When design modules execute quickly, the major slowdown in performing queries can be introduced by a delay in the connection phase inherent to TCP protocol. This delay can be eliminated by using HTTP request pipelining. On the other hand, when design modules execute slowly, they can be re-implemented with parallel algorithms, if possible, to speed up their execution.

13. Fat-tree Network Design

> Trees sprout up just about everywhere in computer science.
>
> *Donald E. Knuth*
> THE ART OF COMPUTER PROGRAMMING: VOL. IV–A, COMBINATORIAL ALGORITHMS

We present an algorithm to automatically design two-level fat-tree networks, such as ones widely used in large-scale data centres and cluster supercomputers. The two levels may each use a different type of switches from design database to achieve an optimal network structure. Links between layers can run in bundles to simplify cabling. Several sample network designs are examined and their technical and economic characteristics are discussed.

The characteristic feature of our approach is that real life equipment prices and values of technical characteristics are used. This allows to select an optimal combination of hardware to build the network (including semi-populated configurations of modular switches) and accurately estimate the cost of this network. We also show how technical characteristics of the network can be derived from its per-port metrics and suggest heuristics for equipment placement.

13.1. Introduction

Parallel computers use many types of networks to interconnect its computing elements. Frequently used topologies include stars, meshes, tori and trees.

"Beowulf"-style cluster supercomputers often employ fat-tree topologies built using readily available off-the-shelf InfiniBand hardware. We describe an algorithm that allows to automatically design fat-tree networks with a variety of objective functions, with the most obvious example being the total cost of network. The algorithm is implemented in a software tool [89].

This algorithm is used as a part of the CAD system for cluster supercomputers that we propose in this thesis. Such a system iterates through different combinations of hardware, varying the number of compute nodes and other parameters. Thus, designing an interconnection network for every hardware combination under review is a self-contained and highly repetitive operation that must be performed efficiently.

Many researchers of fat-tree networks concentrate on general properties of such networks and big fabrics that could be built using them. We focus on real-life scenarios, tailoring network designs to the number of network endpoints and available switches. For

example, our approach allows to select optimal configurations of modular switches, with just the right number of leaf modules installed.

Current ASIC technology enabled the appearance of readily available, off-the-shelf InfiniBand switches with $P = 36$ ports. This allows to build two-level fat-trees with as much as $P^2/2 = 648$ cluster nodes. For many typical installations this is enough.

However, vendors also provide high-radix modular switches, which internally implement a two-level fat-tree. Switches with up to $P = 648$ ports (in non-blocking configurations) are available, hence networks with more than 200K nodes can be built with the proposed algorithm – this far exceeds the demands of even the most powerful today's supercomputers.

On the other hand, intermediate-sized designs that do not use the full capacity provided by switches tend to have unused ports unless designed carefully. If no network expansion is anticipated, unused ports represent a waste of hardware resources. Therefore our algorithm tries to minimise the number of unused ports. Additionally, the algorithm reports if links between switches can run in bundles. Such bundles can be implemented with cables that aggregate multiple links, e.g., 12x instead of 4x InfiniBand cables. This results in a lower number of cables and reduced cable bulk.

During the design process, other characteristics of interconnection networks, such as reliability, can be estimated and used as design constraints or as a part of a complex objective function.

The rest of the chapter is organised as follows. Section 13.2 describes existing work in the field of fat-tree networks and their economic issues. Section 13.3 introduces the main algorithm, and section 13.4 discusses it. In section 13.5 we conduct a sample run of the algorithm and present the results. Section 13.6 explains how to obtain technical and economic characteristics of fat-trees using per-port metrics, while section 13.7 discusses design for future expansion. Finally, section 13.8 concludes the chapter.

13.2. Related Work

Fat-trees were initially introduced by C. Leiserson [56]. The mathematical formalism to describe their structure, "k-ary n-trees", was proposed by Petrini and Vanneschi [79]. Zahavi [116] further introduces two other formalisms for describing fat-trees, *Parallel Ports Generalized Fat-Trees*, where links between switches can run in parallel, and *Real Life Fat-Trees* where bandwidth between layers stays constant to guarantee content-free operation.

A tool called *NetWires* [23] was created by H.G.Dietz as part of the bigger project *Cluster Design Rules* [22]. *Netwires* is able to design different types of interconnection networks, including trees, tori and a specific *Flat Neighbourhood Network*, using user-supplied parameters, and outputs a wiring diagram. Aside from the number of required switches, no other technical or economic characteristics are assessed. Our approach is different in that we only require a few input parameters from the user, and iterate through other parameters automatically, trying to find a combination that yields an optimal value to a certain objective function subject to constraints.

Gupta and Dally [36] suggested a tool to optimise network topology, in the broad class of hybrid Clos-torus networks. Cost, packaging and performance constraints can be specified. This tool is most valuable for building custom interconnection solutions when arbi-

trary topologies are feasible, contrary to the case of using commodity switches where most parameters are fixed but optimisation can take actual prices into account.

Al-Fares *et al.* [2] proposed to use fat-trees for generic data centre networks, using commodity hardware. Farrington *et al.* [31] followed up, suggesting to build a 3,456-port data centre switch with commodity chips ("merchant silicon") internally connected in a fat-tree topology. They also advice to use optical fibre cables with as much as 72 or even 120 separate fibres (strands) to minimise the volume and weight of cable bundles for inter-switch links.

Mudigonda *et al.* [63] introduced *Perseus*, a framework to design fat-tree and HyperX topologies for data centres, and elaborated on cable tracing issues. However, fat-tree topologies built by *Perseus* use identical switches on all levels.

Parallel applications typically exhibit locality of communications. Therefore in multi-level non-blocking fat-tree networks the bandwidth offered by upper levels may remain underutilised. On intermediate levels, switch ports can be redistributed so that the number of links to the lower level is bigger than to the upper level. This reduces "fatness" of the tree, providing substantial hardware savings in terms of switches and links.

Navaridas *et al.* [68] introduced such a reduced topology, *thin-tree*, and analysed its behaviour using simulation and several synthetic workloads. Overall, for the mix of workloads, different configurations of the reduced topology were found beneficial in terms of "performance/cost" ratio compared to traditional fat-trees, especially when collective operations were only lightly used. They add, however, that in the absence of a topology-aware scheduler, neighbouring processes may be assigned to physically distant processing nodes, requiring full bandwidth at upper levels and thus rendering reduced topologies useless. Necessity of topology-aware scheduling is additionally highlighted in [69].

Kamil *et al.* [51] similarly proposed a reduced topology, but used communication patterns of actual parallel applications for analysis.

Kim *et al.* [52] introduced a flattened butterfly topology, providing detailed analysis of cost breakdown for electrical cables. 12x InfiniBand cables, aggregating three 4x links, were shown to be more economical than separate 4x cables and to additionally reduce cable bulk. Their subsequent work [53] compared cost models for electrical and active optical cables, showing that in 2008 prices, optical cables are less expensive starting from 10m. Parker and Scott [75] further advocate for the adoption of optical interconnects.

Singla *et al.* [87] proposed to abstain from rigid network structures such as fat-trees, and connect switches in a random order, in a topology called *Jellyfish*. They found that with the same performance figures and the same network equipment as the fat-tree, their topology supports more servers (performance results were obtained via simulation with random permutation traffic). Another benefit of *Jellyfish* is the ability of incremental expansion.

13.3. Algorithm

Let us consider the algorithm to design fat-tree networks with two levels of switches (namely, *edge* and *core* layers). Suppose we have two databases, for edge and core switches, respectively, with each switch characterised primarily by the number of its ports. For the number of ports of a specific edge switch we will use the designation P_E, and for a core switch we will use P_C.

Some switch models can be used for building both core and edge levels, and can be present in both databases. The two layers of network can employ different types of switches, but switches within the same layer are identical.

Let $\mathbb{E}$ and $\mathbb{C}$ be the sets of edge and core switches. Each i-th switch is characterised by its model and the number of ports, e.g.: $\mathbb{C} = \{\langle M_{C_i}, P_{C_i}\rangle\}$. These sets are the algorithm's input. Their structure allows them to contain several models of switches with the same number of ports but with differing characteristics, such as cost, reliability, energy consumption, etc.

For blade servers, which are installed into enclosures, edge-level switches are also installed in the same enclosures, and thus $\mathbb{E}$ usually contains only one element – a single switch, compatible with the enclosure. Ordinary rack-mounted servers can, on the contrary, use a variety of edge-level switches.

Apart from $\mathbb{E}$ and $\mathbb{C}$, other inputs for the algorithm are N, the number of compute nodes that need to be interconnected, and Bl, the blocking (oversubscription) factor, which denotes the decrease in bandwidth available to compute nodes compared to a full, non-blocking fat-tree.

The outputs of the algorithm are models of edge and core switches used to obtain the optimal design, as well as E and C, the number of edge and core switches, respectively, and f, the value of the objective function.

In a fat-tree network with a blocking factor Bl and edge switches with P_E ports, $P_{En} = \lfloor P_E \cdot (Bl/(1+Bl)) \rfloor$ of those ports are used to connect compute nodes, and the rest are used to connect the edge switch to the core layer. Under these conditions, in order to connect all N nodes, $E = \lceil N/P_{En} \rceil$ edge switches are required.

The remaining ports on edge switches are connected to core layer switches. When building the core layer, each port on an edge switch is connected to a different core switch. It means that a core switch must have at least as many ports as there are edge switches. For example, first ports of all edge switches will connect to the same core switch. As a result, core switches must have at least $P_{Cmin} \geq E$ ports.

Similarly, if a core switch has $P_C \geq P_{Cmin}$ ports, it can be connected to a maximum of P_C edge switches, each of those having P_{En} compute nodes connected to it. Therefore, the maximum number of nodes that could be connected is $N_{max} = P_C \cdot P_{En}$.

The algorithm structure is as follows. First we check for two trivial cases where a full two-level fat-tree network is not required. Then we iterate through a set of edge switches. For every edge switch in the set, we evaluate multiple possible network designs, trying suitable core switches, and choose one of them. Finally, the best design over all iterations is selected.

Let us describe the algorithm by stages.

1. The first stage is to check for the trivial case of two blade enclosures (line 2). In this set-up, there are two enclosures of $P_E/2$ servers, each fitted with built-in edge switches with P_E ports. Half of the ports of each switch are connected to servers. The two switches can be directly connected together with $P_E/2$ cables, and a core level is not necessary.

   A cheaper alternative is to replace one of the switches with a "pass through" panel, also allowing to directly connect this enclosure's servers to the remaining second switch.

Algorithm 2 Design a two-level fat-tree network

Input:
N: Number of nodes to interconnect
Bl: Blocking factor
$\mathbb{E}, \mathbb{C}$: Sets of edge and core switches
Goal: Optimal network structure:
E, C: Number of edge and core switches
Bl_r: Resulting blocking factor
B: Number of links in a bundle
L: Number of cables
f: Objective function for the optimal network structure

{ First trivial case: }
if (Using blade servers) **and** (Only two enclosures) **then**
  Trivial case 1: connect enclosures with cables
  Compute f_1
end if
{ Second trivial case: }
if $\exists \langle M, P \rangle \in \mathbb{E} \cup \mathbb{C} : P \geq N$ **then**
  { If there exists a switch with N or more ports }
  Trivial case 2: use star network
  Compute f_2
end if
{ Main loop: iterate through edge switches }
for all edge switches $\langle M_{E_i}, P_{E_i} \rangle \in \mathbb{E}$ **do**
  $P_{En_i} \leftarrow \lfloor P_{E_i} \cdot (Bl/(1+Bl)) \rfloor$ { Ports to nodes }
  $P_{Ec_i} \leftarrow P_{E_i} - P_{En_i}$ { Ports to core level }
  $Bl_r \leftarrow P_{En_i}/P_{Ec_i}$ { Resulting blocking }
  $E_i \leftarrow \lceil N/P_{En_i} \rceil$ { Number of edge switches }
  for all core switches $\langle M_{C_j}, P_{C_j} \rangle \in \mathbb{C}$ **do**
    if $P_{C_j} \geq E_i$ **then** { Core switch suitable }
      { Try core switch M_{C_j} }
      $B \leftarrow \min(P_{C_j} \div E, P_{Ec_i})$ { Links in a bundle }
      $C \leftarrow \lceil P_{Ec_i}/B \rceil$ { Number of core switches }
      $L \leftarrow N + E_i \cdot P_{Ec_i}$ { Number of cables }
      Compute $f_{i,j} = f(E_i, C_j)$
    end if
  end for
end for
Choose optimal combination of M_C and M_E: $f_3 = min\ f_{i,j}$
Output optimal network structure: $f = min(f_1, f_2, f_3)$

(In case of rack-mounted servers, these complications are irrelevant, because two blocks of $P_E/2$ servers can always be connected with a single edge switch with P_E ports).

We check if this configuration satisfies design constraints, and if yes, compute the value f_1 of the objective function (in particular, expandability constraints could be violated).

2. The second stage concerns the trivial case of a star network (line 7). If there exists a switch, in either $\mathbb{E}$ or $\mathbb{C}$, with enough ports to accommodate all N nodes, it can be used to build a star network. If several such switches exist, we choose one. Similar to the above case, the value of the objective function, f_2, is then computed.

3. The main loop iterates over available edge switches using index i.

   a) For every switch model, we calculate: P_{En_i}, the number of ports that are connected to compute nodes (line 14), P_{Ec_i}, the number of ports connected to the core level (line 15), Bl_r, resulting blocking factor (line 16), and finally E_i, the number of required edge switches (line 17).

   b) We then iterate through all core switches using index j (line 18). If the number of ports on the core switch makes it suitable, we perform the following actions.

      i. Calculate B, the number of links that run in parallel between edge and core switches (line 21).

      The core level is built in the following way. We take one core switch. For every edge switch, we connect its first port to the core switch. As we have E edge switches, this operation will occupy E ports on the core switch.

      Now, we repeat this step several times until we run out of ports on the core switch. If this step is performed for a total of B times, then each of the edge switches becomes connected to the core switch with a bundle of B links. B can be obtained with a simple equation: $B = P_{C_j} \div E$.

      In certain rare cases with high blocking factors (see Example 3 below), only one core switch is necessary to connect together all edge switches, and then links from all ports on the edge switch directed towards the core level form a single bundle $B = P_{Ec_j}$. Line 21 handles this scenario using the min function.

      ii. After determining B, we calculate the number of core switches C.

      iii. At this point, the number of edge and core switches becomes known, hence we can calculate the value of the objective function $f_{i,j}$ for this particular fat-tree configuration.

   c) We choose the optimal fat-tree configuration: $f_3 = min\ f_{i,j}$ (or, alternatively, present a human designer with several choices)

4. From all combinations obtained with the previous steps (trivial cases 1, 2) and main loop (3), we choose the one with the optimal value of the objective function (line 29).

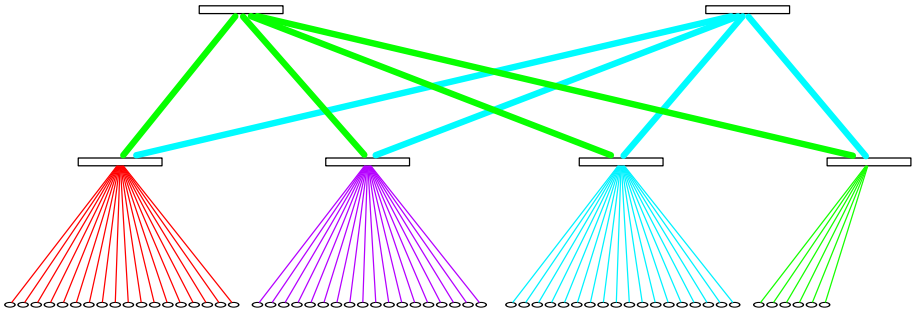


Figure 13.1.: Network of $N = 60$ nodes, created by the proposed algorithm. Thick lines between edge and core levels represent bundles of nine network links.

Example 1. Suppose we need to interconnect $N = 60$ nodes using 36-port switches ($P_E = P_C = 36$) with a non-blocking network ($Bl = 1$). The algorithm would return $E = 4$ edge switches, $C = 2$ core switches, and $B = 9$ links in a bundle.

The wiring diagram for the resultant network is shown in Figure 13.1. Note that on the rightmost edge switch only 6 ports are utilised, and 12 ports are left unused. Lines between edge and core switches are thicker to represent multiple links connecting switches in edge and core layers. In this case, bundles of $B = 9$ links are used. Running links in bundles allows for greater maintainability. Additionally, using 12x InfiniBand cables that aggregate three 4x links allows to decrease the number of physical cables in a bundle to only three.

Example 2. Let us design a network for $N = 1200$ nodes, using a blocking factor of $Bl = 2$. We will use edge switches with $P_E = 36$ ports and core switches with $P_C = 108$ ports. Out of 36 ports on the edge switch, $P_{En} = 24$ will be connected to compute nodes, and the remaining $P_{Ec} = 12$ ports will be connected to the core layer. This provides the blocking factor of $Bl = 24/12 = 2$. The algorithm would return $E = 50$ edge switches, $C = 6$ core switches, and $B = 2$ links in a bundle.

Example 3. Let us now design a network for $N = 280$ nodes with an artificially high blocking factor $Bl = 11$, using 36-port switches (such a configuration is unsuitable for HPC workloads, and is more relevant for generic data centre environments). The algorithm would distribute ports on edge switches in the following way: $P_{En} = 33$ ports will be connected to compute nodes, and $P_{Ec} = 36 - 33 = 3$ ports will be connected to the core level. The resulting blocking factor is $Bl_r = 33/3 = 11$. The number of edge switches is $E = 9$. Only three ports on edge switches are available for connecting to the core level, therefore they will form a single bundle: $B = 3$. The number of core switches is then $C = 1$.

Note that, when the number of edge switches E is determined, there are two possible scenarios of connecting compute nodes to edge switches: (1) connect as many nodes to each switch as possible, and leave the last switch underutilised (see Fig. 13.1), or (2) distribute compute nodes uniformly between all switches. The latter scenario can, in very rare cases, lead to a lower number of required core switches. However, it also leads to difficulties when expanding the network, because there may not be enough free space in racks. Our software tool [89] that implements the algorithm actually checks for that condition, and uses this scenario only if it provides hardware savings *and* if the user didn't specify preference for expandability.

13.4. Discussion

13.4.1. Switch Configurations

We consider two types of switches. First is an ordinary commodity non-modular switch with some redundant components. An example would be a typical off-the-shelf 36-port InfiniBand FDR switch, with redundant fans and an optionally redundant power supply, but with a non-redundant management board.

The second type is the modular switch. Consider, for example, a 144-port InfiniBand switch, equipped with 9 line cards, each allowing to connect 16 nodes in a non-blocking configuration. Fabric boards are used to provide internal fabric of the switch, and all four fabric boards must be installed to make the switch non-blocking. Line cards and fabric boards are installed into the chassis, which also contains redundant power supplies, redundant fans, and redundant management boards.

(Such a switch itself contains a two-level fat-tree, with links between core and edge levels running in bundles of $B = 4$ and implemented as traces on its backplane printed circuit board. Parameters of this fat-tree network are as follows: $P_E = 32$, $P_C = 36$, $E = 9$, $C = 4$).

Modular switches have large port counts, this reduces the overall number of switches in the network, improving manageability and simplifying cable routing. Additionally, they allow for future expandability by adding more line cards when required. These benefits often outweigh their higher prices per port.

When the number of nodes to be interconnected is lower than the number of ports provided by the fully configured modular switch, a reduced configuration can be used, with fewer line cards installed. This allows to significantly decrease the cost of the switch compared to the full configuration.

Different reduced configurations are treated as different models of switches in the databases $\mathbb{E}$ and $\mathbb{C}$, because they have differing technical and economic characteristics.

13.4.2. Design Constraints and Objective Functions

Objective functions can be diverse, and various design constraints can be specified. Let us confine ourselves to a single example. Suppose we need to choose one of the following switches for the core level: (a) 144-port fully configured modular switch, or (b) partially configured 324-port modular switch, with 144 configured ports. The former takes up less space in a rack, while the latter provides for future expansion.

If constraints on equipment size are imposed, the 144-port switch will be used, and the 324-port switch might not be even tried if it violates constraints. Conversely, if constraints on future expandability are imposed, the 144-port switch might get discarded. If no constraints are imposed, exhaustive search will be performed: the value of the objective function (e.g., the total cost of ownership of the network) will be calculated for both variants to make the decision.

13.4.3. Cable Count

Cables that interconnect edge and core levels are laid out at installation time, and later updates are difficult and costly.

Therefore, when future use of spare ports is anticipated, we recommend establishing a full fabric between edge and core levels. As we have E edge switches, whose P_{Ec} ports are connected to core level, we need $E \cdot P_{Ec}$ cables to establish a full fabric between the layers. Additionally, N cables are needed to connect compute nodes to the edge level.

The complete expression for the number of cables is therefore $L = N + E \cdot P_{Ec}$. For example, in Figure 13.1 the number of cables is $L = 60 + 4 \cdot 18 = 132$.

For blade servers, the connection of compute nodes to edge switches doesn't require cables, therefore the first summand in the above formula is eliminated.

13.5. Experimental Results

We apply the proposed algorithm to a real-life scenario, perform detailed calculations and discuss economic implications. Prices are subject to change over time, but this does not affect generality of conclusions.

We build a fat-tree network for a cluster of $N = 224$ blade servers. The cluster is built with 14 enclosures, each of them containing 16 blade servers. Every enclosure is fitted with an edge switch with $P_E = 32$ ports.

13.5.1. Design Database

We will use two possible core switches: (a) a 36-port monolithic switch with a price of $11,000, which is roughly $306 per port, and (b) a modular switch which can be configured with up to 108 ports, in multiples of 18. In the network design tool this modular switch is represented as six switches of 18, 36, ..., 108 ports. Therefore, $\mathbb{C}$ contains seven items in total.

The modular switch consists of a chassis ($25,000), 3 fabric boards necessary to make the switch non-blocking ($9,000 each), and a required number of line cards (up to 6), providing 18 ports each ($13,000 each). Full configuration costs $130,000, or $1,204 per port. This is a four times higher price per port compared to a simple 36-port switch.

Modular switches have a lower port density, therefore using them can unexpectedly increase the total space taken by network equipment. If only limited space is available, this can be dealt with by imposing constraints on equipment size when running the algorithm.

13.5.2. Possible Core Level Configurations

According to the algorithm, on the edge level $E = 14$ switches will be used. On the core level, there are seven possible choices of core switches. The least expensive configuration of the core level ($88,000) is obtained when using eight 36-port monolithic switches.

Reduced configurations of the modular switch with 18, 36, 54 and 72 ports result in unreasonably high costs, and we don't analyse them here. They could also be discarded using the following heuristic: modular switches are cost-effective when configured close to their full capacity.

Of special interest are, however, configurations with 90 and 108 ports. Both of them require $C = 3$ core switches. The 90-port configuration will be used, as it is slightly cheaper ($117,000) than the full 108-port configuration ($130,000). The cost of core level with this

configuration is therefore $3 \cdot \$117,000 = \$351,000$. This is roughly 4 times more expensive than with a 36-port switch.

13.5.3. Factoring In Other Costs

We continue to compare two configurations of the core level – with 36-port switches and with 90-port switches. Let us now factor in the cost of 14 edge-level switches, located in enclosures (one switch per enclosure, $11,000 each), and the cost of 224 cables (as per section 13.4.3) of $80 each (an averaged price for cables of this length, calculated manually). The per-port total costs of the two networks are $1,160 and $2,334, respectively – a twofold difference.

If we further add the cost of blade servers, equipped with dual CPUs, memory and InfiniBand adapters ($9,600 per each server), and cost of 14 enclosures ($7,500 per each), we will receive the total costs of the computer cluster, $2,515,320 and $2,778,320, for networks made of monolithic and modular core switches, respectively. The difference per connected server diminishes to 10,4%. It means that for a small premium we can attain a possibility of future network expansion and greatly simplify cabling.

Additionally, these calculations demonstrate that using blocking networks, such as *thin-trees*, will only marginally reduce total cost of the supercomputer, while potentially having severe consequences on performance.

It is worth noting that cost of *cables* is very low compared to the cost of entire computer cluster. This justifies the use of rough approximations of cable costs when designing entire computers (see also section 16.3).

13.6. Per-port Metrics

Let us consider a particular case of a network where edge and core switches are identical, and all ports are occupied. Some useful metrics can be derived for such networks.

Let us denote port count on edge and core switches by P. The network can connect a maximum of $N = N_{max} = P^2/2$ nodes. There will be P edge and $P/2$ core switches, for a total of $3P/2$ switches. As switches have P ports each, the total number of ports on all switches will equal to $3P^2/2 = 3N$, which is thrice the number of nodes. In other words, for each of N connected nodes, the two-layer fat-tree network employs three ports (and a three-layer network employs five).

Several important characteristics, such as network cost and power consumption, are "additive" in a sense that x identical switches cost x times more than a single switch and consume x times more power. The same applies on a per-port level: a network of identical arbitrarily connected switches, with a total port count of y, costs y times more and consumes y times more power than per-port cost and power consumption, respectively.

This allows to easily determine a rough estimation of cost, power, rack space, weight and possibly other characteristics of a network that supports N nodes by simply multiplying corresponding per-port characteristics of switches by $3N$, without the need for detailed analysis.

For example, a 36-port switch mentioned in section 13.5.1 has a cost of $306 per port. Typical power consumption reported by manufacturer is 152W with copper cables, which

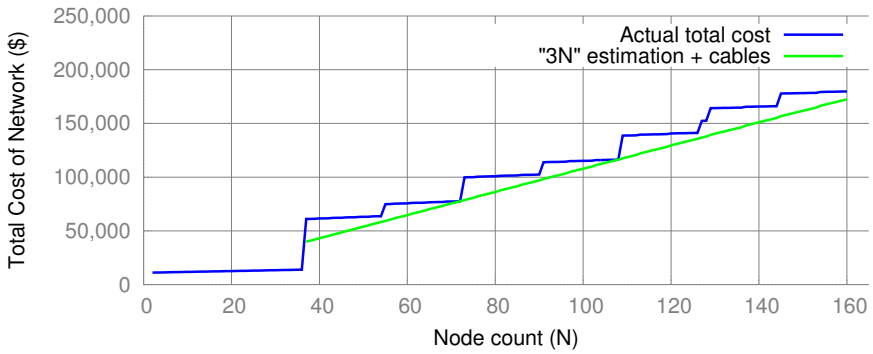


Figure 13.2.: Network cost, estimated and actual.

is 4,22W per port. The switch occupies 1U of rack space, hence "per-port rack space" is 1/36.

For a full configuration of $N = 648$ nodes, power consumption is $3N$ times per-port consumption (8,204W). Cost is $3N$ times per-port cost ($594,864). Occupied rack space is $3N$ times "per-port rack space" and equals 54U.

This estimation is also correct when the number of nodes N is X times smaller than N_{max}, where X is a non-trivial factor of $P/2$. In this case links between core and edge layers run in bundles of X. For example, if $P = 36$, valid values for X are 2, 3, 6 and 9. The estimation will thus be accurate for clusters of 324, 216, 108 and 72 nodes.

In all other cases there will be spare ports on core and possibly edge layers, and the above approach will systematically underestimate metric values, because the actual network will have more than $3N$ ports. Therefore, the estimation provides the *lower bound* on metric values.

Figure 13.2 provides an example. The blue line represents the actual cost of network built with 36-port switches, including cables, for $N \in [2, 160]$ nodes. A region of $2 \leq N \leq 36$ nodes represents a trivial case of star network, with only one switch: no fat-tree is required, hence the cost of network is kept low. Starting from 37 nodes, a two-layer fat-tree is used. Stepped behaviour of the blue curve is explained by increased switch count for every additional $P/2 = 18$ nodes. Monotonic increase inside a step is caused by increased cable count for every connected node.

The green line starts at 37-th node and represents the above estimation: $3N$ multiplied by per-port cost, plus the cost of cables. At 72 and 108 nodes it exactly matches the actual cost, as discussed above, but in other points a discrepancy is observed, with the median value of 12%.

This result allows to quickly obtain engineering evaluations of fat-tree characteristics without referring to the algorithm.

13.7. Designing for Future Expansion

Expanding existing fat-tree networks can be a difficult task. While adding edge level switches is easy, core level switches might not have spare ports necessary for expansion if this was not taken care of during design phase. We suggest to design a core level for the

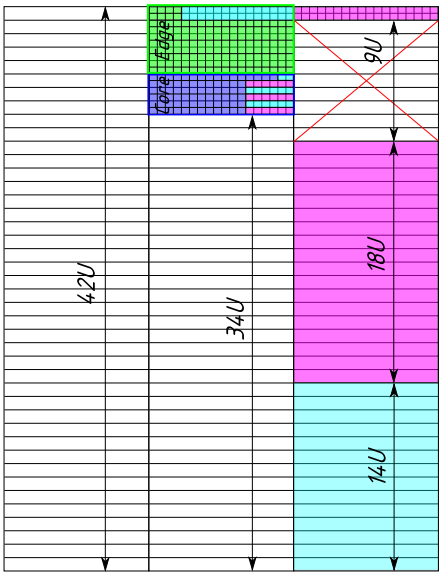


Figure 13.3.: Equipment placement for partly expandable configuration.

largest anticipated number of ports, and then gradually connect more nodes via additional edge switches as the need arises.

We demonstrate with a real-life scenario that failure to properly construct the core level can lead to non-expandable networks. Suppose we need to build a computer cluster using 1U compute nodes and commodity switches with $P = 36$ ports. Additional design constraint is that the cluster shall initially occupy two standard 42U racks with as many nodes as possible, and be expandable to three racks in the future – imitating lack of space in the machine room.

If we do not take expandability into account, the network design process proceeds as follows. Two racks contain 84U of space. With $N = 84$ nodes we require $E = 5$ edge and $C = 3$ core switches which would occupy additional 8U, thus exceeding allotted space. Hence we reduce node count to $N = 76$.

Resulting equipment placement is presented in Figure 13.3, with a close-up of the area of interest in Figure 13.4.

Initially there are two racks, the left and the middle. Five edge switches are located in the top of the middle rack, followed by three core switches. All remaining space is occupied by $N = 42 + 34 = 76$ compute nodes.

Let us discover opportunities for expandability when the third rack becomes available. First, we can use spare ports in already installed edge and core switches. There are 28 spare ports in the topmost edge switch, shown in cyan, and 32 spare ports in three core switches, shown in cyan and magenta.

Using 28 spare ports in the edge switch allows to connect 14 more nodes which would be placed to the bottom of the newly available right rack, denoted by a cyan block. Of 28 ports, half would be connected to the new nodes, and the remaining half would be connected to corresponding cyan ports of core switches.

Next opportunity for expansion lies in installing a sixth edge switch in the top of the right rack, shown in magenta. 18 ports of this switch will be connected to 18 nodes in

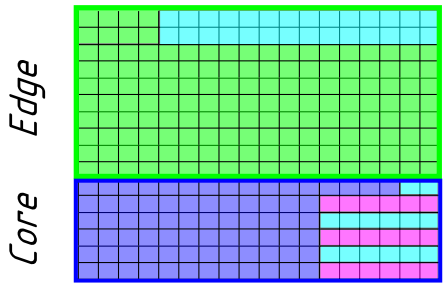


Figure 13.4.: Close-up of the area of interest.

the right rack, denoted by a magenta block. Remaining 18 ports would be connected to corresponding magenta ports of core switches.

Now opportunities for expansion are exhausted. A total of $N = 108$ nodes were connected, and nine units of rack space cannot be utilised, violating the design constraint. Further expansion would require redesigning the core layer: adding more switches and rewiring connections. For large clusters this is a complex and costly task that should be avoided if possible.

Instead, we can design a network from the start to accommodate the largest anticipated number of nodes. In this case, three racks can house $N = 126$ nodes. The network will consist of $E = 7$ edge and $C = 4$ core switches, occupying in total 11U of rack space. Hence node count shall be respectively reduced to $N = 115$. All three racks will be fully populated.

This allows to expand the cluster by 7 additional nodes, compared to the previous variant. However, this also incurs increased network cost, as 11 switches are used instead of 9, so design decisions have to be carefully balanced.

For this newly designed expandable network, there are two alternative variants of equipment installation in the initial two racks:

1. Install all $C = 4$ core and $E = 7$ edge switches at once. This requires 11U of racks space, and leaves space for 73 nodes. Additional 42 nodes will be added when a new rack is available.

2. Install all $C = 4$ core switches and as many edge switches as required to fill up two racks with nodes, namely, $E = 5$ edge switches. This requires 9U of rack space, and allows to install 75 nodes. Additional 40 nodes and two edge switches will be added when a new rack is available.

The latter variant allows to populate initial two racks with more nodes and reduce original investment in edge switches, as their procurement can be delayed until the expansion stage.

13.8. Conclusions

We presented the algorithm to automatically design two-layer fat-tree networks with arbitrary blocking factors. We applied proposed algorithm to design several networks and

analysed their characteristics. We demonstrated that a lower bound (and a rough approximation) for many technical and economic characteristics of the whole network can be easily obtained from per-port metrics. We also discussed expandability of fat-trees.

14. Torus Network Design

In this chapter we present a simple algorithm to calculate the number of switches in a torus network, based on the number of compute nodes to be interconnected and, optionally, a blocking factor. Results obtained using this algorithm are later used to compare technical and economic characteristics of tori to fat-tree networks.

14.1. Related Work

Torus networks have found widespread use in supercomputing. IBM used a 3D torus network in BlueGene/L, and a 5D network [21] in BlueGene/Q. A 6D mesh-torus network was used in "K Computer" [1]. Both are *direct networks*, where compute nodes are connected directly to their neighbours, as opposed to *switched fabrics*, where nodes are first connected to switches, and then switches are connected to each other in a torus topology. The example of the latter is a 3D torus network for the Gordon supercomputer [98].

Torus networks are inherently prone to congestion, but this is mitigated by designers by increasing the number of dimensions. Commenting on the Gordon project, Strande [97] quotes the following benefits of torus networks: (a) lower cost compared to fat-trees and (b) easy linear scaling along one of dimensions. However, such scaling may result in unbalanced topologies, leading to bigger latencies and higher congestion on the links in that dimension. Strande also mentions that the torus topology uses short cables, which makes the use of fibre optical cables unnecessary, leading to further cost savings.

Navaridas and Miguel-Alonso [67] analysed performance of 2D switch-based torus topologies and fat-trees for up to 7,680 compute nodes, on a range of workloads, using simulation techniques. They conclude that performance degradation from using torus networks, compared to fat-trees, can reach 20..40%, and sometimes more, on communication-intensive workloads, which limits applicability of tori in larger installations.

Cámara *et al.* [19] introduced the technique to turn unbalanced rectangular 2D and 3D tori to twisted tori by rearranging peripheral links, which improves performance characteristics as well as regains network symmetry.

14.2. 3D Dual-rail Torus Network of the Gordon Supercomputer

Gordon supercomputer [98] uses InfiniBand switches with $P = 36$ ports of 4X QDR technology. Switches form a 4x4x4 torus; each switch has 6 neighbours, to which it connects with 3 links, thereby utilising 18 ports out of 36. 17 more ports are used to connect 16 compute nodes and one I/O node.

The network is *dual-rail*, therefore there are actually two tori made of switches, and compute and I/O nodes have two network interfaces, one of which is used to connect to the switch in the first torus ("rail"), and the other to the second one. Currently, one rail is

used for MPI, and the other one for I/O traffic. According to Strande [97], there are plans to use both rails simultaneously to provide failover capabilities and improve bandwidth.

14.3. Algorithm for Designing Torus Networks

We propose the algorithm to calculate the number of switches in a torus network, using as input the number of compute nodes to be interconnected and, optionally, a blocking factor that determines the distribution of ports on a switch between compute nodes and neighbouring switches. The algorithm is suitable to design networks built with commodity hardware, such as Gordon's network.

As torus networks are inherently prone to congestion, imposing additional blocking at the switch level is very disadvantageous. However, sometimes blocking is stipulated by the hardware manufacturer, and cannot be avoided. For example, in [67] the hardware under review was a blade chassis equipped with $N = 20$ compute nodes and an InfiniBand switch with $P = 36$ ports. Only 16 ports of the switch were used to connect it to the outside world, which resulted in $Bl = 20/16 = 1{,}25$ blocking factor. In order to build torus networks for such hardware with the proposed algorithm, we need to specify the blocking factor as an input.

The algorithm tries to build a network using identical switches with P_E ports. Let us describe the algorithm by stages. In line 1 we check if the switch has enough ports to connect all N nodes. In this case, we use the star topology with only one switch and exit.

Otherwise, we will build a ring or a torus. In lines 8..10 we calculate the number of switch ports that go to compute nodes and to the neighbouring switches, and then recalculate the blocking factor for the network. On line 11 we derive the minimal number of switches required to connect N nodes with a given blocking factor. The actual torus will contain slightly more switches (generally, the increase is within 20% for small networks, and within several percent for the large ones).

On line 12, we use a heuristic to determine the number of torus dimensions, based on the number of switches. It is important to note that there are no hard rules when choosing the number of dimensions. Choosing a low number of dimensions for a high number of compute nodes leads to increased network diameter and therefore latencies. On the other side, choosing a too high number of dimensions for a low number of compute nodes does not provide network performance benefits but results in complex cabling patterns. In the case of direct networks this scenario also requires network adapters with an unnecessarily large number of ports.

The optimal number of dimensions depends on the communication pattern of the application, and can be reliably determined, for any given application, only through benchmarking on real hardware or by using simulation such as in [67]. Therefore we relied on using a heuristic.

Currently, the dimension choice heuristic returns the number of dimensions as per Table 14.1, up to $D = 5$. The layout of switches in the maximal configuration for that number of dimensions is provided in the last column of the table for reference.

If the heuristic returns $D = 1$, then we use the ring topology (line 14). Otherwise, we use the torus topology, and need to calculate the number of switches along each of D dimensions by rounding $\sqrt[D]{E}$ to the nearest integer (line 17).

This creates a topology close to an ideal square, cube, etc. Packaging constraints, however, may preclude from using this particular ideal layout, and in the resulting unbalanced torus the number of switches along dimensions may differ significantly. The number of switches, E, still remains the same as returned by the algorithm, allowing to correctly calculate equipment cost and other metrics.

On the next step, we calculate the number of switches in the last dimension (line 18) and recalculate the total number of switches as the product of switch counts along all dimensions (line 19).

The number of cables is determined on line 21. The number of switch ports facing to neighbouring switches, P_{Ec}, is divided by two, because two ports are connected with one cable. This is then multiplied by the number of switches E. Compute nodes are connected with additional N cables. The network is expandable from N up to $E \cdot P_E$ compute nodes. Inter-switch links run in bundles of approximately $P_{Ec}/(2 \cdot D)$, therefore it is often possible to use cables that integrate several links (such as a 12x InfiniBand cable that integrates three 4x links) to reduce the number of physical cables, simplifying installation.

Sample output of the algorithm for commodity InfiniBand switches with $P_E = 36$ ports and a non-blocking network ($Bl = 1$) is presented in Table 14.2.

14.4. Cost Comparison of Torus and Fat-tree Networks

We used real life equipment costs provided by Mellanox Technologies to derive costs of fat-tree and torus networks for up to 3,888 compute nodes. We utilised the tool for automated design of cluster interconnection networks [89]. Equipment costs are given for the older generation of equipment (InfiniBand QDR), and technical characteristics are summarised in Table 14.3. Cable cost is assumed to be $80.

We consider three models of switches. The first of them, the 36-port switch, is used for building torus networks, and is also utilised on edge level of fat-tree networks. The other two are modular switches that have 108 and 216 ports in their maximal configurations. The actual number of supported ports depends on the number of installed line cards, which leads to 6 and 12 configurations of these switches, respectively. Each configuration has its own set of technical characteristics as well as cost.

The set of equipment described above allows to build non-blocking fat-tree networks with up to $N_{max} = P_E \cdot P_C/2 = 36 \cdot 216/2 = 3{,}888$ nodes. On Fig. 14.1 we plot costs of non-blocking as well as 2:1 blocking fat-tree networks, and torus networks. As expected, the cost of 2:1 blocking fat-trees is lower than of their non-blocking counterparts; but reduction in cost is less than twofold. Torus networks are consistently cheaper than fat-trees; however, their inherent blocking may have detrimental effect on application performance that will not be offset by lower costs.

We also consider an alternative way of building fat-trees: using 36-port switches for both core and edge layers. This allows to build non-blocking fat-tree networks with up to $N_{max} = 36 \cdot 36/2 = 648$ nodes. Such networks are characterised by complex wiring patterns between the two layers, but are marginally cheaper to build. Fig. 14.2 is essentially a close-up of the previous figure, focusing on values of N up to 648 nodes, with an additional curve representing costs of the alternative fat-tree building method.

As the diagram indicates, using 36-port switches for building fat-trees does indeed lead

Algorithm 3 Design a torus network

Input:
N: Number of nodes to interconnect
Bl: Blocking factor
P_E: Number of switch ports
Goal: Optimal network structure:
D: Number of torus dimensions
$d = \langle d_1, \ldots, d_D \rangle$: Number of switches along each dimension
E: Total number of switches
Bl_r: Resulting blocking factor
L: Number of cables
f: Objective function for the optimal network structure

1: **if** $P_E \geq N$ **then**
2: { If there exists a switch with N or more ports }
3: **print** Topology: star
4: $E \leftarrow 1$; $Bl_r \leftarrow 1$; $L \leftarrow N$
5: Compute f
6: Exit
7: **end if**
8: $P_{En} \leftarrow \lfloor P_E \cdot (Bl/(1 + Bl)) \rfloor$ { Ports to nodes }
9: $P_{Ec} \leftarrow P_E - P_{En}$ { Ports to other switches }
10: $Bl_r \leftarrow P_{En}/P_{Ec}$ { Resulting blocking }
11: $E \leftarrow \lceil N/P_{En} \rceil$ { Minimal number of switches }
12: $D \leftarrow GetDimCount(E)$ { Heuristic for the number of torus dimensions}
13: **if** $D = 1$ **then**
14: **print** Topology: ring
15: **else**
16: **print** Topology: torus
17: $d_i \leftarrow round(\sqrt[D]{E}) \mid i = 1 \ldots D - 1$ { Number of switches along dimensions }
18: $d_D \leftarrow \lceil E/d_1^{D-1} \rceil$ { Switches in the last dimension }
19: $E \leftarrow \prod_{i=1}^{D} d_i$ { Actual number of switches }
20: **end if**
21: $L \leftarrow N + E \cdot P_{Ec}/2$ { Number of cables }
22: Compute f

Switch count, E	Topology	Dimensions, D	Max. configuration
2 or 3	Ring	1	—
up to 36	Torus	2	6x6
up to 125		3	5x5x5
up to 2401		4	7x7x7x7
more than 2401		5	(As appropriate)

Table 14.1.: Heuristic for the number of torus dimensions

Compute nodes, N	Dimensions, D	Torus topology	Supercomputer of comparable size
1,000	3	4x4x4	Gordon [98]
6,000	4	4x4x4x6	Stampede [102]
8,000	4	5x5x5x4	Tianhe-1A [66]
10,000	4	5x5x5x5	SuperMUC [55]
19,000	4	6x6x6x5	Titan [72]

Table 14.2.: Sample output for Algorithm 3

Switch applicability	Switch model	Port count	Size, U	Weight, kg	Power, W	Cost, $
Torus, Fat-tree (edge layer)	Mellanox Grid Director 4036	36	1	7,7	202	10,820
Fat-tree (core layer)	Mellanox IS5100	18	7	75,1	516	78,500
		36		77,8	606	90,000
		54		80,6	696	101,500
		72		83,3	786	113,000
		90		86,1	876	124,500
		108		88,9	966	136,000
–"–	Mellanox IS5200	18	10	115,7	516	125,500
		36		118,4	606	137,000
		54		121,2	696	148,500
		72		123,9	786	160,000
		90		126,7	876	171,500
		108		129,5	966	183,000
		126		132,2	1,056	194,500
		144		135,0	1,146	206,000
		162		137,7	1,236	217,500
		180		140,5	1,326	229,000
		198		143,3	1,416	240,500
		216		146,0	1,506	252,000

Table 14.3.: Characteristics of InfiniBand QDR equipment

Network type	Non-blocking	2:1 blocking
Topology	Star	Two-level fat-tree
Edge level switch	Mellanox IS5200 (162 ports)	Mellanox Grid Director 4036 (36 ports)
Core level switch	N/A	Mellanox IS5100 (90 ports)
Power, W	1,236	2,290
Weight, kg	137,7	140,0
Size, U	10	14
Cost, $	229,500	218,960

Table 14.4.: Structure comparison for two types of fat-tree networks, for $N = 150$ nodes.

to certain cost savings: for $N = 648$ nodes, per-port cost of such networks is roughly $1,060, while for the usual way of building fat-trees, using modular switches on the core level, the per-port cost is roughly $1,930. However, these savings should be weighed against the cost of compute nodes: if the latter is much higher than the per-port cost of the interconnection network, then cost savings might not justify increased wiring and maintenance complexity of this type of networks.

Example 14.1 *Let us assume the cost of a compute node is $5,000. If per-port cost of two types of interconnection networks is $1,000 and $2,000, respectively, then savings from using the network of the first type is* $7000/6000$*, or roughly 17%. Factoring in costs of other equipment, as well as operating expenses, further dilutes savings.*

Figure 14.2 is particularly helpful to emphasise the structure of networks generated by the network design tool [89]. Consider, for example, the case of non-blocking and 2:1 blocking fat-trees, for $N = 150$ compute nodes. The costs of these two networks are very close, but their structure is entirely different, which is summarised in Table 14.4.

If the tool is requested to design a non-blocking network, it chooses a star topology with a single modular switch. If, however, a 2:1 blocking network is requested, the result is a two-layer fat-tree, with 36-port switches on the edge level and a 90-port switch on the core level. The latter network is chosen because it is marginally (5%) cheaper. At the same time, it draws 85% more power and requires 40% more space in the rack.

This example illustrates two points: (A) more complex criterion functions, such as total cost of ownership, should preferably be used instead of capital costs; (B) trying to design blocking networks doesn't necessarily save considerable amounts of money, therefore designers should consider non-blocking networks first.

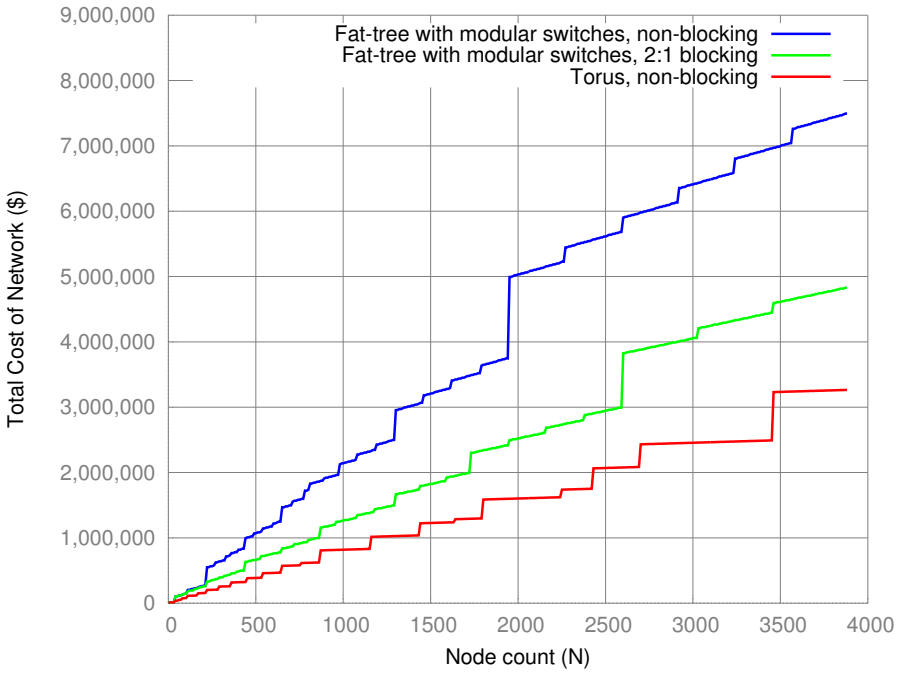


Figure 14.1.: Cost comparison of fat-tree and torus networks

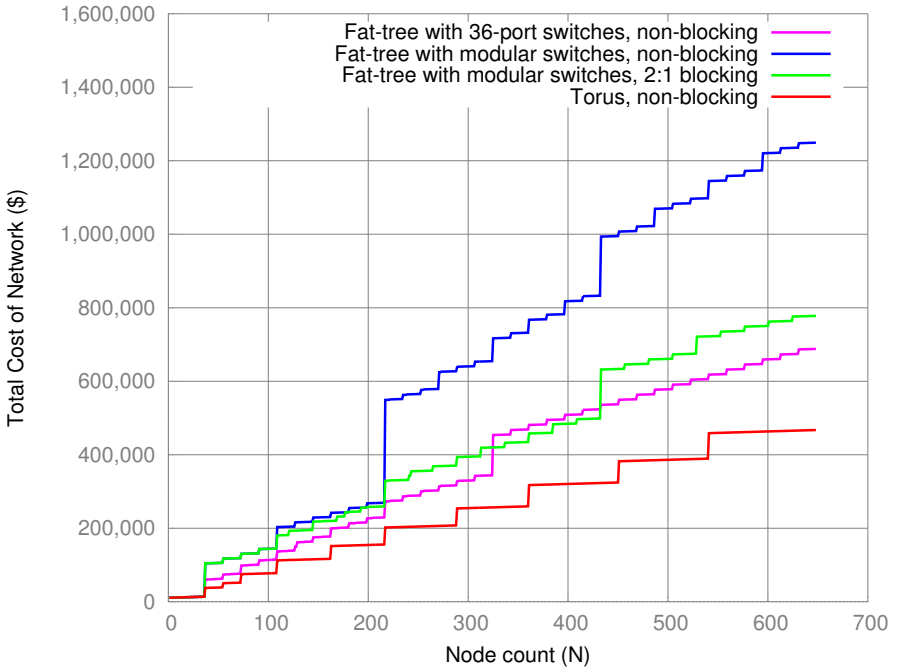


Figure 14.2.: Cost comparison of alternative fat-tree building methods

15. Designing Other Subsystems of Computer Clusters

15.1. Order of Design Stages

The order of design stages is determined by the cluster design algorithm (Chapter 11). As per the algorithm, for every configuration of compute node, given the minimum required performance, we determine the number of compute nodes, and then design the interconnection network. This way we obtain figures for power consumption of computing and network equipment. The next stage designs a storage system, which gives its power consumption (this stage is not implemented in the prototype CAD tool).

Practically all consumed power is transformed into heat that must be taken away by the cooling system. Cooling system has its own power consumption. Therefore the next stage is to design an uninterrupted power supply (UPS) system which will provide power to all classes of aforementioned equipment: computing, network, storage, and cooling.

The UPS system produces a small amount of heat during its operation, and the cooling system should be designed to account for it (we deal with this later).

We also specifically make the case for providing the cooling system with uninterrupted electrical power. For legacy air-cooled installations, a power outage could result in air conditioners being switched off, while computing equipment continued working and could overheat if shutdown procedures were not performed quickly. Such overheats may result in accumulating permanent hardware damage later manifesting itself as faults. Therefore with air-cooled equipment connecting a cooling system via a UPS is essential, even if it results in additional cost and size of power supply equipment.

Newer water cooling systems are more forgiving to power outages, due to water's higher thermal capacity. However, as we review later, power consumption of water cooling systems represents a small share of computing equipment's power consumption, which makes it reasonable to connect cooling equipment via UPS.

Power consumption of compute nodes has a cascading effect on other equipment. With all other parameters, including performance, being equal, using low-power compute nodes not only leads to power savings of computing equipment, but also leads to lower required capacities of cooling and power supply systems, which directly translates to lower costs, and in large installations can also lead to space savings.

Another factor that influences overall power consumption of the supercomputer installation is the uneven load of compute nodes with jobs, because idle nodes consume significantly fewer power. For example, according to "Hewlett-Packard" [40], "HP DL360 G7" servers, equipped with two "Intel Xeon X5670" CPUs, can draw 88W of power when idle and 193W when performing compute work. Consequently, with water cooling, the rate of water flow through idle or even powered-off servers can be reduced, leading to further savings.

15.2. Storage System

A substantial body of knowledge exists on designing storage systems, including the following three works by "Hewlett-Packard Labs" employees. Anderson *et al.* [4] present *Disk Array Designer*, the tool that designs storage systems according to capacity and performance requirements of different workloads. Ward *et al.* proposed [113] *Appia*, the framework for designing storage area networks (SAN) – sets of switches and links that connect storage devices to clients, given traffic flow patterns. Amiri and Wilkes suggest [3] to use Markov chains to design storage systems with availability requirements.

In this thesis, we don't provide any particular algorithm for designing storage systems, and refer the reader to related work outlined above. Instead, we propose that storage vendors implement storage system design algorithms, tailored to their hardware, in web-based modules that can be queried by the cluster design tool. The modules will receive as their input the design requirements for the storage system, such as capacity, performance and reliability, and will return to the tool technical and economic characteristics of the storage system, in line with other subsystems of compute clusters: cost, operating expenses, power consumption, size, weight, etc.

Tao *et al.* of "Whamcloud, Inc." describe [101] designing storage systems based on the Lustre parallel filesystem, explaining the role of all elements in the "storage pipeline" and highlighting issues as subtle as the position of host channel adaptors (HCAs) in PCIe slots on motherboards of Lustre storage servers.

Their work explains that requirements of capacity, performance and cost must be balanced. Based on their methodology, we provide the following economic analysis of storage systems with regard to different criterion functions. Suppose we need to build a storage system with the throughput requirement of 50 GB/s, and this figure can be attained with 20 enclosures fitted with high-capacity (3 TB) disks, with each enclosure offering 2,5 GB/s of throughput. Each enclosure provides 144 TB of usable space, for the total capacity of 2,9 PB. However, if we only need 1 PB of storage space, it doesn't mean we can simply decrease the number of disk drives three-fold, because performance critically depends on the number of drives in the system ("spindles").

Alternatively, we can equip enclosures with disks of the same performance but lower capacity (1 TB). Thus a single enclosure will still provide the throughput of 2,5 GB/s, but the usable space of the enclosure will be decreased to 48 TB. We still need 20 enclosures to attain the required throughput of 2,5 GB/s. The system storage capacity will be roughly 1 PB. Both variants of the storage systems are summarised in Table 15.1.

We now pose the question: if both variants are suitable, which one is preferable? We show that preference heavily depends on the choice of the criterion function.

In terms of pure cost, variant "B" is likely to be slightly cheaper (and therefore preferable), due to the lower cost of low-capacity disks. However, the cost of disk drives substitutes just a small share of the total cost of the storage system, and certain market fluctuations can even lead to small disks becoming on par or more expensive than high-capacity disks. In this situation, cost is not a robust criterion function, as we cannot use it to reliably differentiate between the two variants.

We can employ the criterion function from the "cost/performance" class. The question is how to define performance. If we define throughput to be the measure of performance, this simply reduces the case to the previous one (and "B" is preferable again), because

Metric	A	B
Size of individual disk	3 TB	1 TB
Usable space of enclosure	144 TB	48 TB
Enclosure throughput	2,5 GB/s	
Number of enclosures	20	
Total storage capacity	2,9 PB	1 PB

Criterion function	Preferable variant
Cost	B
Cost / throughput	B
Cost / capacity	A
Cost / (throughput * capacity)	A

Table 15.1.: Comparison of technical characteristics for two types of storage systems.

throughput is the same for both variants. Therefore we cannot use "cost/throughput" either.

If we use "cost/capacity", then variant "A" with its higher capacity is clearly preferable, as it provides three times more storage space at about the same price. However, this metric is not general, because it does not take throughput into account, not to mention other possible performance metrics such as IOPS (I/O operations per second).

With the emergence of "big data" applications, another possible metric is the time required to read the contents of the computer's operating memory from the storage. However, it boils down to throughput, and doesn't take other performance metrics or cost into account.

More balanced evaluation could be based on the more complex criterion function such as "cost/(throughput*capacity)". Here, capital costs should preferably be replaced with the total cost of ownership; this will account for power consumption and equipment size (which translates to required floor space) in an unbiased way.

Indeed, suppose that we shift from hard disk drives with rotating spindles to solid-state drives (SSD). They have higher performance but lower capacity, which leads to increase in the number of enclosures needed to reach required capacity, which in turn has effect on required floorspace. On the other side, SSDs have lower power consumption which leads to energy savings, while higher potential fault rate due to wear-out leads to higher expenses when replacing faulty drives. All these opposing trends can be accounted for when using total cost of ownership.

The approach with total cost of ownership is also useful for comparing existing solutions with the new technology based on completely different principles, such as comparing tape-based archival storage with "massive array of idle disks" (MAID) technology.

15.3. Cooling System

Air has been used for cooling of computing and other electronic equipment since its inception. In the recent decade, density of computing equipment was rising steadily, culmi-

nating in the advent of blade servers and similar hardware architectures that closely pack heat-generating components – CPU and GPU chips – into limited spaces. The "Lomonosov" supercomputer that first entered the TOP500 list in November 2009 [106] generated unusually high values of heat, up to 65 kW per rack.

Raising density of computing equipment requires its complete redesign and therefore incurs significant engineering costs that are later shifted to consumers. However, when air cooling is used, high density of computing equipment doesn't guarantee overall density on the data centre level. Indeed, the "Lomonosov" used blade chassis that housed 16 servers in 7U of rack space, or 96 servers per standard 42U rack. Overall, per 60 racks of computing equipment, additional 40 racks of cooling equipment were required [7]. If density of computing equipment was lowered by a factor of two, traditional rack-mounted servers could be used, significantly decreasing capital costs, and additionally providing more practical cooling regimes. At the same time, rack count would only increase by 60%. (See also discussion of blade servers in section 7.2.6).

Woods [114] reviews current industry practices in air cooling of data centres, quoting significant savings that result from placing data centres in cold climatic zones. Heat can then be rejected from interior air to the exterior, either by using heat-exchangers or through mixing two flows directly.

In 2008, Atwood and Miner [9] conducted a test for using outside air for cooling purposes. Temperature range for supplied air was artificially limited to 18..32°C: if the outside air was too cold, it was mixed with interior air before entering machine room, and if the air was too hot, it was cooled to 32°C by a traditional cooling system.

However, humidity was not controlled, and dust filtering was "minimal". At the end of the 10-month testing period, servers and the inside of the machine room were covered with dust, but failure rates were not significantly bigger (4,46% versus 3,83% in the main Intel's data centre and 2,45% in the reference installation that used direct expansion cooling at all times). Humidity of supplied air ranged between 5..30%, mainly staying within 10..15%. Estimated reduction in energy consumption for cooling purposes was 74%.

They also note that even if cooling is required in extremely hot weather conditions, it is only necessary to cool the air to 32°C which was found to be acceptable, not to the industry standard 20°C. This means that less cooling equipment is required than usually, resulting in savings in capital expenses. However, the scenario described by Atwood and Miner doesn't address the problem of contaminated outside air.

In 2009, "Microsoft" built a data centre in Ireland [61], using cold outside air for cooling needs and thus reducing operating expenses. It is noted, however, that the usual direct expansion cooling system is still in place, and will be used when outside air is unsuitable for cooling, due to its temperature or contamination. Therefore, if air contamination is possible on the site, the usual cooling system will need to be installed to organise air recirculation, and thus savings in capital costs cannot be realised (but savings in operating expenses are still in effect, because air contamination is an unlikely event, and the cooling system, although present, might never be required to run).

In 2011, "Dell" certified [70] some of its servers to run at air temperatures of 40°C for 900 hours per year and at 45°C for 90 hours per year, specifically to allow the use of outside air. Also in 2011, "Hewlett-Packard" announced [39] their modular data centre, "HP EcoPod", designed for cooling with outside air, switching to a standard direct expansion cooling system when necessary.

Based on the work referenced above, we arrived to the following conclusions for air-cooled computing equipment:

1. Cooling with outside air is feasible, it requires the least equipment (hence savings in capital costs) and leads to substantial energy savings (hence savings in operating costs). However, filters should be used to prevent dust deposits in the machine room.

2. The safe operating temperature range for outside air is between $T_L = 18$°C and $T_H = 32$°C. With vendor warranties, the upper bound, T_H, could be further raised for a limited number of hours of operation per year.

3. If on-site climate statistics indicate that air temperatures below T_L are possible, air mixing equipment should be installed to mix outside air with interior air before supplying it to the machine room.

4. Additionally, if climate statistics indicate that air temperatures above T_H are possible, a traditional cooling system of capacity P_{part} should be installed, with the intention to cool outside air to the level of T_H. This is not mutually exclusive with the previous item, because in certain climatic zones, such as deserts, temperature variations between night and day can be large enough to require the use of both methods of air preparation.

5. If air contamination is possible, the cooling system described above should be accompanied by either a standard direct expansion cooling system, or, as suggested by Woods [114], an air-to-air heat exchanger. The latter will transfer heat from interior air to the outside one without mixing them. These additional systems will require installation (increasing capital costs), but may never require operation (if air never becomes contaminated), thereby they do not necessarily add much to operating costs.

We also propose the following decision chart (Figure 15.1) for selecting cooling methods for air-cooled computing equipment.

15.4. Calculating Partial Cooling Capacity

As shown above, in hot climates it may be required to cool outside air to the level of T_H, using a standard direct expansion cooling system, before supplying air to the machine room. It is therefore necessary to calculate capacity of this cooling system.

With current industry standards, a cooling system needs to remove as much heat as the computing equipment generates; let us denote this as P watts. However, when using outside air, it is only necessary to cool it down to T_H, which usually requires removing substantially fewer heat, P_{part}, than P.

We show how to calculate P_{part}, the cooling capacity of such a system, based on statistical climate data. For this purpose we use the bar graph, similar to the one appearing in [114], that contains data for on-site temperature distribution throughout the year (see Figure 15.2).

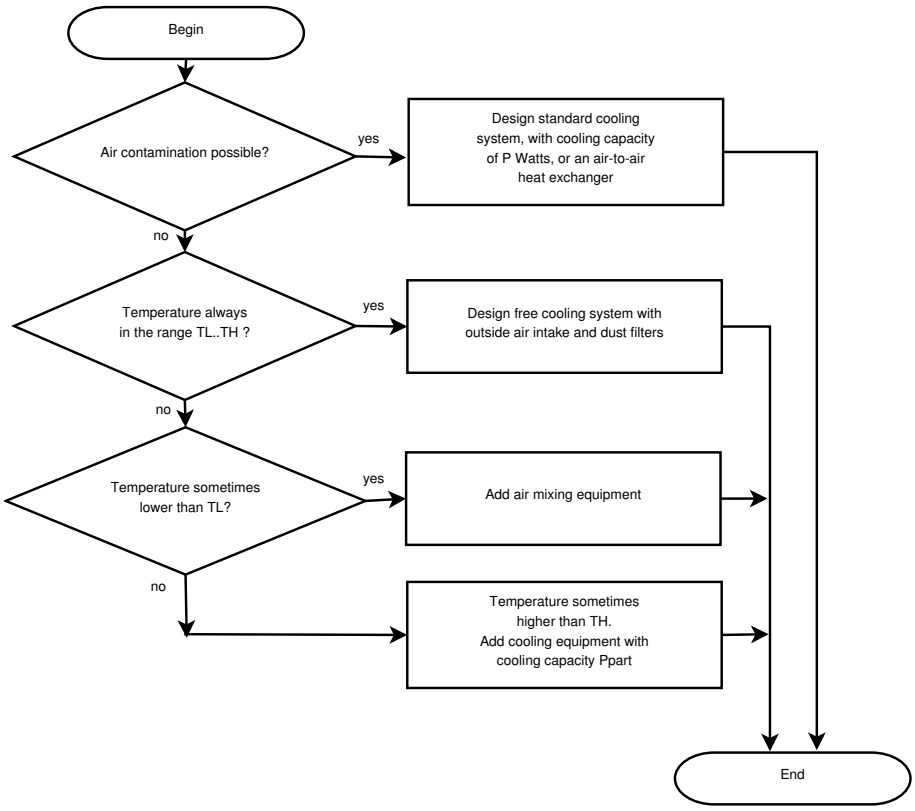


Figure 15.1.: Decision chart for selecting cooling methods for air-cooled computing equipment

This figure depicts climate data for a fictional site. For all 8,760 hours per year – that is, all year round – the temperature is higher than 20°C, but it never rises to 36°C or above. We choose the temperature, T_{max}, up to which the data centre must still be operational at full scale; most often this will be the highest temperature observed on the site.

As we assume the highest safe server operating temperature to be $T_H = 32$°C, we need to cool incoming air from at most $T_{max} = 36$°C down to $T_H = 32$°C, by $\Delta T = T_{max} - T_H = 4$°C. (If we are prepared that the data centre will run in degraded mode, with some computing equipment turned off, during hours of peak heat, we can also choose a lower value for T_{max}).

Now, we need to calculate the amount of heat that must be extracted from the outside air to cool it by ΔT degrees. We use the typical amount of air that is required to pass through servers to cool them, such as $f = 160$ cubic feet per minute (CFM) per kilowatt (kW) of equipment power [65]. First, we convert units of measurement to SI units (equations (15.1) and (15.2)). Then we calculate the volume of air required to pass per second through equipment with airflow f and power consumption P (equation (15.3)). Then we use air density (ρ) to calculate the mass of air ((15.4)), and finally use air heat capacity (c) to calculate the amount of heat that needs to be extracted ((15.5)). Therefore, each second Q joules of heat must be extracted, which essentially means the cooling capacity of the air cooling system is numerically equal to $P_{part} = Q$ (equation (15.6)).

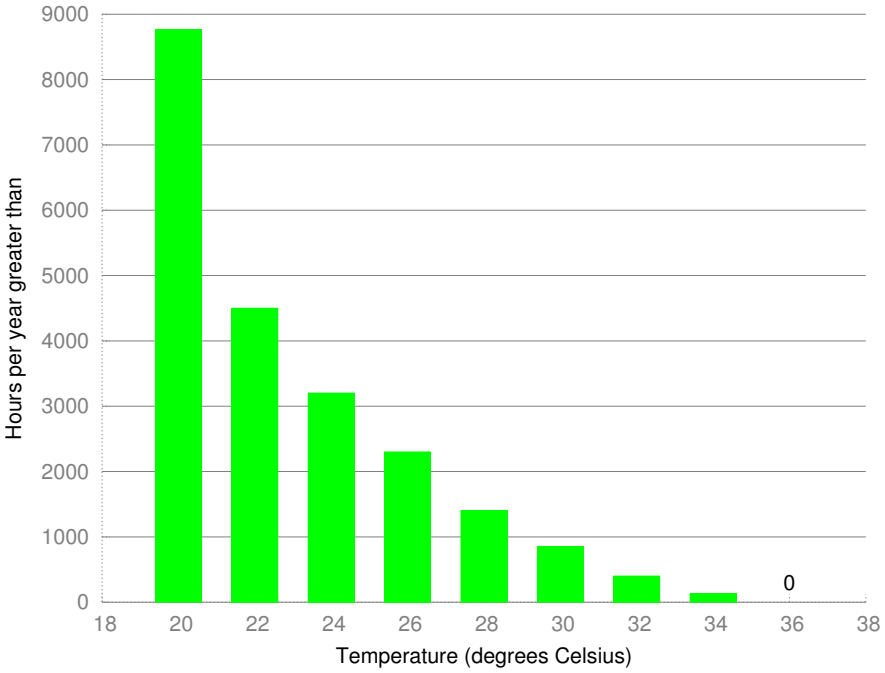


Figure 15.2.: Climate data for a fictional site

$$1 \text{ ft}^3 = 28,3 \cdot 10^{-3} \text{ m}^3 \tag{15.1}$$

$$1 \text{ ft}^3/(\text{min} \cdot \text{kW}) = 0,47 \cdot 10^{-6} \text{ m}^3 \text{ / } (\text{sec} \cdot \text{W}) \tag{15.2}$$

$$V = f \cdot P \cdot 0,47 \cdot 10^{-6} \text{ m}^3 \tag{15.3}$$

$$m = \rho \cdot V \text{kg}, \quad \text{where } \rho = 1,2 \text{ kg/m}^3 \tag{15.4}$$

$$Q = c \cdot m \cdot \Delta T = f \cdot P \cdot \Delta T \cdot 0,57 \cdot 10^{-3} \text{ Joules}, \quad \text{where } c = 1012 \text{ J}/(\text{kg} \cdot ^\circ\text{C}) \tag{15.5}$$

$$P_{part} = Q \text{ Watts} \tag{15.6}$$

Let us perform calculations according to these formulae, for a data centre with power consumption of IT equipment of $P = 1$ MW, located on the site that we described above, with ΔT=4°C.

f=160 CFM/kW
ΔT = 4°C
$P = 10^6$ W
$Q = 364800$ Joules
$P_{part} = 364800$ W

As follows from the calculation, in the conditions specified above, the required cooling capacity of the system that uses outside air is approximately 36% of the standard direct expansion cooling system. This translates into proportional reduction in required cooling equipment, with associated savings in capital and operating expenditure. Additionally, this system only operates a limited number of hours per year, when the outside temperature is higher than T_H, thereby further decreasing TCO.

Cooling with outside air is only beneficial when $P_{part} < P$. If $P_{part} = P$, cooling with outside air and closed-circuit cooling both require the same amount of energy and cooling equipment. As follows from equation (15.5), when this happens,

$$\Delta T_{peak} = \frac{Q}{f \cdot P \cdot 0,57 \cdot 10^{-3}} = \frac{1754}{f} \text{°C} \tag{15.7}$$

If we assume $f = 160$ CFM/kW as we did above, $\Delta T_{peak} = 10,96 \approx 11$°C. This means that cooling with outdoor air is beneficial up to ambient temperature of $T_{peak} = T_H + \Delta T_{peak} = 32 + 11 = 43$°C, or almost anywhere in the world throughout the year. Beyond T_{peak} (very rare weather conditions), using outdoor air is not economically viable, and closed-circuit cooling should be used.

When outdoor air temperature approaches $T_{peak} = 43$°C, cooling with outdoor air does not provide energy benefits, while traditional closed-circuit cooling still has greater control over humidity and dust concentration, which makes it preferable until outdoor temperature drops in the evening.

15.5. Comparing Cooling Solutions

In this section, we compare capital and operating costs of the following cooling solutions:

1. (A) Cooling is performed with outside air only; standard direct expansion cooling system is installed in case of peak heat or air contamination, but is not operated;
2. (B) Same as the previous case, but the standard cooling system operates for 5% of the year, due to outdoor temperatures exceeding $T_H = 32$°C or air contamination;
3. (C) Standard direct expansion cooling system, operating continuously; cooling with outside air not implemented.

We assume the climatic conditions of the fictional site described above (and hence $\Delta T = 4$°C), power consumption of IT equipment $P = 1$ MW, and electricity price of $0,15 per kW·h (price for industrial consumers in Germany in May 2012 [29], converted to US dollars for consistency).

For our calculations we use direct expansion cooling unit "ACRD 500" produced by "APC" [8], with the cost of $17,800. This unit's maximum cooling capacity is 37 kW, and maximum power consumption is 16 kW. Variants (A) and (B) feature an under-provisioned cooling system of 10 units, designed only to cool outside air down to $T_H = 32$°C. Variant (C) uses a full-fledged cooling system operating all year round, and with electricity price that we assumed operating costs in 4 years are much higher than capital costs.

The analysis is simplified in that with (A) and (B) variants the capital and operating costs of air intake and mixing equipment – large fans that bring outside air into machine

	A	B	C
No. of direct expansion cooling units	10	10	27
Power consumption of cooling system, kW	160	160	432
Operating time per year, h	0	438	8,760
Energy consumption per year, kW·h	0	70,080	3,784,320
Capital costs of cooling units, $	$178,000	$178,000	$480,600
Operating costs of cooling units, per year, $	0	$10,512	$567,648
TCO of the cooling system in 4 years, $	$178,000	$220,048	$2,751,192

Table 15.2.: Comparison of capital and operating costs of cooling methods.

room, possibly mixing it with interior air in cold weather – is not accounted for. Another simplification is that costs of rooftop condensers, a required component of direct expansion cooling systems in all three variants, are also not taken into account.

As can be seen, using a standard cooling solution represented in variant (C) leads to the highest expense, compared to variants (A) and (B) that use outdoor air for cooling. Without the simplifications outlined above the difference would be less pronounced, though.

We can also supplement this analysis with capital and operating costs of the entire cluster computer. Based on calculations from Chapter 17, we conclude that a cluster computer designed for peak performance that consumes 1 MW of power has a TCO of roughly 32 million dollars. In other words, even the most expensive cooling system in variant (C) would represent only a small addition (7,8%) to the computer's TCO.

15.6. Liquid-based Cooling Methods

Using water or other liquids as the cooling medium at server level presents several benefits: reduced (or completely eliminated) vibration and noise in the machine room and reduced power consumption for cooling systems. Equally important is that the heat taken away by circulating water can be reused later. This allows to use the same amount of energy twice: first time for powering computing equipment, and second time for other useful purposes, such as heating buildings or greenhouses (see below). Benefits of using water cooling at server level should be weighed against risks of leakage that can result in damage.

"IBM" has been using water cooling since 1966, starting with their "System/360" mainframe computer [48]. In the standard servers market segment, "IBM" provides water-cooled "iDataPlex dx360 M4" server [47], such as the one used in "SuperMUC" computer [55]. Other vendors provide similar solutions: "Aurora Tigon" by Eurotech [30] (allowing inlet water temperatures in the range of 18..52°C), "RSC Tornado" by "RSC Group" [83], and a future water-cooled computer in the Netherlands to be installed by "Bull" in 2013 [16] (with inlet water temperature initially limited to only 35°C).

Hybrid cooling solutions that utilise both water and air have also been known for many years, recently being able to demonstrate remarkable efficiency. However, in this use case water does not reach temperatures high enough to allow heat reuse. For example, "Google" uses air-cooled servers, and heat is rejected by passing air through water-cooled

coils [35]. Heat is then removed from the water by various methods, and there is no information on successful heat reuse.

This technology allowed "Google" by year 2012 to attain power usage effectiveness (PUE) of 1,12, which signifies overhead of 12% for every unit of power spent by IT equipment. Cooling equipment power consumption represents just a share of those 12%, therefore we can safely assume that the target goal of power consumption for best-of-breed cooling systems should be on the order of 10% or less of the power consumed by the computing equipment they cool.

Despite the efficiency, Google's technology lacks the benefit of heat reuse. Additionally, concerns were raised [25] that discharging large amounts of heat into the Baltic Sea may have environmental impact, even though the return water is only slightly warmer than the inlet water.

Some components of water-cooled servers, such as power supply units, remain cooled by ambient air. The same holds true for storage and network equipment. This means that air conditioners are still necessary in the machine room, but their required cooling capacity is dramatically reduced, resulting in space and cost savings.

With regard to floor planning, we propose to establish two distinct zones in the machine room: one for legacy air-cooled storage and network equipment, the traditional environment with inherent noise and strong air flows, and one for the compute nodes which are primarily water-cooled. In the latter zone more friendly environmental conditions can be maintained, including reduced noise levels and light air flows. This equipment separation has effects on cable routing.

Technical and economic characteristics for a large-scale water-cooled supercomputer installation, SuperMUC [55], were obtained by Brehm *et al.* [15]. Compute nodes in SuperMUC are cooled by passing hot water (currently with inlet temperature of 40°C; up to 45°C is possible) through copper tubes and heat sinks attached to server components that generate the most heat. This allows to remove 90% of heat generated by the server; the remaining 10% is removed by air cooling, which is therefore much less intense than in traditional data centre environments.

Networking and storage equipment is located in racks equipped with rear door heat exchangers, but this requires cold water preparation. Below we estimate savings resulting from the use of hot water.

Pumping hot water through rooftop cooling towers allows to decrease its temperature in a free-cooling process by $\Delta T = 6$°C. Therefore, pumping 1 litre of water per second allows to remove this amount of heat:

$$Q = c_{water} \cdot m \cdot \Delta T = 25,1 \cdot 10^3 \text{ Joules}, \quad c_{water} = 4181 \text{ J/(kg} \cdot \text{°C)}, \tag{15.8}$$

which is equivalent to cooling capacity of 25,1 kW. Therefore, to remove 3 MW of heat (the power consumption of the entire SuperMUC machine), the required water flow is approximately 120 litres/sec, or 0,12 m^3/sec. According to [15], energy consumption for pumping hot water is 0,36 kW·h/m^3, while the analogous figure for cold water includes efforts required to cool the water to $T = 14$°C, and equals 1,46 kW·h/m^3.

Savings resulting from using hot water instead of cold water are 1,1 kW·h/m^3, and with the required water flow of 0,12 m^3/sec this corresponds to 0,132 kW·h/sec. With the electricity price of $0,15 per kW·h as used above, this translates to $1,711 per day and

$624,412 per year, or €468,000 per year. This mostly coincides with figures of savings in operating costs as quoted in [15].

Additional equipment needed for hot water cooling requires capital investment of €800K, and the total cost of ownership of SuperMUC is €83M (ibid). If cold water would be used instead of hot water for cooling, this would reduce capital expenditures by €800K, but raise operating expenditures by €468K each year. We can estimate that in 5 years this would lead to the increase in TCO from €83M to €84,54M. Therefore, hot water cooling provides approximately 1,86% reduction in TCO. Additional benefits can be realised if heat contained in the outlet water is reused, as described below.

15.7. Waste Heat Reuse

Andrews and Pearce estimate [5] that reusing 5,1 MW of waste heat from industrial processes allows to sustain a tomato greenhouse of 15,800 m^2, with yields of tomato crops of 735 tonnes per year. We then estimate [92] that reusing 3 MW of waste heat from SuperMUC would yield tomato crops of 432 tonnes per year. In 2009, per-capita supply quantity of tomatoes in Germany was 18,3 kg [32]. Therefore, the greenhouse utilising heat from SuperMUC can supply enough locally grown tomatoes to cover the needs of 23,000 people. The same idea of heat reuse applies to data centres of any nature.

The challenge is to create a heat reuse solution that would be (A) easily reproducible by greenhouse owners, (B) suitable for use in situations of heat supply from sources other than data centres, (C) scalable from several hundreds of kilowatts to several megawatts, and (D) tolerant to variations in heat supply (for example, at times when supercomputer is only lightly loaded with jobs, and therefore doesn't produce enough heat to completely support the greenhouse).

We propose the following plan of actions:

1. Use the monitoring system to understand how much energy consumed by the supercomputer can actually be captured and reused for agricultural purposes;

2. Understand the impact of work load level on heat output. This item is important because variation in heat output is quoted in [5] as a factor that prevented widespread adoption of industrial waste heat reuse since this technology was first used in late 1970s;

3. Perform feasibility studies for a range of European climates, particularly targeting northern climates where greenhouses are the preferred method of growing vegetables;

4. Create several designs of heat reuse systems for greenhouses, ranging in size and tailored to different climatic regions. Ideally, the system should be able to utilise heat from various sources, including industrial processes, and be easily scalable so that single-type blocks can be connected together to heat larger greenhouses.

The main components of the proposed system will be: (A) pipes to deliver water to parts of the greenhouse, (B) pumps and valves to control the flow of hot water, (C) monitoring

and control industrial automation system for pumps and valves for automatically maintaining optimal growing conditions, and (D) auxiliary heat supply (such as a gas burner) when main heat supply from a data centre is not sufficient.

15.8. Power Supply System

In this section we provide a literature review of power supply options. At a minimum, power supply system of computer clusters consists of power distribution equipment. In many cases uninterruptible power supply (UPS) is additionally required to facilitate clean shutdown of the computer in case of power failure, or to switch to a backup generator.

"SuperMUC" [55] uses a hybrid system that consists of both flywheel-based and battery-based UPS systems. Rooms for electrical equipment take up 18% of total data centre floor space, or roughly 100 m^2 per MW of power [15].

There exist alternative solutions that allow to avoid UPS systems altogether. "Google" equips their servers with individual batteries [85] to reduce the number of power conversions from AC to DC and back, additionally claiming financial savings compared to a centralised UPS system; however, this has not been verified.

"Bull" optionally installs so-called "ultra-capacitors" in their servers that protect from power outages of up to 300 msec [17]. This approach is only applicable with reliable electric power providers, but allows to manage without a UPS system, resulting in reduced capital expenditures and floor space as well as operating costs associated with maintenance of UPS systems such as replacement of failed batteries.

Proposals exist to minimise the number of power conversions and deliver direct current (DC) to servers. A study conducted by Ton *et al.* in 2006 demonstrated that providing 380 V DC power provided about 7% higher efficiency than a standard 208 V AC power distribution system commonly used in the USA [104]. However, a later 2008 study conducted by "The Green Grid Association" [103] found no significant difference in efficiencies of eight typical power distribution configurations in data centres, including supply of 380 V DC and 48 V DC power.

Finally, certain supercomputer installations are equipped with service and monitoring networks that monitor environmental parameters, allow to switch compute nodes on or off remotely, control the UPS system that delivers power to IT equipment, etc. When designing a power supply system, it is advisable to provide separate uninterruptible power for this network, because monitoring and control functions must be available even after main power failure.

15.9. Algorithm for UPS Design

Graph representation of configurations of technical systems that we introduced in Chapter 10 is general enough to be applicable to represent UPS systems as well. In this section, we propose a greedy algorithm for designing UPS systems.

We use a simple UPS model "Liebert APM" manufactured by "Emerson Network Power" [80] which is available in three configurations summarised in Table 15.3. The UPS system ships in a single rack, and free space in the rack can be used either for power conversion blocks (one to three blocks, 15 kW each) or for batteries. Therefore the configuration with

Power rating, W	Backup time, min	Heat output, W	Weight, kg	Cost, $	Cost per kW, $/kW
15,000	49	900	417	35,000	2,333
30,000	21	1,800	451	41,000	1,367
45,000	12	2,700	485	47,000	1,044

Table 15.3.: Configurations of *Liebert APM* UPS system.

the highest power rating has the lowest backup time, and vice versa. We represent the configurations with a graph shown in Figure 15.3.

Graph traversal proceeds from "Start" to "End", and there are three possible paths. One of them, corresponding to the power rating of 30 kW, is highlighted. Visiting each vertex along the path sets or updates values of corresponding characteristic. For example, there are three identical vertices describing three power conversion blocks that could be installed into the system. Each such block adds 15 kW of power capability, 34 kg to weight, 900 W to heat output (due to unideal UPS efficiency) and 6,000 dollars to cost. Traversing the highlighted path results in setting of the following values of characteristics:

```
ups_model=Liebert APM (up to 45kW)
ups_size_racks=1
ups_weight=451
ups_cost=41000
ups_power_rating=30000
ups_heat=1800
ups_backup_time=21
ups_cost_per_kw=1367
```

These characteristics describe the configuration completely; their values are further used in the computer cluster design procedure. Traversing all three available paths yields values of technical characteristics as they were summarised in Table 15.3. Now that we described how configurations become generated from the graph, let us proceed with the algorithm (see Algorithm 4).

The proposed algorithm uses as its input the minimum power rating of the UPS system to be designed, together with the set of UPS configurations, generated by traversing the graph as shown above. Each configuration is described mainly by its power rating P_i and cost C_i. Additionally, constraints on other technical characteristics, such as minimum backup time, can be supplied in A. The algorithm proceeds in the following way:

1. Create a copy of the input database and remove from it all configurations whose power rating is lower than P_{min} (line 1). Function $f(Set \mid C_1, C_2, \ldots)$ imposes constraints $C_1, C_2, \ldots$ on members of Set, removes members that do not satisfy the constraints, and returns the result.

2. If the resulting set is non-empty (line 2), then there exist configurations that can deliver the required power rating. Choose the configuration that brings minimum to C_i over the set $\mathbb{U}_c$ – that is, the cheapest configuration in this set (line 3);

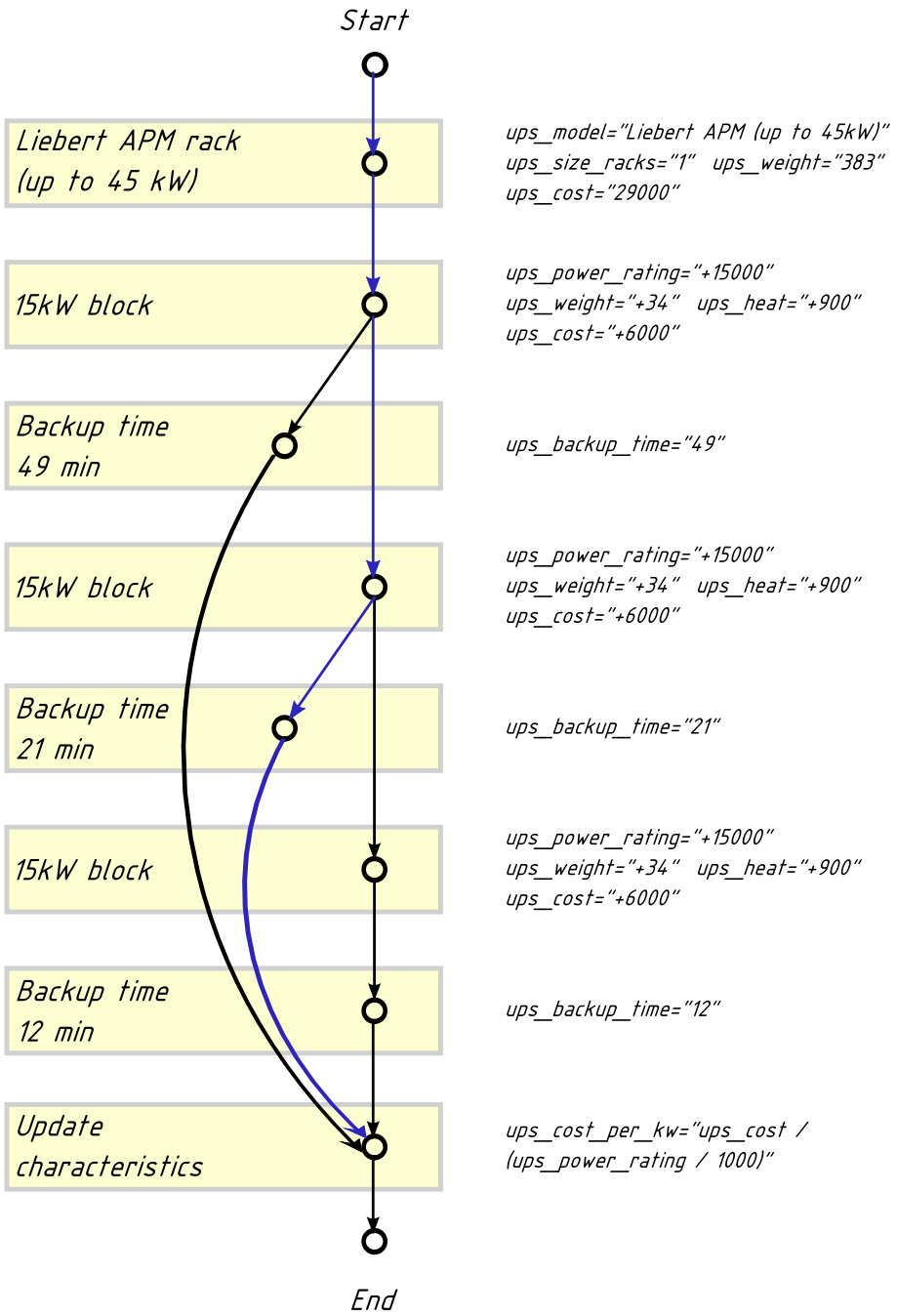


Figure 15.3.: Graph representation of configurations of *Liebert APM* UPS

Algorithm 4 Design a UPS system

Input:

P_{min}: Minimum power rating of the UPS system
$\mathbb{U} = \{\langle P_i, C_i \rangle\}$: Set of UPS configurations, where:
P_i: power rating of the configuration, and
C_i: cost of the configuration
A: Additional constraints, such as constraints on minimum backup time, etc.

Goal: Optimal UPS structure:

$\langle P_1, C_1 \rangle; \langle P_2, C_2 \rangle$: Configurations used for the optimal design
b_1, b_2: Number of UPS blocks of the first and second type
P: Resulting power rating of the entire system

1: $\mathbb{U}_c \leftarrow f(\mathbb{U} \mid P_i \geq P_{min}, A)$ {Impose constraints on power rating}
2: **if** $\mathbb{U}_c \neq \emptyset$ **then** {A single configuration is sufficient}
3: $U = \langle P_1, C_1 \rangle \leftarrow \arg\min_{\mathbb{U}_c} C_i$ { Choose the cheapest configuration }
4: $P \leftarrow P_1$
5: $b_1 \leftarrow 1; b_2 \leftarrow 0$
6: Exit
7: **else** {Employ a greedy algorithm}
8: $\mathbb{U}_c \leftarrow f(\mathbb{U} \mid A)$ {Impose additional constraints, if any}
9: $U_1 = \langle P_1, C_1 \rangle \leftarrow \arg\min_{\mathbb{U}_c} \frac{C_i}{P_i}$ { Choose a configuration with the lowest cost per kW }
10: $b_1 \leftarrow P_{min} \div P_1$
11: $P \leftarrow P_1 \cdot b_1; P_{rem} \leftarrow P_{min} - P$
12: **if** $P_{rem} > 0$ **then**
13: $\mathbb{U}_c \leftarrow f(\mathbb{U} \mid P_i \geq P_{rem}, A)$ {Impose constraints on power rating}
14: $U_2 = \langle P_2, C_2 \rangle \leftarrow \arg\min_{\mathbb{U}_c} C_i$ { Choose the cheapest configuration }
15: $P \leftarrow P + P_2$
16: $b_2 \leftarrow 1$
17: **end if**
18: **end if**

3. The overall power rating P of the designed UPS system then equals the rating of the selected configuration (line 4), and the UPS system consists of only one block (line 5);

4. However, we could end up with the set $\mathbb{U}_c$ being empty (line 7). This signifies that there do not exist configurations in the database that alone can provide the required power rating. Therefore, it will be necessary to utilise several (b_1) identical blocks to build a larger system, almost completely covering the required rating, and then possibly one more block to cover the remaining deficit.

5. For this purpose, we create a copy of the input set and impose any additional constraints specified by the user, but no constraints on power rating (line 8), Then we choose a configuration that brings minimum to the value of $\frac{C_i}{P_i}$ over the set $\mathbb{U}_c$ – that is, a configuration with the lowest cost per kW of power rating (line 9). This behaviour – choosing the locally optimal solution – is specific to greedy algorithms;

6. The rationale behind this choice is to obtain required power rating P_{min} for the minimal possible cost. In this particular case, the configuration U_1 with a power rating of P_1 was selected, and we need b_1 blocks with this configuration to almost completely cover our requirements in power rating (line 10);

7. The remaining deficit that needs to be covered, if any, is calculated in P_{rem} (line 11). We again create a copy of the input database, this time leaving only configurations whose power rating is enough to cover the deficit P_{rem} (line 13);

8. We then choose the cheapest of those configurations, U_2 (line 14). It's power rating P_2 is then added to the overall power rating P (line 15). Finally, we specify that one additional UPS block was required by returning $b_2 = 1$ (line 16).

Several examples of algorithm's output are given in Table 15.4. When the algorithm performs partitioning of the UPS system into blocks, we assume that each such block will power its part of IT equipment, and that a single UPS system with the required power rating is simply not available.

Greedy algorithms are not guaranteed to produce optimal solutions, but we believe that with real life equipment prices our algorithm will be able to design UPS systems with costs close to optimal (this should be verified on a larger database of equipment). For example, the power rating of 60 kW can be satisfied in two ways: 2*30 kW blocks or 45+15 kW blocks. Interestingly, with the (somewhat artificial) prices that we used for this modelling, both configurations would have the same cost and cost per kW.

However, the 2*30 kW configuration has a backup time of 21 minutes, while the 45+15 kW has a backup time of 12 minutes for the 45 kW part and 49 minutes for the 15 kW part. The overall backup time is determined by the lesser value, and is therefore 12 minutes. (Moreover, if a high backup time is required, such as 40 minutes, it can only be provided by multiple 15 kW configurations, despite their higher costs).

As shown in the table, the greedy algorithm chose the 45+15 kW configuration. But if additional constraints were specified on backup time (see constraints A in Algorithm 4), such as $T_{min} = 20$ minutes, then the algorithm would choose the 2*30 kW configuration.

Req. power rating, W	Resulting power rating, W	UPS partitioning	Backup time, min	Cost, $	Cost per kW, $/kW
20,000	30,000	1*30,000	21	41,000	1,367
40,000	45,000	1*45,000	12	47,000	1,044
60,000	60,000	1*45,000+ 1*15,000	12	82,000	1,367
80,000	90,000	2*45,000	12	94,000	1,044
100,000	105,000	2*45,000+ 1*15,000	12	129,000	1,229
150,000	150,000	3*45,000+ 1*15,000	12	176,000	1,173

Table 15.4.: Sample outputs from the UPS design algorithm

16. Equipment Placement and Floor Planning

In this chapter, we describe activities related to placement of hardware. We start with the discussion of partitioning strategies, then propose heuristics for placing equipment into racks. Finally, we develop an algorithm to calculate the size of floor space required to accommodate a given number of equipment racks.

16.1. Partitioning Strategies: Consolidation vs. Distribution

Hardware of cluster computers consists of blocks of different types, connected together according to a certain structure. Computing and storage hardware is connected to network equipment via communication cables, while the power supply system is connected to all other equipment via power cables. There are two alternative strategies for partitioning hardware into blocks: consolidation and distribution.

Consolidation strategy assumes using hardware blocks of the biggest possible size, while distribution strategy tries to build smaller, independent blocks. For example, with consolidation we can build one large UPS system capable of providing electric power to a large array of racks. Alternatively, we can divide racks into "islands" of one or a few racks, with each island served by its own independent smaller UPS system; this would be an example of distribution. These islands would now be able to operate independently with regard to power supply. We can further increase independence of blocks by providing a dedicated small uninterruptible power supply system to each compute node – such as a battery, as used by "Google" in its servers.

The strategy of distribution increases independence of hardware blocks – that is, they are capable to continue operation even if other blocks fail. This can be valuable for some workloads, such as those found in search engines, where unavailability of some search results is not critical. For most other parallel workloads, including HPC workloads, failure of one compute node often means that computation will need to be restarted. In such cases, extreme distribution and independence of individual compute nodes that it brings is less relevant; more important is reliable operation of a large block of equipment.

For example, a large UPS system can be designed with built-in fault tolerance mechanisms such as redundancy, providing power to the whole cluster computer. Large technical systems enjoy economy of scale, and therefore a large UPS is likely to be more economical than a thousand small UPS systems of comparable power supply capability. One of the reasons is that a single large battery is easier and cheaper to manufacture than, say, ten small batteries of the same aggregated capacity. In this example, consolidation results in savings.

Similar arguments of consolidation and distribution apply to network hardware: we can design a giant switch with 3,456 ports; one such InfiniBand switch was once built by Sun Microsystems, and another design was proposed by Farrington *et al.* [31]. Alternatively,

we can use a traditional approach with 36-port top-of-rack switches for the edge level of the network, and bigger switches for the core level. In this case, however, a "consolidated" 3,456-port switch might not be cheaper because its parts are not mass-produced. Also, because of the vast quantity of cables that connect to such switch, it is somewhat unwieldy to install and maintain.

16.2. Distributed Structure for High Survivability

In certain cases, independence of blocks is attractive because of improved survivability. Consider a military anti-missile defence system that includes a radar and a cluster computer with several spatially dispersed blocks capable of independent operation. All blocks can work together as a single cluster computer, delivering highest performance and therefore highest precision in calculating missile coordinates. If some blocks are destroyed, the remaining blocks can continue operation, although in degraded mode with lower precision, which might still be enough to destroy the missile.

In this situation, it is beneficial to design the cluster computer in such a way that it can operate as a whole, and at the same time each of constituent blocks (say, a rack of equipment) can operate independently. When some blocks are destroyed or some communication links are lost, the remaining blocks can continue operation as a single computer of degraded configuration. In the worst-case event, the only remaining single block can still work independently.

This approach requires to build infrastructural systems of the cluster computer in a distributed rather than in a consolidated way. Each block must be provisioned with all necessary infrastructure: power supply, network and local storage systems; in this case it will be capable of operating independently from the other blocks.

Some infrastructural systems within a block may be more "distributed" than the others. For example, suppose that the independent block consists of two racks of compute nodes. We can have a single network switch serving both racks – that is, a consolidated solution. At the same time, we can have a single UPS supplying power to both racks (a consolidated solution), or if it deems appropriate, we can have two independent UPS systems, one for each rack (a distributed solution); finally, we can even have separate batteries in each compute node (an even more distributed one).

However, this increased extent of distribution in the power supply system doesn't by itself guarantee increased reliability of power supply for the block as a whole. It only brings independence of power supply of racks or compute nodes within a block, which may be of little relevance: the two racks within the block cannot operate completely independently, as we still have a single network switch per block. In other words, infrastructural systems *can* have different degrees of distribution, but it may not have practical benefits.

16.3. Equipment Placement Heuristics

Strategy of partitioning equipment into blocks has a profound effect on what equipment comprises a block, as well as the structure of connections within a block and between blocks. However, when a specific structure of a block has been decided upon, the next step can be performed: placing chosen equipment into racks and locating racks on the

floor. In this section, we propose heuristics for placing compute and network equipment into racks. The heuristics are applicable in the case when there is a block of compute nodes connected to a single network switch, and all items of equipment should preferably remain physically close to each other. This is the case of fat-tree topologies and indirect torus topologies.

Mudigonda *et al.* [63] propose a straightforward cable routing algorithm which relies on a prior placement of nodes and switches into racks in a manner that minimises total cable length. Such positioning resembles a knapsack problem, so a number of heuristics were introduced in the cited paper.

We generalise their findings by considering equipment items of differing physical sizes, and propose our own heuristics. We show below that fat-trees are perfectly suitable for packing racks as densely as possible, or for leaving as much blank space in every rack as necessary (e.g., for other equipment), and also allow for a smooth transition between these extreme cases. After running the network design algorithm, the number and types of required switches are obtained, hence the size they occupy becomes available. The size of compute nodes becomes known at an even earlier stage. Therefore equipment placement can begin using the following heuristics.

1. Modular switches are physically indivisible and should be placed first. Otherwise one may end up with partially filled racks, with no space in any of them enough to house relatively big modular switches, and new empty racks would be required.

2. Space for other indivisible equipment can be reserved in the same manner.

3. The principal "building block" of a fat-tree network is an edge switch and nodes connected to it. It is logical to keep this switch and its nodes in the same rack (with blade servers, this occurs by itself). They are connected with relatively short cables. One or more such building blocks are added to a rack until one of the following budgets is exceeded:

   a) The remaining free space in the rack cannot accommodate another building block;

   b) Weight budget of the rack, stipulated by the floor load limit, is exceeded (important mainly for raised floors);

   c) Power consumption of equipment in the rack goes above capacity of the power supply or cooling systems.

   Now we can't add the next new block to the rack, but there could still be an opportunity to add individual compute nodes from that block. We calculate the remaining budget of all characteristics (space, weight, power, cooling, etc.) for the current rack, and estimate the number of compute nodes that can be placed in this rack at a later time, and then proceed to the next rack.

4. The previous step – placing blocks into racks – is repeated several times, until we have enough semi-filled racks to accommodate the next block by "spreading" it among these racks.

This strategy allows to fill racks as densely as possible, but results in irregular wiring patterns, as noted by other researchers. However, minimising the number of racks not only saves floorspace, it also allows to decrease the length of cables running between distant racks. If there is no goal to save space, this heuristic could be omitted.

Example 16.1 *When using commodity 1U InfiniBand switches with $P = 36$ ports and 1U compute nodes, a building block consists (in the case of a non-blocking network) of one switch and 18 nodes, for a total of 19U. Two such blocks can be placed in a standard 42U rack, and 4U of blank space will remain. After filling five racks in this manner, the resultant 20U of space are enough to house the next 19U block.*

5. Although an edge switch and its nodes should preferably be kept in a single rack, they don't necessarily have to be adjacent. In fact, other researchers recommend to place edge switches to the top of the rack (hence the colloquial name: "top-of-rack switches"). This decreases the length of cable bundles that run between racks, potentially enabling the use of shorter cables.

6. Racks in a row are filled until the end of a row in a machine room is encountered. The next rack to be filled is chosen across the aisle that separates rows.

   This behaviour ensures that parts of the next block to be spread among several racks will remain close to each other. Placement proceeds in a serpentine pattern until all equipment is placed.

These heuristics are general enough to enable placement of compute nodes and switches of differing physical sizes into racks of differing heights even within one installation. After placement is complete, a cable routing algorithm can be run to calculate cable lengths and hence costs.

We present an example of using these heuristics in Figure 16.1, deliberately demonstrating filling the racks as densely as possible. In this example, we need to design a non-blocking fat-tree network for $N = 396$ compute nodes, using twenty-two 36-port edge switches and two 198-port core switches. We then place all computing and network equipment into standard 42U racks. (Note: this example serves demonstration purposes. In a real life situation, instead of a multitude of interconnected switches, a single large modular switch would be preferred to simplify cabling and improve reliability).

The large indivisible blocks of equipment are installed first. Then, we install blocks of compute nodes, and put edge switches for the blocks to the top of corresponding racks. Finally, after installing seven such blocks into four racks, we have enough spare space in the racks to place one more block of compute nodes; however, this time it has to be "spread" among all four racks (the exact location of the corresponding edge switch for this block can be chosen arbitrarily). Two units of space in the rightmost rack are empty and can be used further. Overall, the first four racks could accommodate 8 blocks of equipment out of 22.

Figure 16.2 further presents the top view of 14 racks, arranged in two rows. The first four racks from the upper row correspond to those in Figure 16.1. Rack filling starts in a clockwise direction. First, the indivisible equipment is placed; we arbitrarily chose to place block $A1$ into rack 1, and two core switches, 10U each, into rack 5. Then we start placing

Figure 16.1.: Front view of four racks filled using the proposed heuristics. There are seven contiguous blocks of compute nodes; their edge switches are placed in top positions of corresponding racks. There is also the 8th block which is "spread" among the four racks.

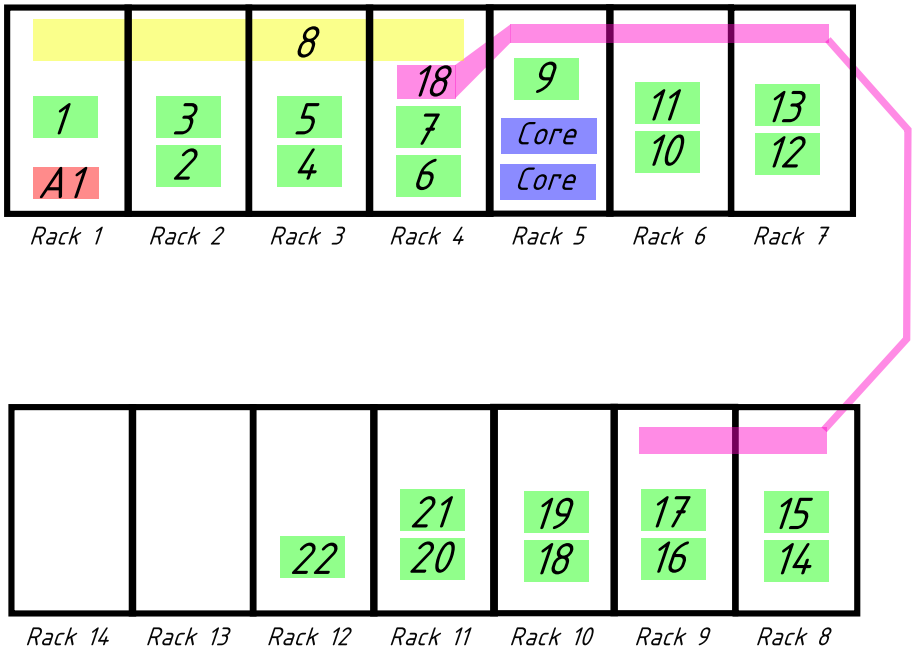


Figure 16.2.: Top view of 14 racks filled using the proposed heuristics. The first four racks in the top row are racks from Figure 16.1. Boxes and bands, drawn not to scale, represent blocks of equipment; numbers are box IDs. Blocks 8 and 18 are "spread", other blocks are contiguous.

blocks – compute nodes and their edge switches. The first 7 blocks are contiguous, and the 8th block is spread among the four racks, as described above.

As rack 5 contains two 10U core switches, only one contiguous block 9 could be placed into it. Contiguous blocks 10 to 17 are placed into racks 6 to 9. At this moment, racks 4 to 9 contain 21U of empty space, which is enough to place block 18, spread among these racks. As can be seen from the figure, this block crosses the aisle that separates rows; the corresponding edge switch can be placed into either row. Blocks 18 to 22 are placed into racks 10, 11 and 12.

The final placement occupies 12 racks. This dense placement approach is not very elegant because of the irregular wiring patterns in spread blocks. However, if we didn't use the strategy of block spreading, blocks 8 and 18 would have to be placed into racks 12 and 13, so using strictly regular wiring patterns would cost us one additional rack.

16.4. Algorithm for Floor Planning

After equipment has been placed into racks, the number of racks becomes known, and we can position them on the floor. During automated design procedures, we need to quickly calculate the required floor space size for each configuration of the cluster computer. Floor space size affects total cost of ownership (TCO) of the supercomputer, because this space must be either constructed (resulting in capital expenditures) or rented (resulting in operating expenditures).

However, the calculation of the required floor space size must be quick enough to be applicable for automated design workflow. In this section, we propose a simple algorithm that calculates floor space size, based on the inputs such as the number of racks, rack dimensions, and clearances around the racks.

A typical arrangement of two rows of racks and clearances around them are shown in Figure 16.3. Typical rack dimensions are depth $d = 1{,}07$ m and width $w = 0{,}6$ m; these values are for rack model "AR3100" made by APC. Specification for the "Open Rack" [73], proposed by the "Open Compute Project", stipulates the same width, while the depth of the rack can range from 0,35 to 1,22 m, to accommodate different types of equipment. In IBM's "iDataPlex" rack [45], depth and width are swapped compared to the standard rack: $d = 0{,}6$ m and $w = 1{,}2$ m.

In the figure, w and d denote rack width and depth, respectively, c_f is a front (and rear) clearance, c_s is the side clearance, and c_a is the aisle width. l_x and l_y are machine room dimensions.

Values for clearances depend on building construction codes and cooling requirements: air cooling using cold air supplied through raised floors is known to require wide inter-rack aisles to ensure optimal air flow; with water cooling this precaution is not required. In any case, the aisle width c_a must be large enough to allow pulling equipment out of the rack; in practice this means the aisle width must be at least as large as the rack depth.

The algorithm we are proposing calculates the required floor space size by placing R racks into r_y rows (that is, in y dimension), with r_x racks in each of them, and with clearances c_f and c_s around rows and aisles c_a between the rows. The main idea behind the algorithm is that the resulting floor area should be roughly square.

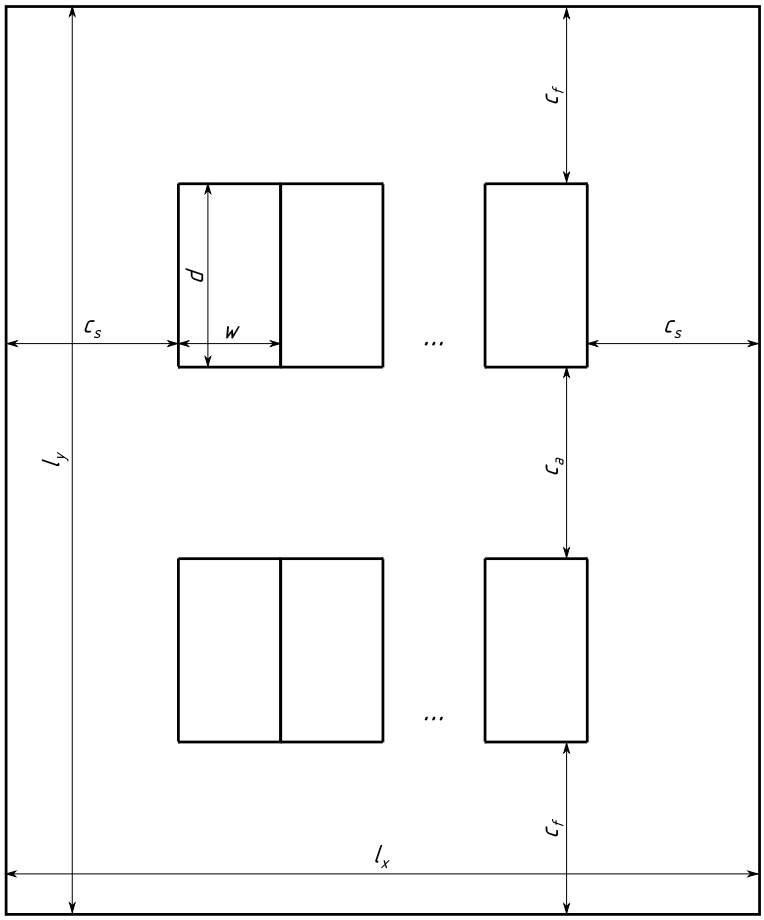


Figure 16.3.: Placement of two rows of racks on the floor, with clearances.

There is another consideration that we must pay attention to. To facilitate easy movement of personnel in a machine room, very long rows should be broken into smaller segments; we denote the length of contiguous segments as l_{xc}, and the distance c_s between the segments should be maintained. A rational value for the segment length is $l_{xc} = 6..10$ m.

The algorithm (see Algorithm 5) works in two stages: on the first stage, it calculates rough approximations for r_x and r_y, the number of racks per row and the number of rows, respectively, based on the assumption of the square shape of the machine room. On the second stage, it refines room's dimensions to its final values.

If there are r_x racks per row, then the length l_x of the machine room is $r_x w$, plus a correcting term $\frac{r_x w}{l_{xc}} c_s$ that accounts for gaps between contiguous segments or racks within a long row, plus two side clearances c_s:

$$l_x = r_x w + \frac{r_x w}{l_{xc}} c_s + 2c_s \tag{16.1}$$

On the other hand, if there are r_y rows of racks in the room, then the width l_y of the room is the depth of all racks $r_y d$, plus the width of aisles $(r_y - 1)c_a$, plus two front clearances c_f:

Algorithm 5 Calculate the arrangement of racks on the floor

Input:
R: Number of racks to place on the floor
w, d: Rack width and depth
c_s: Side clearance
c_f: Front and rear clearances
c_a: Aisle width
l_{xc}: Maximal length of a contiguous segment of racks

Output:
r_x: Number of racks per row
r_y: Number of rows
{ Floor sizes: }
l_x, l_y: Floor length and width
f: Floor space
{ Components of the "floor space formula": }
r_{bs}: Number of racks in big segments
r_{ss}: Number of racks in the last small segment
n_b: Number of big segments

1: { Coefficients of the quadratic equation: }
2: $a \leftarrow d + c_a$; $b \leftarrow 2c_f - 2c_s - c_a$; $c \leftarrow -Rw(1 + \frac{c_s}{l_{xc}})$
3: $D \leftarrow b^2 - 4ac$ { Discriminant }
4: $r_{y_1} \leftarrow (-b + \sqrt{D})/(2a)$; $r_{y_2} \leftarrow (-b - \sqrt{D})/(2a)$ { Roots }
5: $r_y \leftarrow$ round $(\max(r_{y_1}, r_{y_2}))$ { Number of rows }
6: $l_y \leftarrow r_y d + (r_y - 1)c_a + 2c_f$ { Floor width }
7: $r_x \leftarrow \lceil R/r_y \rceil$ { Number of racks per row }
8: $s \leftarrow \lceil r_x w / l_{xc} \rceil$ { Number of contiguous segments }
9: $g \leftarrow s - 1$ { Number of gaps between segments }
10: $l_x \leftarrow r_x w + g c_s + 2c_s$ { Floor length }
11: $f \leftarrow l_x l_y$ { Floor space }
12: $r_{bs} \leftarrow \lceil r_x / s \rceil$ { Number of racks in big segments }
13: $n_b \leftarrow r_x \div r_{bs}$ { Number of big segments }
14: $r_{ss} \leftarrow r_x - n_b \cdot r_{bs}$ { Number of racks in the last small segment }
15: **print** Floor space formula: $(n_b \cdot r_{bs} + r_{ss}) \cdot r_y$

$$l_y = r_y d + (r_y - 1)c_a + 2c_f \tag{16.2}$$

Room shape is supposed to be square, so we can equate the two expressions ($l_x = l_y$), and additionally the total number of racks in the room must be equal to the input value R specified by the user; this gives us a system of equations:

$$\begin{cases} r_x w + \frac{r_x w}{l_{xc}} c_s + 2c_s = r_y d + (r_y - 1)c_a + 2c_f \\ r_x r_y = R \end{cases} \tag{16.3}$$

We use the second equation to express r_y, and substitute it into the first equation. This leads to a quadratic equation:

$$r_y^2(d + c_a) + r_y(2c_f - 2c_s - c_a) - Rw\left(1 + \frac{c_s}{l_{xc}}\right) = 0 \tag{16.4}$$

It can be solved with respect to r_y (lines 1..4). There are two roots; we take the positive one, and round it to the nearest integer (line 5).

We further use the obtained value of r_y as the final number of rows in the machine room. Additionally, this immediately yields the width l_y of the machine room through equation (16.2), see line 6. We then determine the number of racks per row in such a way that the total number of racks in the room, $r_x r_y$, is at least as big as R racks requested by the user: $r_x = \lceil R/r_y \rceil$ (line 7).

Now, we need to calculate how many contiguous segments of racks there will be in a row. There are a total of r_x racks in a row, each with a width of w, and the maximal allowed segment length is l_{xc}, hence the number of segments is $s = \lceil r_x w/l_{xc} \rceil$ (line 8). The number of gaps between the segments is $g = s - 1$, and the width of each gap is the side clearance c_s, hence the width of empty space between all segments is gc_s. This allows to calculate the final length l_x of the machine room (line 10):

$$\begin{aligned} s &= \lceil r_x w/l_{xc} \rceil \\ g &= s - 1 \\ l_x &= r_x w + g c_s + 2c_s \end{aligned} \tag{16.5}$$

At this moment, floor space size can be calculated and reported to the user: $f = l_x l_y$ (line 11). It only remains to give a hint to the designer as to how the racks will be segmented in the row (lines 12..14). Big segments will have $r_{bs} = \lceil r_x/s \rceil$ racks; there will be $n_b = r_x \div r_{bs}$ such segments. The last segment is smaller or even empty, it has $r_{ss} = r_x - n_b \cdot r_{bs}$ racks.

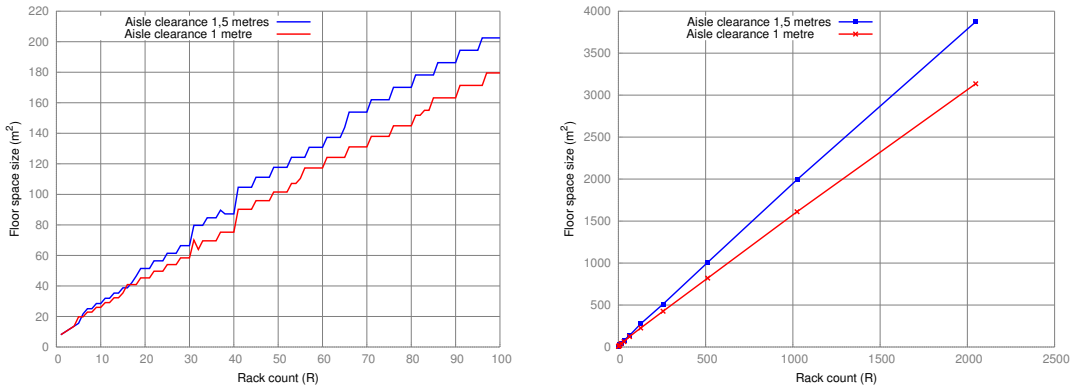
The obtained values allow to build a "floor space formula" that unambiguously defines the location of racks in the machine room: $(n_b \cdot r_{bs} + r_{ss}) \cdot r_y$. A sample formula is $(2 \cdot 9 + 7) \cdot 8$, it describes arrangement of $R = 200$ racks in 8 rows, with 25 racks per row. Each row has three segments: two long (9 racks) and one short (7 racks).

We test the proposed algorithm by calculating floor space size required to place up to $R = 2{,}048$ racks. We use racks with width $w = 0{,}6$ m and depth $d = 1{,}1$ m, and clearances $c_s = c_f = 1$ m. The maximal length of a contiguous segment of racks in a row is $l_{xc} = 6$ m. Two aisle widths were used: $c_a = 1$ m and $c_a = 1{,}5$ m. Experimental results are shown in Figure 16.4.

As can be seen, at $R = 100$ racks, extending the aisle width c_a from 1 to 1,5 m leads to increase in the overall machine room size by only 13%; at $R = 2{,}048$ racks the increase reaches roughly 24%.

The larger the installation, the less is the per-rack space; for instance, in the case of $R = 2{,}048$ racks and $c_a = 1$ m aisles, the per-rack space is 1,52 m^2. However, a single rack physically occupies only $w \cdot d = 0{,}66$ m^2, which means that "empty spaces", used mainly for movement of personnel, take up 130% of space occupied by equipment itself.

The proposed algorithm is simple, and for some input values it produces sub-optimal solutions. For example, for $R = 31$ racks it suggests a 11×3 arrangement, resulting in 33 racks and $f = 70{,}08$ m^2, while the more optimal arrangement would be 8×4, with only 32 racks and $f = 63{,}92$ m^2. However, in the vast majority of cases, it produces accurate results. Another feature of the algorithm is that it is very quick, and its run time does not depend on the number of racks to be placed.



(a) For up to $R = 100$ racks

(b) For up to $R = 2,048$ racks

Figure 16.4.: Floor space size calculated by the proposed algorithm

17. Practical Evaluation of the Algorithm

The goal of this chapter is to evaluate the cluster design algorithm proposed in Chapter 11, using the prototype CAD system described in Chapter 12. For a start, we will analyse technical and economic characteristics of individual compute nodes, and then proceed to characteristics of computer clusters built using these nodes.

The flexibility of our approach allows to calculate any of the following characteristics: (a) characteristics of individual components of a computer system, assigned to vertices of the configuration graph (e.g., `cpu_cores="8"`), (b) characteristics evaluated via arithmetic expressions in the configuration graph (e.g., `node_cost="+349"`), and (c) characteristics evaluated inside the CAD tool (such as power consumption of the entire machine, or its volume of equipment).

We will also examine how cluster characteristics, including capital and operating expenditures, are affected by factors such as performance goals (different workloads and different levels of performance) and interconnection networks characteristics (topology and blocking factor).

17.1. Overview of Equipment

For the purpose of analysis, we employ a configuration graph that generates 264 configurations of the Hewlett-Packard's "BL465c G7" server. The configuration graph is shown in Figure 10.4 in Chapter 10.

The hardware that we use for analysis is of the previous generation – that is, a newer line of servers and CPUs is already available on the market. We deliberately use this outdated hardware because its prices have stabilised, and are not going to change any more. This allows to use our fixed set of hardware as a reference dataset, without the risk of different researchers obtaining different results because of variations in prices. Complete characteristics of hardware and its prices are given in Appendix B.

The "BL465c G7" server can be fitted with 18 possible CPU models, 10 of them are AMD Opteron 6100 series processors (outdated but included for the sake of completeness) that can be coupled with 5 configurations of PC3-10600 DRAM memory. This gives 50 possible configurations of a compute node; but the node can have either one or two CPUs, so this number is doubled to 100 configurations. Another 8 CPU models are from AMD Opteron 6200 series, and these can be coupled with two configurations of faster PC3-12800 DRAM memory, giving 16 configurations, and this number is again doubled depending on whether the second CPU is installed or not, leading to 32 configurations.

In total, these components give 132 configurations. Then, the InfiniBand network adaptor may or may not be installed into the compute node, thereby again doubling the number of configurations to 264.

Compute nodes are connected via an interconnection network with fat-tree (Chapter 13)

Characteristic	Minimal value	Maximal value
CPU clock frequency, GHz	1,6	3,0
CPU cores per compute node	8	32
CPU L3 cache size, MBytes	12	16
RAM size per compute node, GBytes	32	128
RAM size per CPU, GBytes	32	64
RAM size per CPU core, GBytes	2	8
Number of occupied DIMM slots	2	16
Power, W	166	329
Peak floating-point (FP) performance, GFLOPS	64	294,4
Compute node cost, $	2,756	12,796
Cost to peak FP performance ratio, $/GFLOPS	17,5	108,6

Table 17.1.: Extreme values of characteristics across 264 configurations of individual compute nodes.

or torus (Chapter 14) topologies, using InfiniBand QDR switches. Types and characteristics of the switches are listed in Table 14.3 in Chapter 14. (Please note that InfiniBand switches are used also when compute nodes are connected via their Ten Gigabit Ethernet adaptors. This is because prices for corresponding Ten Gigabit Ethernet switches were not available, and we used InfiniBand switches instead. This circumstance is unlikely to have much impact on the outcomes of our analysis because per-port prices for both types of switches are comparable).

Compute nodes and network equipment are powered using uninterrupted power supply (UPS) devices available in three configurations; those configurations and their characteristics are listed in Table 15.3 in Chapter 15.

17.2. Characteristics of Individual Compute Nodes

The CAD tool reads the configuration graph stored in XML files, traverses it, and generates the list of configurations in a comma-separated values (CSV) file; each configuration is represented as a set of fields separated by commas. It is interesting to note that technical and economic characteristics across the configurations are vastly different, see Table 17.1.

All CPU models have a peak performance of 4 floating-point operations per cycle. The minimal peak floating-point (FP) performance of 64 GFLOPS is obtained on configurations with one 8-core CPU running at 2 GHz, the maximal performance of 294,4 GFLOPS is obtained on configurations with two 16-core CPUs running at 2,3 GHz. The difference in peak performance across the configurations is therefore 4,6 times.

By chance, cost difference has a similar value, 4,64 times. The cheapest configuration is a uniprocessor node with one 8-core CPU running at 2,6 GHz, with 32 GBytes of memory and without the InfiniBand network adaptor. The most expensive configuration is a dual-processor node with two 12-core "AMD Opteron 6176" CPUs running at 2,3 GHz, with 128 GBytes of memory and with the InfiniBand adaptor. The latter configuration by no means can be considered optimal; it is overly expensive, mostly because of its outdated

and overpriced CPUs. There is a better CPU model, "AMD Opteron 6276", with 16 cores instead of 12 and 16 MBytes of L3 cache memory instead of 12 MBytes, running at the same clock frequency of 2,3 GHz – and at the same time *less* expensive despite its better characteristics.

When components become outdated, they also become hard to source, and their price can rise due to market laws. At the same time, newer components with better functionality and price may already be available. In general, one should avoid having outdated components in the database; this reduces the number of generated configurations and speeds up analysis. In any case, even if these overpriced components are present in the database, the CAD tool will mark them as non-optimal during analysis.

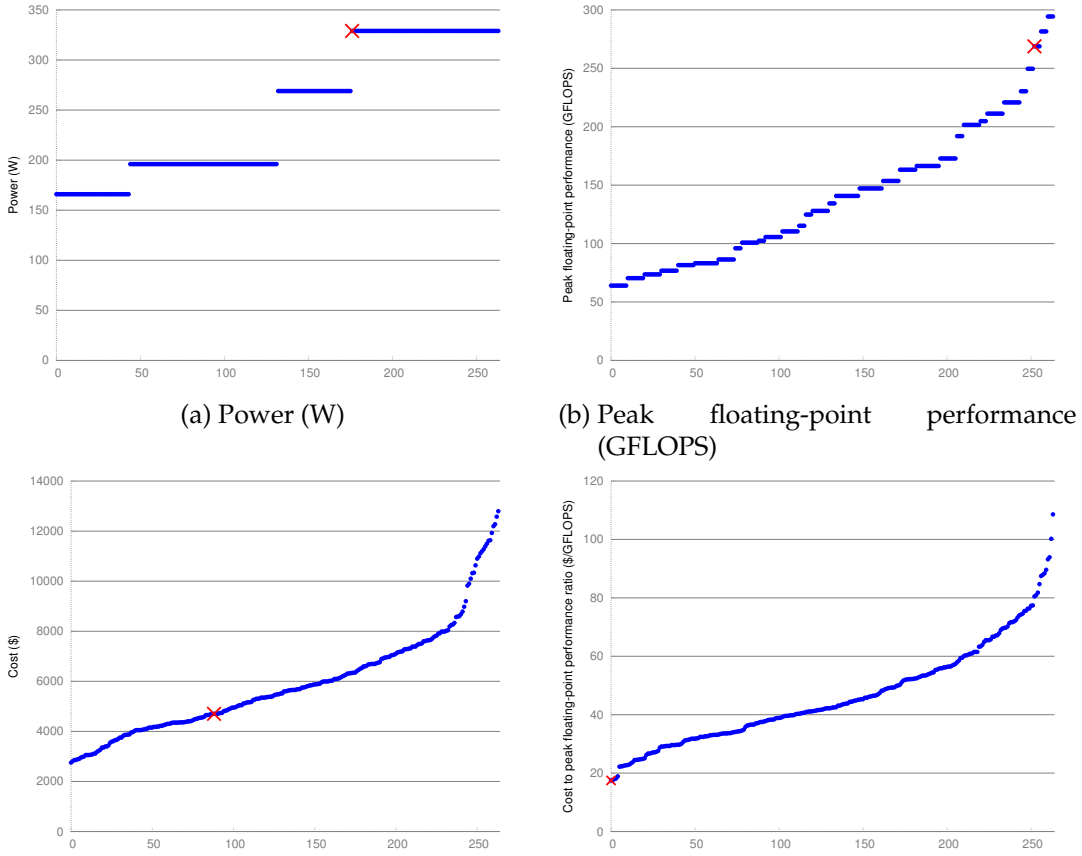
We further compute the "Cost/Peak FP performance" ratio for each configuration, thereby finding a "locally optimal" configuration (that is, optimal on the first stage of the design process, where configurations are generated, and no detailed analysis has been performed yet). This metric is also used as a heuristic to filter out unpromising configurations (see Chapter 8). The difference in the value of this metric across the configurations is even more impressive and reaches 6,2 times. The lower the value of the metric, the better; the lowest value (17,5 $/GFLOPS) is obtained on a dual-processor server with 16-core CPUs running at 2,1 GHz, with 32 GBytes of memory and without the InfiniBand network adaptor.

This configuration costs $4,701; on the charts in Figure 17.1 it is marked with a cross. However, we have to mention that it is not necessarily a *globally optimal* configuration. It is because the metric takes into account only peak floating-point performance of a single compute node: at the early design stage, when a configuration has just been generated but a performance model has not been invoked yet, no performance figures other than peak performance are available. There is another configuration, which is similar to the listed one but does include the InfiniBand adaptor. This latter configuration has a higher value of the "Cost/Peak FP performance" ratio due to its higher cost, and therefore is not locally optimal, but it could be more beneficial for workloads that require intensive communications, although this can only become known after the performance modelling stage.

The highest (and hence the worst) value of the metric, 108,6 $/GFLOPS, is obtained on a uniprocessor server with the 8-core CPU running at 2 GHz, with 64 GBytes of memory and the InfiniBand adaptor; this configuration costs $6,951. As can be seen, the most and the least optimal (in the sense of this metric) configurations are neither least nor most expensive, which means that the cost of a compute node cannot be used to make judgements about optimality of its configuration.

Our next step is to graphically represent the distribution of the bottom four characteristics listed in Table 17.1 using charts. Before plotting each chart, the list of configurations is sorted according to the value of the characteristic being inspected, therefore the order of configurations is not preserved across the charts.

The chart for power consumption (Figure 17.1a) has coarse steps; this is explained by the lack of detailed data on power consumption of compute node components. Indeed, the biggest contributors in power consumption are CPUs, and the only data available to us is the "thermal design power", or TDP, which is simply the typical amount of power consumed by CPUs under load as assigned by the manufacturer, whereas actual power consumption can differ. In this chart we only have four different levels of power consumption: the CPUs that we have in the database have TDP of either 85 W or 115 W, and



(a) Power (W)

(b) Peak floating-point performance (GFLOPS)

(c) Cost ($)

(d) Cost to peak floating-point performance ratio ($/GFLOPS)

Figure 17.1.: Characteristics of 264 configurations of individual compute nodes (order of configurations is not preserved across charts). The cross denotes the configuration with the minimal value of "Cost/Peak FP performance" ratio.

the node can have either one or two CPUs, hence the four levels. Other components in our database that contribute to node's power consumption are RAM and motherboard; due to the lack of detailed data, these are considered constant across all configurations and thus do not create "sublevels" on the chart.

The next chart (Figure 17.1b) depicts peak floating-point performance of configurations in GFLOPS. As we have a wide variety of CPU models, each with its own number of cores and clock frequency, and additionally there is an opportunity to install either one or two CPUs into a compute node, our configurations have many different levels of peak performance.

Intuitively, configurations with a higher peak performance are preferable for floating-point intensive workloads, but only if they are not overpriced. The locally optimal configuration (with the lowest value of "Cost/Peak FP performance" ratio, marked with a cross) turns out to have one of the highest values of performance across all configurations, but still not the highest. That is, there are other configurations with even higher peak perfor-

mance, but their price makes them less attractive.

The chart that depicts cost of configurations is given in Figure 17.1c. As can be seen, there are very affordable as well as very expensive configurations, and the optimal one is in the lower part of the cost range.

Finally, the chart in Figure 17.1d shows configurations sorted by their cost to performance ratio, and the optimal one is obviously the first. This chart serves as a visual aid for understanding the heuristic that we proposed in Chapter 8; indeed, the value of the metric rises rather steeply, so it only makes sense to examine at most the first 20% of configurations.

17.3. Designing for Peak Performance Requirements

Now that we examined technical and economic characteristics of individual compute nodes, we can proceed with characteristics of complete computer clusters built using those compute nodes, together with network and UPS equipment.

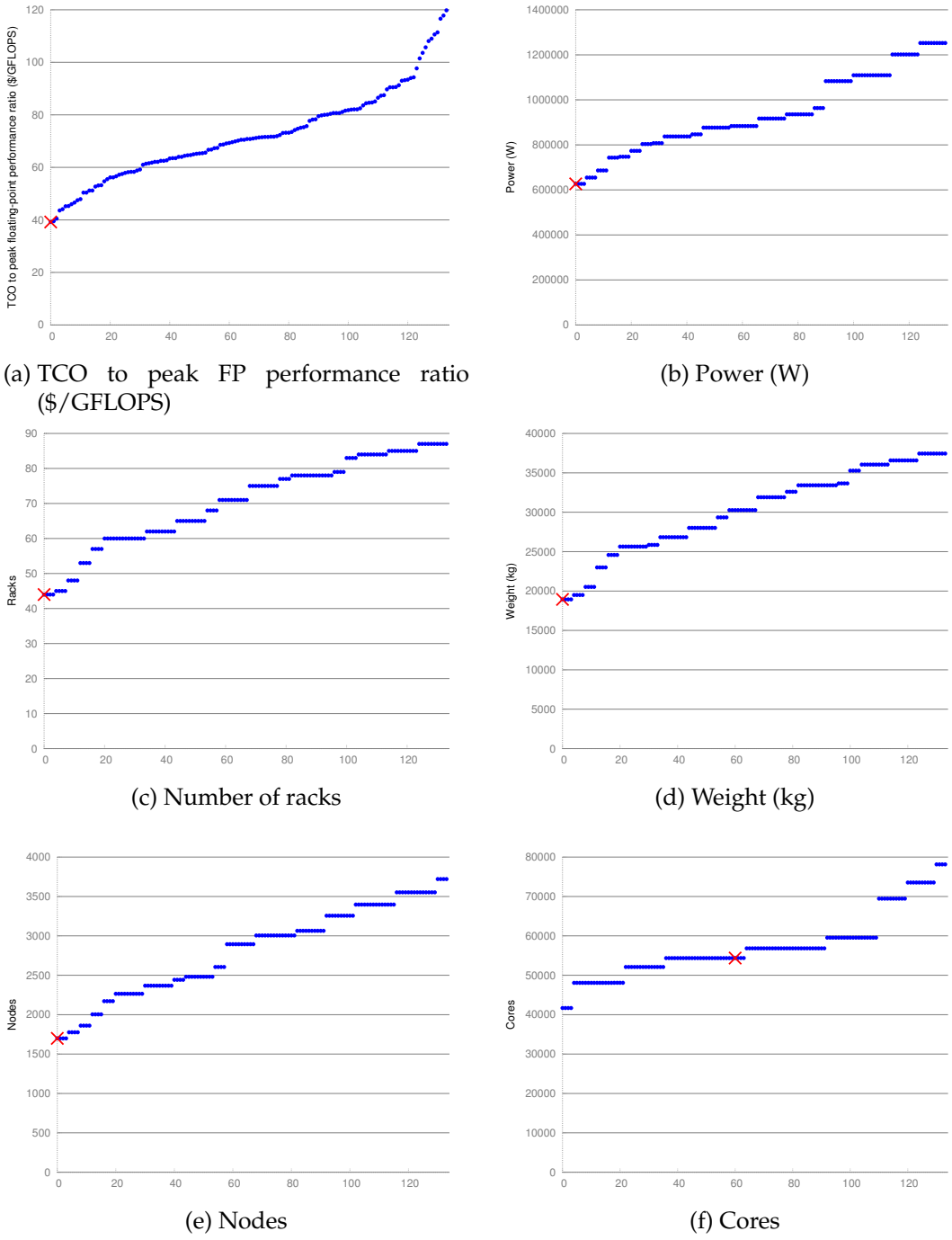
To calculate TCO of solutions, we will use prices [37] advertised by "Hetzner Online AG", the German co-location and hosting provider: electricity price of €0,29 per kW·h, or about $0,37 per kW·h, and data centre space rental costs of €199 per rack per month, or about $257 per rack per month. The lifetime of the machine will be taken at 3 years.

In this section we focus on designing computer clusters based on requirements of peak floating-point performance. In particular, we will design a cluster with peak performance of 500 TFLOPS, using a non-blocking fat-tree network and a UPS system with a minimum of 10 minutes of backup time.

In our database of network hardware we have edge switches with $P_E = 36$ ports and core switches with up to $P_C = 216$ ports (see Table 14.3). Therefore we are limited to building non-blocking fat-tree networks with up to $N_{max} = P_E \cdot P_C/2 = 36 \cdot 216/2 = 3{,}888$ nodes. This, in turn, means that not all configurations of compute nodes are suitable: to attain a 500 TFLOPS level of aggregate peak performance, individual compute nodes must have peak performance of at least $500{,}000/3{,}888 = 128{,}6$ GFLOPS. (Trying to design, say, a 1 PFLOPS cluster instead of a 500 TFLOPS one would further limit available configurations, making our analysis less insightful).

Only 134 of 264 configurations meet this constraint, and the CAD tool disregards the rest, because no fat-tree network can be built using available network hardware. All 134 designed cluster computers have the required performance of 500 TFLOPS or slightly more, but other characteristics differ significantly. Technical characteristics are plotted in Figure 17.2, and economic characteristics are plotted in Figure 17.3. Marked with a cross is the optimal cluster design with the lowest value of the objective function "Cluster TCO/Peak performance" ratio.

The optimal cluster design is based on a dual-processor configuration of a compute node with 16-core CPUs running at 2,3 GHz. Note this is different from the results of section 17.2 where optimality of a configuration was determined by its "Node cost / Node peak performance" ratio (which we called "local optimality"), and where CPUs running at 2,1 GHz were found to be optimal. Using that locally optimal configuration without detailed analysis would be a simple but erroneous choice, because the cluster design based on that configuration ranks 3rd after detailed analysis, and, despite this seemingly simple change



(a) TCO to peak FP performance ratio ($/GFLOPS)

(b) Power (W)

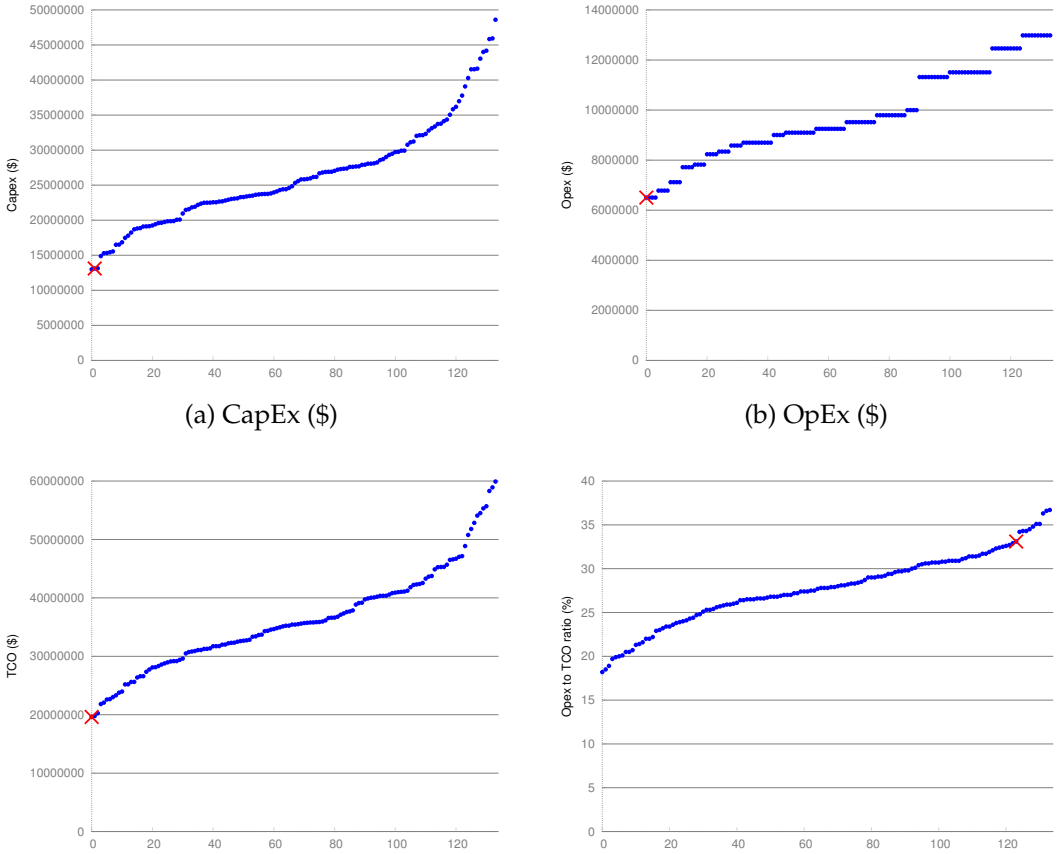
(c) Number of racks

(d) Weight (kg)

(e) Nodes

(f) Cores

Figure 17.2.: Technical characteristics of 134 cluster designs with $R_{peak} = 500$ TFLOPS (order of configurations is not preserved across charts). The cross denotes optimal cluster design with the minimal value of "Cluster TCO/Peak FP performance" ratio.



(a) CapEx ($)

(b) OpEx ($)

(c) TCO ($)

(d) OpEx to TCO ratio (%)

Figure 17.3.: Economic characteristics of 134 cluster designs with $R_{peak} = 500$ TFLOPS (order of configurations is not preserved across charts). The cross denotes optimal cluster design with the minimal value of "Cluster TCO/Peak FP performance" ratio.

of CPU model, that design has a $637,800 higher TCO.

As the charts indicate, the optimal cluster design has the lowest values of several characteristics: "TCO/Performance" ratio (which is obvious), power consumption, number of racks, weight, and number of compute nodes. However, it has a medium number of cores, 54,368: there are designs with fewer cores (such as a cluster with 41,680 cores, based on the powerful 8-core 3 GHz CPUs and ranked 19th) or with more cores (such as a cluster with 78,144 cores, based on low-power 16-core 1,6 GHz CPUs and ranked 13th).

The optimal design also has the lowest operating expense and TCO across all designs. It's capital expense (CapEx, or cost of equipment) is $13,12M and is the 2nd lowest across all designs; the minimal CapEx of $12,99M belongs to another configuration (the chart in Figure 17.3a is not detailed enough to show this). This fact again highlights that low cost of equipment does not necessarily result in the lowest TCO.

Counterintuitively, the optimal design has one of the highest values of "OpEx/TCO" ratio (see Figure 17.3d). The explanation for this phenomenon is the following: as designs

move away from the optimum, their CapEx (and hence TCO) grows rather fast; at the same time, their OpEx grows slower, resulting in the smaller share of OpEx in TCO for those designs.

Another useful finding from this section is the "TCO of 1 TFLOPS" of peak floating-point performance. In our case, the optimal cluster design, with all infrastructural components (network and UPS), has a "TCO/Peak FP performance" ratio of $39,2/GFLOPS (see chart in Figure 17.2a), or $39,200 per TFLOPS. Such a value can be used to make a quick but realistic estimate of the total cost of ownership of a future supercomputer given its peak performance. Alternatively, one can quickly estimate the capability of a supercomputer that one can buy for a given budget; e.g., with the budget of $200,000 the peak performance is $\$200{,}000/39{,}200 = 5{,}1$ TFLOPS.

Of course, as new hardware becomes available on the market, such analysis should be repeated, taking this new hardware – servers, accelerators, etc. – into account, together with its prices. This will allow to derive a new estimate for the "TCO of 1 TFLOPS" of performance. For example, recent accelerators such as GPGPUs or "Intel Xeon Phi" allow to considerably reduce the TCO to performance ratio, compared to the above figure of $39,200 per TFLOPS.

17.4. Designing for ANSYS Fluent Performance Requirements

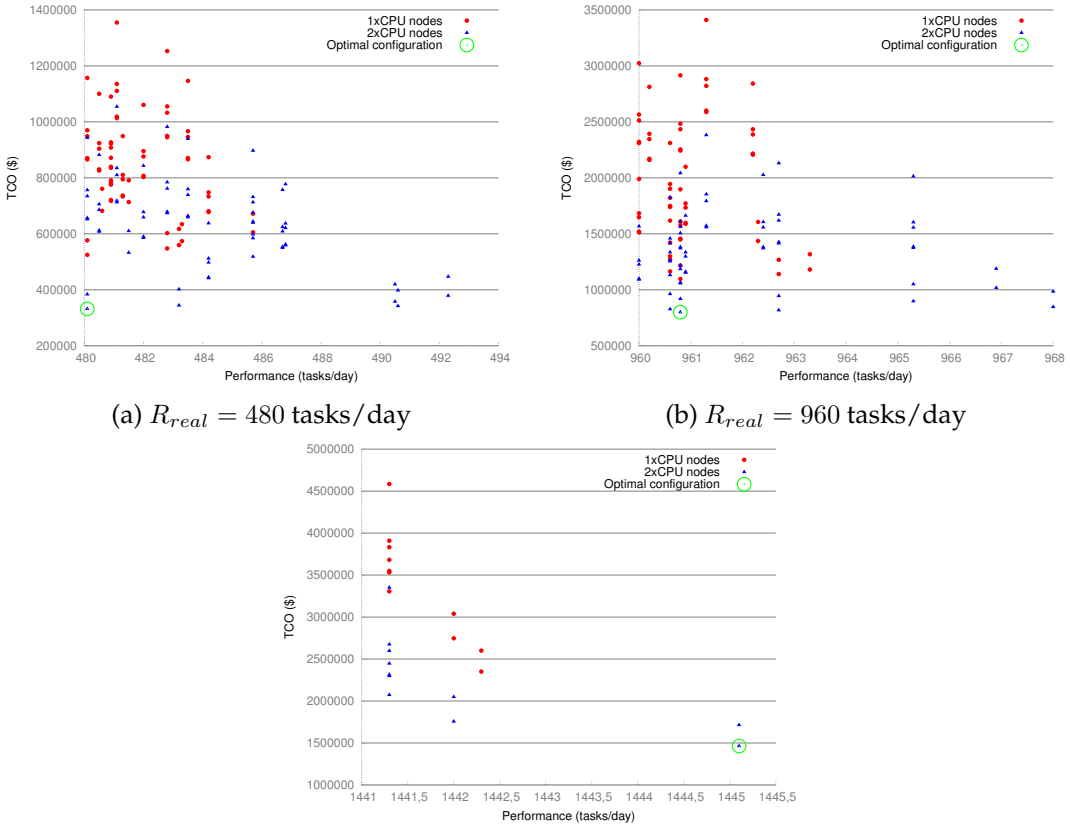
The analysis in the previous section provided demonstration of the capabilities of the algorithm and the CAD system in building optimal cluster designs and accurately estimating their technical and economic characteristics.

However, we can conduct an even more insightful analysis if we try to design a cluster based on the performance requirements of a real application rather than simple peak floating-point performance. In this section we instruct the CAD system to employ the performance model for "ANSYS Fluent" CFD software suite that we developed in Chapter 9 and design clusters capable of delivering a predefined level of real performance, measured in tasks per day, on a benchmark task `"truck_111m"`.

We will design clusters for the following levels of performance: 480, 960 and 1,440 tasks per day. It turns out that these performance levels are only attainable for configurations that use InfiniBand network adaptors; other configurations will be disregarded by the CAD tool.

Altogether we have 132 compute node configurations with InfiniBand adaptors, and all of them are capable of attaining the performance of 480 tasks per day. However, as we increase the desired performance level, the number of suitable configurations decreases: for 960 and 1,440 tasks per day, only 118 and 22 configurations remain, respectively. Each compute node configuration corresponds to one cluster design, and we plot resulting designs for all three cases in Figure 17.4, using geometric interpretation that we introduced in Chapter 6. On every graph, the optimal configuration is circled.

We summarise characteristics of the three designs in Table 17.2. It is important to note that simple, cost-efficient compute node configurations may become unsuitable for clusters with high performance requirements. For example, when moving from 960 to 1,440 tasks per day, the cost-efficient 16-core CPUs running at 2,3 GHz are unable to deliver required performance, and higher clock frequency is required. (In other words, when indi-



(a) $R_{real} = 480$ tasks/day

(b) $R_{real} = 960$ tasks/day

(c) $R_{real} = 1,440$ tasks/day

Figure 17.4.: Cluster designs for real performance of 480, 960 and 1,440 tasks per day for ANSYS Fluent, on the "truck_111m" benchmark. Optimal configurations are circled.

vidual cores are slow, even a large number of them does not help, as performance quickly flattens out). In case of $R_{real} = 1{,}440$ tasks per day, 12-core CPUs running at 2,6 GHz must be used, and trying to further increase performance requirements above 1,440 tasks per day would eventually require the use of the fastest CPUs that we have in the database: the 8-core 3 GHz CPUs.

Another interesting observation from the table is that efficiency of running a parallel workload inevitably decreases when degree of parallelism is increased, hence to maintain linear performance increase the amount of hardware thrown to the task must be increased superlinearly. For example, to double performance from 480 to 960 tasks per day, we need to increase the number of CPU cores from 832 to 1,920 – that is, by the factor of $\times$2,31.

Similarly, the expenses also increase superlinearly, which can be seen by the increasing value of the criterion function, TCO to performance ratio: had the TCO increase been linear with regard to performance, this value would remain constant. This is explained by the amount of high-speed (and therefore not cost-effective) hardware that must be used to attain the highest levels of performance.

In fact, performance of 1,440 tasks per day means that the system will be able to deliver

Characteristic	R_{real}, tasks per day			
	480	960	1,440	1,440 throughput mode
CPU model	AMD 6276	AMD 6276	AMD 6238	AMD 6276
CPU cores per CPU	16	16	12	16
CPU clock frequency, GHz	2,3	2,3	2,6	2,3
CPU cores per compute node	32	32	24	32
Compute nodes	26	60	126	90
CPU cores, total	832	1,920	3,024	2,880
Compute node cost, $	6,596	6,596	5,836	5,301
Network technology	InfiniBand	InfiniBand	InfiniBand	10GigE
Network topology	Star	Star	Star	Star
Network switch	36 ports	72 ports	126 ports	90 ports
Power, W	9,656	22,326	45,210	33,186
Racks	2	3	4	3
Weight, kg	581	894	1,373	1,111
CapEx, $	219,396	554,560	986,916	655,790
OpEx, $	112,395	244,845	476,612	350,443
TCO, $	331,791	799,405	1,463,528	1,006,233
OpEx / TCO, %	33,9	30,6	32,6	34,8
TCO/Perf., $/(tasks/day)	691,1	832,0	1012,8	681,9
Time to solution, sec.	180	90	60	446

Table 17.2.: Characteristics of three cluster designs for different levels of performance of ANSYS Fluent.

one solved task per minute. The user might not need this high rate of system response. Instead, the acceptable alternative could be to have three solved tasks per three minutes, or, say, six tasks per six minutes – that is, by utilising throughput computing mode (see section 9.2). This is the same one task per minute *on average*, and can be perfectly suitable for many situations.

We re-ran the design procedure for 1,440 tasks per day, this time specifying a directive for the performance model that allows to utilise throughput mode. Characteristics of the resulting design are listed in the last column of Table 17.2. This new design makes use of built-in Ten Gigabit Ethernet network adaptors instead of expensive InfiniBand adaptors; this reduces costs of compute nodes.

This cluster still guarantees the throughput performance of 1,440 tasks per day; however, as InfiniBand network is not used, scalability of a single task is limited to only 384 cores, which is the upper reasonable number of cores for Ten Gigabit Ethernet runs with this benchmark (see Chapter 9). Therefore the cluster will be logically divided into 7 blocks of 384 cores each (that is, 12 compute nodes), and one smaller block of 192 cores (6 compute nodes). Each of the bigger blocks will deliver 193,8 tasks per day, which is one task each 446 seconds, or 7,4 minutes; the smaller block will deliver 118,8 tasks per day, or one task

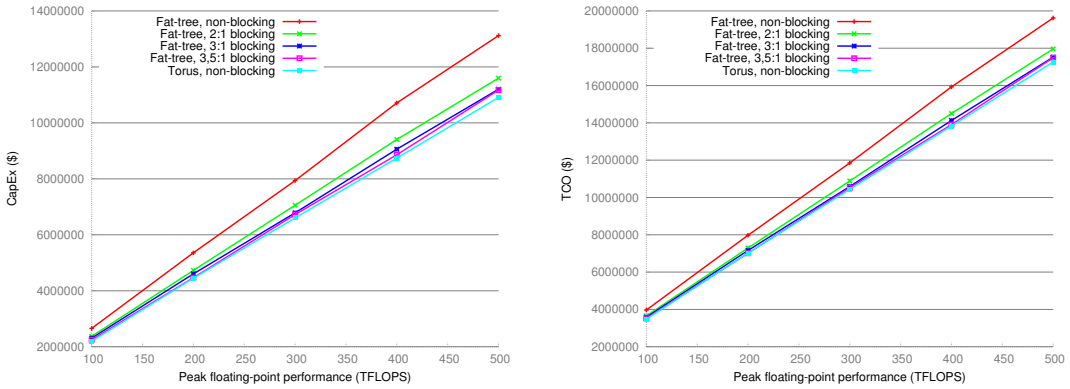
each 12,1 minutes (values obtained via performance model from Chapter 9). The total aggregate performance of all blocks will equal $7 \cdot 193{,}8 + 118{,}8 = 1475{,}4$ tasks per day.

As can be seen, this design has a $\times 7{,}4$ higher time to solution (446 seconds) than the corresponding design for 1,440 tasks per day that is not utilising throughput mode (60 seconds). However, it uses more cost-effective equipment, and requires less of that equipment; as a result, it has a 31% lower TCO, making it an attractive option for many situations where absolute performance of individual compute tasks is not important.

17.5. Impact of Network Topology and Blocking Factor

We have seen in section 14.4 that using a blocking network to interconnect cluster nodes can reduce cost of network equipment (see, for example, Figures 14.1 and 14.2). Additionally, torus networks cost considerably less than fat-tree networks for the same number of nodes.

It is therefore interesting to investigate the impact of network topology and blocking factor with relation to the capital expense and total cost of ownership of the whole computer, and not just the network equipment. For this purpose, we calculated CapEx and TCO of cluster computers with peak floating-point performance of 100 to 500 TFLOPS (the latter corresponds to roughly 1,700 compute nodes). We used non-blocking fat-tree and torus networks as well as three types of blocking fat-trees, with blocking factors 2:1, 3:1 and 3,5:1. These blocking factors correspond to the following distribution of ports on 36-port edge switches (to nodes / to core level): 24:12, 27:9, and 28:8.



(a) CapEx (b) TCO

Figure 17.5.: CapEx and TCO of clusters with the peak floating-point performance of 100 to 500 TFLOPS, with different network designs.

CapEx and TCO of optimal designs are plotted in Figure 17.5. As can be seen from the charts, both economic metrics grow linearly with respect to performance. Moving from the non-blocking fat-tree network to 2:1 blocking network provides certain savings, e.g, TCO is reduced by roughly 8%..9%. However, further increase in blocking factor has negligible effects on economic characteristics. As even the 2:1 blocking can have a degrading effect on parallel application performance (see section 13.2 for a literature review), we recommend to weigh financial savings from blocking topologies against possible performance impact.

Cost savings can still be realised when large "islands" of compute nodes are connected with non-blocking fabrics, as done in "SuperMUC" [55], and when compute jobs are not supposed to cross island boundaries – in these cases, communication within the islands is always non-blocking. However, such configurations require the use of topology-aware job schedulers.

17.6. Impact of UPS Backup Time

The UPS system can be configured in three different ways, with each configuration guaranteeing its own backup time (see Table 15.3 for the list of configurations). UPS systems with backup time of 49 minutes are much more expensive than systems with backup time of 10 minutes that we used for analysis in the previous sections of this chapter. However, cost of the UPS system is just one of factors contributing to total costs. Let us investigate how requirements for UPS backup time affect CapEx and TCO of cluster designs.

For this purpose, we design clusters with peak floating-point performance of 100 to 500 TFLOPS, as we did in the previous section, but this time fixing the network type – a non-blocking fat-tree network – and varying UPS backup time, setting it to 10, 20, and 40 minutes.

Due to the greedy nature of the UPS design algorithm, the UPS system is always designed using blocks that have the lowest cost per kW of power but still satisfy constraints on backup time (see more details about the algorithm in section 15.9). Therefore, when backup time of 10 minutes is specified, all three types of blocks listed in Table 15.3 are suitable, and the 45 kW blocks with the lowest cost per kW are generally used.

When the backup time is 20 minutes, the 45 kW blocks are no longer usable, as they can provide only 12 minutes of backup time; hence, 15 kW and 30 kW blocks must be used instead. Finally, when the required backup time is 40 minutes, the only remaining type of suitable blocks are 15 kW blocks; they have the highest cost per kW which results in expensive UPS systems.

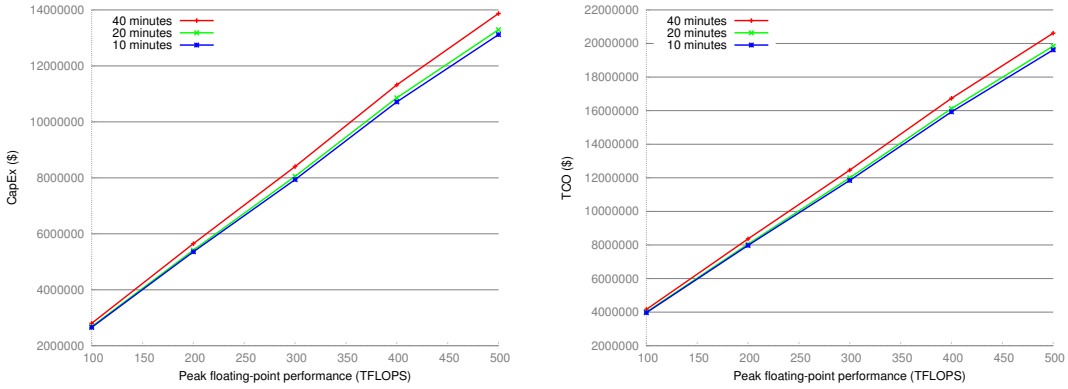
We plot charts with CapEx and TCO of clusters with different UPS backup times in Figure 17.6. As seen from the charts, designing a UPS system with 40 minutes of backup time instead of 10 minutes adds only 4% to 5% to TCO over the cluster's lifetime.

17.7. Components of Total Cost of Ownership

Using results of the previous sections, we can graphically represent shares of TCO components. Optimal cluster designs from sections 17.3 and 17.4 are represented in Figure 17.7. Pie charts for both clusters are surprisingly similar, even though the machines have very different compute node counts: 1,699 and 126 nodes, respectively.

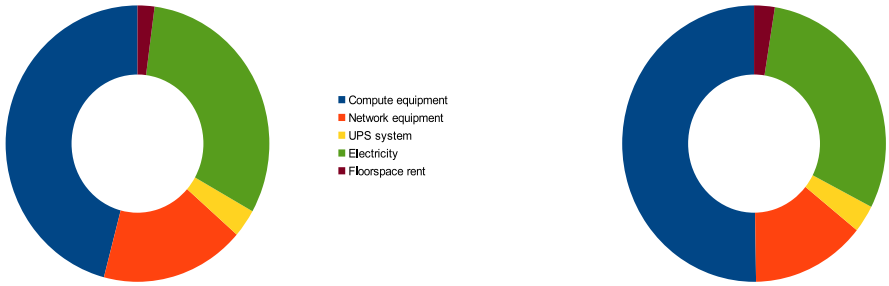
In both cases, electricity costs take up a large share of TCO, which is explained by high electricity prices set by the co-location facility: \$0,37 per kW·h [37], whereas wholesale electricity price for industrial consumers in Germany in May 2012 were on the order of \$0,15 per kW·h [29].

Network equipment comprises a substantial share of TCO, so that using a dual-rail network would essentially double the amount of network equipment, causing increase in TCO. However, this not only leads to increased network throughput, but also to increased



(a) CapEx (b) TCO

Figure 17.6.: CapEx and TCO of clusters with the peak floating-point performance of 100 to 500 TFLOPS, with different UPS backup times.



(a) Cluster designed for peak floating-point performance $R_{peak} = 500$ TFLOPS

(b) Cluster designed for ANSYS Fluent performance 1,440 tasks per day

Figure 17.7.: TCO components for two cluster designs.

resilience to network failures, which can become an important factor in future exascale computers.

In both cases floorspace rental costs represent a small share of TCO; even if doubled, they would not add much. This again reminds that space savings resulting from dense computing solutions such as blade servers do not always justify higher procurement cost of this equipment.

18. Summary and Future Directions

> Your task is not to foresee the future, but to enable it.
>
> *Antoine de Saint Exupéry*

18.1. Summary

In this thesis we discuss the problem of automated design of cluster supercomputers. This type of computing machinery has been known for almost twenty years, but so far most of design decisions are based on engineers' intuition, which is not an adequate replacement to thorough analysis. This thesis changes the situation by providing a framework for automation that allows to (a) quickly explore large design spaces, (b) present reliable quantitative evaluations of technical and economic characteristics of cluster designs, and (c) stimulate engineers to develop their intuition in fields where automation is not yet possible.

The thesis consists of three parts. Part I starts with the explanation of benefits of automation in computer design and reviews related work. It also provides the reader with a gentle introduction into the statement of a problem and our proposed approaches to its solution.

Part II defines a criterion function, which is a major component of any optimisation problem, and describes two methods to deal with combinatorial explosion, namely heuristics and design constraints. It additionally includes a detailed discussion on the role of economics in designing, building and operating supercomputers. The part concludes with the chapter on performance modelling, which introduces the notion of inverse performance models and provides an algorithm to quickly calculate the number of compute blocks required to reach a specified level of performance.

Finally, Part III integrates all relevant automated design processes, from representing configurations of technical systems using graphs, to main algorithm of automated design, to auxiliary algorithms that design cluster subsystems such as interconnection networks and a UPS system. The part concludes with an extensive evaluation of the framework on a number of real life cases, accompanied by estimations of technical and economic characteristics of corresponding cluster designs.

18.1.1. Key Findings

Key findings of Chapter 1:

1. Automation of design procedures leads to many benefits, including (a) exploration of a larger design space, (b) accurate estimations of technical and economic characteristics of the future supercomputer, and (c) automated generation of documentation required to assemble the computer.
2. Automated design of cluster supercomputers is a logical continuation of electronic design automation (EDA) on a higher, system level.
3. Approaches to designing cluster supercomputers can also be used for a more general task of designing large "warehouse-scale" computing facilities.

Key findings of Chapter 2:

The problem of automated system-level design of parallel computers was tackled by different researchers since at least late 1970s. Since each study analysed different aspects of the problem, the results were rather disconnected from each other. Our thesis aims to "connect the dots" by creating a solid integrated automation framework.

Key findings of Chapter 3:

1. The problem of automated design of cluster supercomputers can be formulated as a single-criterion discrete optimisation problem with constraints.
2. Configurations of real life technical systems can sometimes be characterised by unexpected interdependencies between components; these interdependencies can be represented using graph structures further introduced in Chapter 10.

Key findings of Chapter 6:

1. Multi-objective optimisation can be used for our problem, with detecting Pareto front and choosing one of non-dominated solutions.
2. However, we can calculate the value of "TCO to performance ratio" for each cluster design alternative, and use this function as our criterion function, thereby reducing the problem to a simpler single-objective optimisation problem. Besides, the optimal solution is also necessarily a non-dominated solution of multi-objective optimisation.
3. To take into account uncertainties in performance and cost estimations of design alternatives, interval arithmetic can be used.

Key findings of Chapter 7:

1. Changes in computer systems (adding or replacing components) lead to changes in both performance and cost. Performance increase resulting from the change should exceed cost increase, otherwise the change is impractical. In real life situations many counteracting factors come into play, therefore there are no "rules of thumb" that allow to select good components (e.g., whether to use uni- or dual-processor servers, a fat-tree or a torus network, etc.) As a result, automation helps explore large design spaces.

2. Total cost of ownership (TCO) allows to take into account various expenses in a very natural way. For example, instead of introducing power consumption of the computer into the criterion function, we can use cost of consumed electricity as part of the TCO, which is more objective.

3. TCO as a function of the number of compute nodes is not additive.

Key findings of Chapter 8:

1. Structure of compute nodes is represented via multipartite graphs, where each partition corresponds to a specific feature of a node. Partitions contain vertices that correspond to possible implementations of that feature. Each configuration is a unique path in the graph that traverses it from start to end.

2. Increasing the level of detail in description leads to adding more partitions to the graph, and each new partition with n vertices increases the number of paths, and hence configurations, by n times. Adding p partitions increases the number of paths n^p times – that is, exponentially. As a result, we should keep the number of features encoded in a graph to the lowest possible level to minimise the effect of combinatorial explosion.

3. Many compute node configurations will lead to cluster designs that are far from optimal. We can disregard these configurations on as early stage as possible using two mechanisms: heuristics and constraints.

4. The heuristic calculates the value of "Cost / Peak floating-point performance ratio" for each configuration and then disregards 80% of configurations based on this metric. The heuristic works automatically and allows to quickly shrink the design space, but can be unsuitable when designing clusters for workloads that are not floating-point based, such as data mining.

5. Constraints can be imposed by the user on metrics of (a) compute nodes, and (b) the whole cluster design. For example, the user can request configurations with a specific amount of main memory per core, a specific number of cores per CPU, an upper bound on power consumption of the cluster, etc. Constraints require user intuition, and hence are not automatic; however, they are still good at shrinking the design space.

Key findings of Chapter 9:

1. Performance models are mathematical objects of various nature that receive the configuration of the cluster computer, including the number of compute blocks (nodes, CPUs, cores or otherwise), and output computer's performance at a particular task.

2. We introduce the notion of *inverse performance models*, which, given the required performance, return the number of compute blocks needed to attain it.

3. We present a simple two-phase iterative algorithm that implements inverse performance modelling using a sequence of queries to a direct performance model, which in turn can have any internal structure.

4. We also develop a simple direct performance model for "ANSYS Fluent" CAE software suite based on benchmarking results. The model approximates parallel efficiency with a piecewise linear function.

5. To make the CAD system modular, performance models are shipped as modules that are queried by the CAD system via network. This allows to quickly re-orient the CAD system from one performance model to another, including updated versions of models.

Key findings of Chapter 10:

1. Configurations of arbitrary technical systems, including compatibility relations between components, can be represented with multipartite graphs. Undirected graphs have more expressive power, while directed acyclic graphs provide convenient visual clues to the user about compatibility of components. We use directed acyclic graphs in this thesis.

2. Characteristics of technical systems can be easily evaluated when traversing the configuration graph, using expressions assigned to graph's vertices. We provide the grammar for such expressions.

3. We propose graph transformations that simplify visual representation of the graphs, and present the syntax for encoding graphs in XML language.

Key findings of Chapter 11:

1. Combinatorial optimisation procedure relies on generation of valid candidate solutions that are subsequently analysed. Simulated annealing and evolutionary algorithms, such as genetic algorithms, were found to be unsuitable for our task, because they tend to generate invalid configurations; this is due to complex compatibility relations between components of candidate solutions.

2. The main algorithm of the thesis is described.

3. The automation framework can perform thorough search of the design space, so the final solution is formally guaranteed to be optimal – but only within the limits of the framework. However, there are factors that we are unable to account for – such as imprecision of performance models, uncertainties in costs, etc. – that make our solutions good but not necessarily optimal: theoretically, based on the same input data, another supercomputer could be designed that would have a higher performance, or lower cost, or both. This is as expected, as the quality of solutions is always limited by the quality of input data.

4. Our formulation of the design problem is to minimise the total cost of ownership. There is also a dual formulation, with the goal to maximise performance, but we demonstrate that it would lead to longer design times.

Key findings of Chapter 12:

1. The prototype CAD system is modular, with each module implementing its own design stage. One type of modules implement performance models, other modules are used to design networks and UPS systems. This structure allows to use modules from different equipment manufacturers, or to easily switch to new versions of modules with advanced functionality or enhanced precision.

2. As configurations of compute nodes are handled independently by the CAD tool, the entire design process can be parallelised in a trivial way.

Key findings of Chapter 13:

1. We present an algorithm for designing two-level fat-tree networks, choosing the optimal combination of available network switches (including modular switches), with arbitrary blocking factors.

2. Per-port metrics of network switches (such as per-port cost) can be used to quickly estimate corresponding values of many technical and economic characteristics of two-level fat-tree networks built using that switches, by simply multiplying per-port metrics by $3N$, where N is the number of compute nodes to be interconnected. The estimated value is a lower bound on actual value, and discrepancy between the two is generally low.

3. For two-layer fat-tree networks, future expansion is greatly simplified if the core layer is designed from the start to accommodate this expansion. Moreover, procurement of extra edge switches can be delayed until the expansion.

Key findings of Chapter 14:

1. We present an algorithm to design multi-dimensional torus networks. The number of dimensions is automatically chosen by a heuristic.

2. Cost comparison reveals that torus networks are significantly cheaper than non-blocking fat-trees, while fat-trees with a blocking factor of 2:1 occupy a medium position between them.

Key findings of Chapter 15:

1. In air-cooled installations, using cold outdoor air can significantly reduce energy consumption for cooling purposes, compared to traditional cooling systems based on direct expansion, which reduces operating expenses. If air contamination is unlikely, direct expansion cooling systems are not required at all, which leads to further reduction in TCO.

2. Decision chart for selecting cooling methods for air-cooled computing equipment is presented. For hot climates, the chart recommends designing a small cooling system, to cool outside air to allowed server operating temperature. The method for calculating the partial cooling capacity P_{part} of such a system is proposed.

3. Water cooling allows to easily reuse heat extracted from compute equipment, for example, for heating greenhouses. A greenhouse utilising heat from the SuperMUC computer can supply enough locally grown tomatoes to cover the needs of 57,000 people.

4. We present a greedy algorithm to design UPS systems, based on choosing UPS blocks with the lowest cost per kW.

Key findings of Chapter 16:

1. There are two strategies for partitioning hardware into blocks: consolidation and distribution. Consolidation can lower costs due to effects of economy of scale, while distribution increases independence of blocks, improving survivability.

2. We provide a set of heuristics for placing cluster equipment into racks, delivering either regular wiring patterns or dense packaging. The heuristics are applicable in the case of fat-tree and indirect torus network topologies.

3. We propose an algorithm to calculate the size of floor space required to accommodate a given number of equipment racks, taking into account dimensions of racks and clearances between them.

Key findings of Chapter 17:

1. Compute nodes can have several hundreds of configurations, depending on the level of detail in their description. These configurations have vastly different technical and economic characteristics. Configurations that are unsuitable or unpromising can be quickly filtered out, thereby decreasing the time required for detailed analysis.

2. Analysis indicated that for a sample 500 TFLOPS cluster computer based on our reference hardware dataset, the cost of 1 TFLOPS of peak floating-point performance is $39,200.

3. When having the lowest possible time to solution is not important, the required level of performance can be attained by using throughput mode. Clusters designed for throughput mode have a significantly longer time to solution, but at the same time they are less expensive.

4. Using fat-tree networks with a 2:1 blocking factor allows to reduce TCO by roughly 8%..9%. Further increasing the blocking factor has negligible effect on TCO but may cause performance degradation.

5. Raising the UPS backup time from 10 to 40 minutes increases TCO by 4%..5%.

6. Electrical power costs represent a significant share of TCO, especially in settings where electricity is expensive, such as when renting data centre space from co-location providers.

7. Floor renting costs represent a small share of TCO; this means that the use of expensive blade servers is unlikely to be justified solely by the reduced equipment footprint.

18.2. Future Directions

18.2.1. Designing for Reliability

Reliability will be of utmost concern for future exascale computers [24]. In its current state, the proposed design automation framework does not address reliability concerns. There are, however, three ways in which it can be extended.

First, we can prescribe reliability characteristics (such as mean time to failure) to individual components of compute nodes, encoding them in the configuration graph. This way, configurations with less reliable components can be disregarded by using constraints (see section 8.3), right after configurations are generated from the graph. Second, if reliability of a compute node cannot be easily calculated from reliability of individual components, then the configuration can be sent for inspection to the reliability model, implemented as a separate module of the CAD system, similar to performance modelling.

Finally, reliability of infrastructural components – interconnection network, UPS and storage systems, etc. – should be calculated within CAD modules that design these components, and returned to the CAD system for analysis.

Reliability of the designed cluster computer can further be used to estimate the failure rate of its components, and thereby the required size of the pool of substitution equipment that must be maintained for quick repair, as well as the repair staff head count.

18.2.2. Reflections on Trust for Web Services

It is supposed that equipment vendors can make available custom design modules, intended for designing infrastructural components and based on equipment of that particular vendor. For example, the UPS design module that we review in section 15.9 is based on the "Liebert APM" UPS model manufactured by "Emerson Network Power".

The internal structure of such design modules does not necessarily have to be open; the CAD system views it as a black box. However, the genuineness of the results returned by such third party modules cannot be checked, because the structure of the modules is not open. This presents a danger of the "trojan" supply of information, and suggests an idea of maintaining a white list of trusted organisations.

18.2.3. Towards a Decision Support System

The main goal of design automation is to help the engineer in making decisions. The proposed framework inspects the whole design space and eliminates unpromising or unsuitable candidate solutions, explaining the reason for dismissal to the engineer.

This allows the engineer to understand why the CAD tool made that specific decision, but does not enhance the intuition. To mitigate this, the CAD tool can supply the engineer with the wealth of *hints*. Let us consider two examples.

First, suppose the user has compute nodes each capable of performance of 10 tasks per day, and wants to build a cluster computer capable of performance of 200 tasks per day. Even with linear performance scaling (which is unlikely in practice), they will need at least 20 such compute nodes. Now, if each node consumes 300 W of power and weights 15 kg, then the *lower bound* on power consumption and weight of the cluster is 6 kW and 300 kg,

respectively. The GUI of the CAD tool can then display these values, to give a hint to the user that the resulting cluster computer will have *at least* these values of corresponding characteristics.

The CAD tool can also prevent the user from setting constraints on technical characteristics that contradict the lower bounds found above, as an additional indication that no designs would match those unrealistic constraints. For example, if the above procedure found that for the three configurations of compute nodes present in the database the lower bounds on cluster power consumption are 6, 7 and 9 kW, respectively, then it makes no sense for the user to specify the power constraint of 5 kW, because all designs are guaranteed to violate that constraint. Preventing the user from setting unrealistic constraints also allows to refrain from launching useless design procedures.

The second example concerns costs. Suppose the user specifies a cost constraint in the CAD tool, and designs that exceed the budget are automatically disregarded with a corresponding message that the cost constraint was violated. In practice, the cost constraint is exceeded when the minimal performance level specified by the user required to use too many compute nodes to stay within the budget. It would be helpful if the CAD tool not only informed the user that the constraint was violated, but also indicated how big the violation was.

All these hints enhance the user's intuition, allowing them to develop more realistic expectations as to what kind of designs they can arrive at.

18.2.4. Role of Automation in System-Level Design

The challenge with supercomputer design is that there is no end-to-end design methodology, and different levels of design hierarchy are handled by different professional communities. As a result, technical and economic characteristics of low-level stages, such as microprocessor design, are loosely related to corresponding characteristics on the system level.

Consider a team that is designing a new microprocessor, trying to cater to future uncertain market preferences which are prone to irrational changes. The team uses design methodologies specific to their fields, namely radio electronics and semiconductor fabrication. They make design trade-offs: for example, they can use die space for lots of simple cores, or put less cores and use the remaining space for cache memory.

Then, on the board level, the chips are placed on a board and connected to each other and to memory. Again, there are opportunities for making design trade-offs: for example, how many chips to place on a board, and how to connect them to each other. Board designers are more concerned by radio electronics constraints, such as interference, rather than by limitations of semiconductor technology. They use specific EDA tools and corresponding methods. The modules – microprocessors, memory and network chips – are considered to be finished devices, without delving into their internal structure.

On the system level, designers have to possess a yet another set of skills, this time largely in the fields of electrical and mechanical engineering. As a result, predicting how low-level design choices made on the chip level can impact performance of a future large-scale system incorporating tens of thousands of such chips is not easy. Design automation allows to expose effects of low-level design decisions on system-level metrics.

Appendices

A. ANSYS Fluent Benchmark Data

Performance data used to construct performance model for ANSYS Fluent computational fluid dynamics (CFD) software was obtained using version 13.0 of the software, on a benchmark simulating air flow around a truck body (`truck_111m`). Data presented in the tables below was taken from ANSYS website [6].

There is one dataset for Ten Gigabit Ethernet network type, three for InfiniBand, and one for proprietary SGI NumaLink. The column with scaled performance rating, if present, is used to plot all experimental data on a single graph (Fig. 9.4).

Cores	Performance rating
24	22
384	247,2

Table A.1.: Benchmark data for "Cisco UCS C200 Intel Xeon 5670", Ten Gigabit Ethernet network, clock frequency $f = 2,93$ GHz

Cores	Performance rating
96	84,2
192	179,8
384	349,9
768	679,2
1536	1239,6
3072	1928,6

Table A.2.: Benchmark data for "SGI Altix Ice 8400EX Intel Xeon 5690", InfiniBand network, clock frequency $f = 3,47$ GHz

Cores	Performance rating	Performance rating, scaled to $f = 3,47$ GHz
96	51	70,8
192	99,1	137,6
384	195,3	271,1
768	369,4	512,7
1536	658,5	914,0

Table A.3.: Benchmark data for "SGI Altix Ice 8400 AMD Opteron 6180", InfiniBand network, nominal clock frequency $f = 2,5$ GHz

Cores	Performance rating	Performance rating, scaled to $f = 3,47$ GHz
96	84,6	100,2
192	161	190,7
384	324,3	384,1
768	626,1	741,5

Table A.4.: Benchmark data for "IBM DX360 M3 Intel Xeon 5670", InfiniBand network, nominal clock frequency $f = 2,93$ GHz

Cores	Performance rating	Performance rating, scaled to $f = 3,47$ GHz
64	41,5	53,9
128	74	96,2
256	138,9	180,5
512	274,7	357,0

Table A.5.: Benchmark data for "SGI UV 1000 Intel Xeon E7 8837", proprietary NumaLink network, nominal clock frequency $f = 2,67$ GHz

B. Hardware of Compute Nodes

In this thesis we used Hewlett-Packard's "BL465c G7" servers as cluster compute nodes. In the tables below we list characteristic of the node itself and its components: CPUs, main memory, and network adaptors. All information about compatibility of components (e.g., what CPUs can be fitted into the server) as well as prices were taken from "HP BladeSystem Power Sizer Tool" by Hewlett-Packard [38], and power consumption of components was additionally estimated using "HP Power Advisor Tool" [40].

Note that certain CPU models, although outdated, were included for completeness. For example, the older 12-core "AMD Opteron 6176" CPU can be replaced with a newer 16-core "AMD Opteron 6276" model: at the same clock frequency it has more cores and more L3 cache memory, while having a lower price.

The criterion function that we use, "Cost/Performance", makes such outdated and overpriced components less favourable, therefore the CAD system will never mark them as optimal. However, not having these components in the production database in the first place means that fewer configurations have to be analysed, leading to faster design times.

Codename	Feature size, nm	Cores	Model	Frequency, GHz	L3 Cache, MBytes	TDP, W	Cost, $
Magny-Cours	45	8	6128 HE	2,0	12	85	649
			6132 HE	2,2	12	85	749
			6134	2,3	12	115	609
			6136	2,4	12	115	859
			6140	2,6	12	115	1149
		12	6164 HE	1,7	12	85	869
			6166 HE	1,8	12	85	1015
			6172	2,1	12	115	1189
			6174	2,2	12	115	1339
			6176	2,3	12	115	1449
Interlagos	32	8	6212	2,6	16	115	349
			6220	3,0	16	115	649
		12	6234	2,4	16	115	479
			6238	2,6	16	115	569
		16	6262 HE	1,6	16	85	649
			6272	2,1	16	115	649
			6274	2,2	16	115	779
			6276	2,3	16	115	949

Table B.1.: Characteristics of AMD Opteron CPUs

To calculate power consumption of each configuration, we add up power consumption of all components. For a CPU, the power consumption is considered to be equal to its TDP. Power consumption of the motherboard is listed in Table B.4. There was no reliable data on power consumption of memory modules and network adaptors, therefore we considered all memory configurations to consume 18 W, irrespective of the type and number of memory modules used; power consumption of network adaptors was set to zero.

Applicability	Memory type	Low voltage	Layout	Total size, GBytes	Cost, $
AMD Opteron 6100 series CPUs	PC3-10600R	–	4x8 GBytes	32	796
	PC3-10600R	–	8x4 GBytes	32	840
	PC3L-10600R	Yes	2x16 GBytes	32	1798
	PC3L-10600R	Yes	8x8 GBytes	64	1592
	PC3L-10600R	Yes	4x16 GBytes	64	3596
AMD Opteron 6200 series CPUs	PC3-12800R	–	4x8 GBytes	32	996
	PC3-12800R	–	8x8 GBytes	64	1992

Table B.2.: Characteristics of main memory configurations

Technology	Link speed, Gbps	Ports	Cost, $
10 Gbit Ethernet	10	2	0 (Built-in)
4X QDR InfiniBand	40	2	1295

Table B.3.: Characteristics of network adaptors

Characteristic	Value
Model	BL465c G7
Power, W	63
Weight, kg	6
DIMM Slots	16
Cost, $	1411

Table B.4.: Characteristics of Hewlett-Packard's "BL465c G7" server.

Bibliography

[1] Y. Ajima, T. Inoue, S. Hiramoto, and T. Shimizu. Tofu: Interconnect for the K computer. *Fujitsu Sci. Tech. J*, 48(3):280–285, 2012. http://www.fujitsu.com/downloads/MAG/vol48-3/paper05.pdf.

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[3] Khalil Amiri and John Wilkes. Automatic design of storage systems to meet availability requirements. https://www.hpl.hp.com/research/ssp/papers/HPL-SSP-96-17.pdf, 1996.

[4] Eric Anderson, Susan Spence, Ram Swaminathan, Mahesh Kallahalla, and Qian Wang. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems (TOCS)*, 23(4):337–374, 2005.

[5] Rob Andrews and Joshua M. Pearce. Environmental and economic assessment of a greenhouse waste heat exchange. *Journal of Cleaner Production*, 19(13):1446–1454, 2011.

[6] ANSYS. Release 13.0 test cases benchmarks – external flow over a truck body, model with 111 million elements. http://www.ansys.com/Ansys/en_us/Support/Platform%20Support/Benchmarks%20Overview/Archives/ANSYS%20Fluent%20Archives/Release%2013.0%20Test%20Cases/External%20Flow%20Over%20a%20Truck%20Body%20111m%20%28truck_111m%29, December 2011.

[7] APC by Schneider Electric. APC by Schneider Electric provides infrastructure for the Lomonosov supercomputer (in Russian). http://www.apc.ru/cgi-bin/news_full.cgi?id=328, 2010.

[8] APC by Schneider Electric. ACRD 500 InRow direct expansion cooling unit. http://www.apc.com/products/resource/include/techspec_index.cfm?base_sku=ACRD500&type=1, 2012.

[9] Don Atwood and John G. Miner. Reducing data center cost with an air economizer. http://www.intel.com/it/pdf/Reducing_Data_Center_Cost_with_an_Air_Economizer.pdf, 2008.

[10] Pavan Balaji, Rinku Gupta, Abhinav Vishnu, and Pete Beckman. Mapping communication layouts to network hardware characteristics on massive-scale blue gene systems. *Computer Science - Research and Development*, 26(3):247–256, 2011.

[11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[12] Christian Bischof, Dieter an Mey, and Christian Iwainsky. Brainware for green HPC. *Computer Science-Research and Development*, 27(4):227–233, 2012.

[13] Peyman Blumstengel and Thomas Arenz. Leverage of memory technology for energy-efficient HPC. In *Proceedings of International Supercomputing Conference*, ISC'11, June 2011.

[14] A.N. Bozhko and A.Ch. Tolparov. Structural synthesis on elements of limited compatibility *(in Russian). Science and Education*, 5, 2004.

[15] Matthias Brehm, Axel Auweter, Herbert Huber, and Torsten Wilde. Energy efficient HPC systems: Concepts, procurement & installation. In *Proceedings of International Supercomputing Conference*, ISC'12, June 2012.

[16] Bull. SARA selects Bull for delivery of new petascale national supercomputer. http://www.wcm.bull.com/internet/pr/new_rend.jsp?DocId=763893&lang=en, November 2012.

[17] Bull. Bull extreme computing catalogue. http://www.bull.com/catalogue/overall.asp?cat=bullx&line=bxs-node, January 2013.

[18] Wolfgang Burke. Evolution of HPC at BMW. In *Proceedings of International Supercomputing Conference*, ISC'11, June 2011.

[19] J.M. Cámara, M. Moretó, E. Vallejo, R. Beivide, J. Miguel-Alonso, C. Martínez, and J. Navaridas. Mixed-radix twisted torus interconnection networks. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10. IEEE, 2007.

[20] Christine S. Chan, Yanqin Jin, Yen-Kuan Wu, Kenny Gross, Kalyan Vaidyanathan, and Tajana S. Rosing. Fan-speed-aware scheduling of data intensive jobs. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pages 409–414. ACM, 2012.

[21] D. Chen, N.A. Eisley, P. Heidelberger, R.M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D.L. Satterfield, B. Steinmacher-Burow, and J.J. Parker. The IBM Blue Gene/Q interconnection network and message unit. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–10. IEEE, 2011. http://mmc.geofisica.unam.mx/edp/SC11/src/pdf/papers/tp19.pdf.

[22] William R. Dieter and Henry G. Dietz. Automatic exploration and characterization of the cluster design space. Technical Report TR-ECE-2005-04-25-01, University of Kentucky, Electrical and Computer Engineering Dept., April 2005. http://www.engr.uky.edu/~dieter/pub/TR-ECE-2005-04-25-01.pdf.

[23] Henry G. Dietz. NetWires interconnection network graphing tool. http://aggregate.org/NETWIRES/.

[24] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.C. Andre, D. Barkai, J.Y. Berthou, T. Boku, B. Braunschweig, et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3, 2011.

[25] Nicolas Dubé. Next generation datacenters: The road to net-zero: Energy, carbon, water. In *Proceedings of International Supercomputing Conference*, ISC'12, June 2012.

[26] Marc Duranton, Sami Yehia, Bjorn De Sutter, Koen De Bosschere, Albert Cohen, Babak Falsafi, Georgi Gaydadjiev, Manolis Katevenis, Jonas Maebe, Harm Munk, Nacho Navarro, Alex Ramirez, Olivier Temam, and Mateo Valero. The HiPEAC vision. http://www.hipeac.net/roadmap, 2010.

[27] Douglas Eadline. Disposable HPC. http://www.linux-mag.com/id/7792/, May 2010.

[28] Douglas Eadline. Low cost/power HPC. http://www.linux-mag.com/id/7799/, June 2010.

[29] Europe's Energy Portal. End-user energy prices for industrial consumers. http://www.energy.eu, 2012.

[30] Eurotech. Aurora Tigon. http://www.eurotech.com/en/hpc/hpc+solutions/aurora+hpc+systems/Aurora+Tigon, 2012.

[31] N. Farrington, E. Rubow, and A. Vahdat. Data center switch architecture in the age of merchant silicon. In *17th IEEE Symposium on High Performance Interconnects*, pages 93–102. IEEE, 2009.

[32] Food and Agriculture Organization of the United Nations. Statistics at FAO. http://www.fao.org/statistics/, November 2013.

[33] National Science Foundation. High performance system acquisition: Enhancing the petascale computing environment for science and engineering. http://www.nsf.gov/publications/pub_summ.jsp?ods_key=nsf11511, December 2010.

[34] Marco Di Girolamo, Giovanni Giuliani, Omer H. Abdelrahman, Domenico Sannelli, Juan Carlos Lopez egea, André Giesler, Olli Mammela, Mikko Majanen, and Marcus Kessel. FIT4Green: Final evaluation report on pilots of full-featured enhanced control plug-in and control desk for federated data centre. Technical Report FP7-ICT-2009-4 Project N°249020 FIT4Green, HPIS, July 2012. http://www.fit4green.eu/sites/default/files/attachments/documents/FIT4Green_D6.4_FINAL.pdf.

[35] Google. Google data centers. Efficiency: How we do it. https://www.google.com/about/datacenters/efficiency/internal/.

[36] Amit K. Gupta and William J. Dally. Topology optimization of interconnection networks. *IEEE Computer Architecture Letters*, 5:10–13, January 2006.

[37] Hetzner Online AG. Pricing for Colocation Rack Basic. https://www.hetzner.de/en/hosting/produkte_colocation/rack, July 2013.

[38] Hewlett-Packard. HP BladeSystem Power Sizer Tool. http://h71028.www7.hp.com/ActiveAnswers/cache/347628-0-0-0-121.html.

[39] Hewlett-Packard. HP POD 240a. http://www.hp.com/go/ecopod.

[40] Hewlett-Packard. HP Power Advisor Tool. http://www.hp.com/go/HPPowerAdvisor.

[41] Hewlett-Packard. HP Moonshot system. h17007.www1.hp.com/us/en/enterprise/servers/products/moonshot/, 2013.

[42] Torsten Hoefler. Software and hardware techniques for power-efficient HPC networking. *Computing in Science & Engineering*, 12(6):30–37, 2010.

[43] Pao-Ann Hsiung, Chung-Hwang Chen, Trong-Yen Lee, and Sao-Jie Chen. ICOS: An intelligent concurrent object-oriented synthesis methodology for multiprocessor systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 3(2):109–135, 1998.

[44] IBM. IBM Standalone Solutions Configuration Tool. http://www-947.ibm.com/support/entry/portal/docdisplay?brand=5000008&lndocid=MIGR-62168.

[45] IBM. iDataPlex Rack Type 7825. http://publib.boulder.ibm.com/infocenter/idataplx/documentation/topic/com.ibm.idataplex.doc/dg1bdmst.pdf, May 2008.

[46] IBM. IBM unveils new POWER7 systems to manage increasingly data-intensive services. http://www-03.ibm.com/press/us/en/pressrelease/29315.wss, February 2010.

[47] IBM. IBM iDataPlex dx360 M4 server. http://www-03.ibm.com/systems/x/hardware/rack/dx360m4/, 2012.

[48] IBM Research. IBM's history and future in water cooled computing. http://www.dipity.com/ibm_research/IBMs-History-in-Walter-Cooled-Computing/, 2012.

[49] Engin Ïpek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 195–206. ACM, 2006.

[50] P.J. Joseph, Kapil Vaswani, and Matthew J. Thazhuthaveetil. A predictive performance model for superscalar processors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 161–170. IEEE Computer Society, 2006.

[51] Shoaib Kamil, Leonid Oliker, Ali Pinar, and John Shalf. Communication requirements and interconnect optimization for high-end scientific applications. *IEEE Transactions on Parallel and Distributed Systems*, 21:188–202, 2010.

[52] J. Kim, W.J. Dally, and D. Abts. Flattened butterfly: a cost-efficient topology for high-radix networks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, pages 126–137. ACM, 2007.

[53] J. Kim, W.J. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 77–88. IEEE Computer Society, 2008.

[54] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 249–258. ACM, 2007.

[55] Leibniz-Rechenzentrum. SuperMUC. http://www.lrz.de/services/compute/supermuc/.

[56] Charles E. Leiserson. Fat-trees: universal networks for hardware-efficient supercomputing. *IEEE Transactions on Computers*, 34:892–901, October 1985.

[57] Mark Sh. Levin. Combinatorial optimization in system configuration design. *Automation and Remote Control*, 70(3):519–561, 2009.

[58] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[59] John D. McCalpin. Toward a grand unified theory of high performance computing. Technical report, IBM HPC Systems Scientific Computing (ScicomP) User Group Meeting, August 2004. http://www.cs.virginia.edu/~mccalpin/SCICOMP10-McCalpin.pdf.

[60] John McDermott. R1: A rule-based configurer of computer systems. Technical Report CMU-CS-80-119, Carnegie-Mellon University, April 1980.

[61] Rich Miller. Inside Microsoft's Dublin mega data center. https://www.datacenterknowledge.com/inside-microsofts-dublin-mega-data-center/, 2009.

[62] Sanjay Mittal and Felix Frayman. Towards a generic model of configuration tasks. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, volume 2, pages 1395–1401, 1989. http://ijcai.org/Past%20Proceedings/IJCAI-89-VOL-2/PDF/087.pdf.

[63] Jayaram Mudigonda, Praveen Yalagandula, and Jeffrey C. Mogul. Taming the flying cable monster: A topology design and optimization framework for data-center networks. In *Proceedings of USENIX Annual Technical Conference*, ATC'11, June 2011.

[64] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Producing wrong data without doing anything obviously wrong! In *ACM Sigplan Notices*, volume 44, pages 265–276. ACM, 2009.

[65] Rajesh Nair. Airflow issues: Silent enemy of efficient cooling. https://www.datacenterknowledge.com/archives/2011/05/19/airflow-issues-silent-enemy-of-efficient-cooling/, 2011.

[66] National Supercomputing Center in Tianjin (NSCC). Tianhe-1A. http://www.nscc-tj.gov.cn/en/.

[67] Javier Navaridas and Jose Miguel-Alonso. Indirect cube: A power-efficient topology for compute clusters. *Optical Switching and Networking*, 8(3):162–170, 2011.

[68] Javier Navaridas, Jose Miguel-Alonso, Francisco Javier Ridruejo, and Wolfgang Denzel. Reducing complexity in tree-like computer interconnection networks. *Parallel Computing*, 36(2-3):71–85, 2010.

[69] Javier Navaridas, Jose Antonio Pascual, and Jose Miguel-Alonso. Effects of job and task placement on parallel scientific applications performance. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 55–61. IEEE, 2009.

[70] James Niccolai. Dell warranties servers for 'fresh-air cooling'. https://www.computerworld.com/s/article/9218709/Dell_warranties_servers_for_fresh_air_cooling_, 2011.

[71] José Noudohouenou, Vincent Palomares, William Jalby, David C. Wong, David J. Kuck, and Jean Christophe Beyler. Simsys: a performance simulation framework. In *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. ACM, 2013.

[72] Oak Ridge National Laboratory (ORNL). Titan. http://www.olcf.ornl.gov/titan/.

[73] Open Compute Project. Open rack. http://www.opencompute.org/projects/open-rack/, January 2013.

[74] Philip M. Papadopoulos, Mason J. Katz, and Greg Bruno. NPACI rocks: Tools and techniques for easily deploying manageable Linux clusters. *Concurrency and Computation: Practice and Experience*, 15(7-8):707–725, 2003.

[75] Mike Parker and Steve Scott. The impact of optics on HPC system interconnects. In *17th IEEE Symposium on High Performance Interconnects*, pages 143–147. IEEE, 2009.

[76] Partnership for Advanced Computing in Europe (PRACE). Procurement strategy. www.prace-ri.eu/IMG/pdf/D7-6-2.pdf, December 2008.

[77] Douglas M. Pase and Hari Reddy. Tuning the performance of ANSYS FLUENT on IBM BladeCenter HS21 and HS21 extended memory. ftp://public.dhe.ibm.com/eserver/benchmarks/wp_Fluent_021307.pdf, February 2007.

[78] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacher, et al. Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB//CSD-02-1175, UC Berkeley, March 2002.

[79] F. Petrini and M. Vanneschi. k-ary n-trees: High performance networks for massively parallel architectures. In *Parallel Processing Symposium, 1997. Proceedings., 11th International*, pages 87–93. IEEE, 1997.

[80] Emerson Network Power. Liebert APM on-line UPS, 15-90kW. http://www.emersonnetworkpower.com/en-US/Products/ACPower/NetworkUPS/Pages/LiebertAPMOn-lineUPS15-90kW.aspx.

[81] Prometeus GmbH. TOP500 supercomputing sites. http://www.top500.org.

[82] Kelly Quinn, Daniel Fleischer, Jed Scaramella, and John Humphreys. IDC whitepaper: Forecasting total cost of ownership for initial deployments of server blades. ftp://ftp.compaq.com/pub/products/servers/blades/idc-tco-deployment.pdf, June 2006.

[83] RSC Group. New Russian RSC Tornado SUSU energy efficient supercomputer. http://rscgroup.ru/en/news/44/, November 2012.

[84] Gilad Shainer, Tong Liu, Jeff Layton, and Onur Celebioglu. LS-DYNA best-practices: Networking, MPI and parallel file system effect on LS-DYNA performance. In *Proceedings of the 11th International LS-DYNA Users Conference*, Deaborn, Michigan, June 2010. http://www.mellanox.com/pdf/whitepapers/wp_LS-DYNA_Best_Practices.pdf.

[85] Stephen Shankland. Google uncloaks once-secret server. http://news.cnet.com/8301-1001_3-10209580-92.html, April 2009.

[86] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ACM SIGARCH Computer Architecture News*, volume 30, pages 45–57. ACM, 2002.

[87] A. Singla, C.Y. Hong, L. Popa, and P.B. Godfrey. Jellyfish: Networking data centers randomly. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 17–17. USENIX Association, 2012. https://www.usenix.org/system/files/conference/nsdi12/nsdi12-final82.pdf.

[88] Konstantin S. Solnushkin. ANSYS Fluent simple performance model at ClusterDesign.org. http://clusterdesign.org/ansys-fluent-simple-performance-model/.

[89] Konstantin S. Solnushkin. Fat-tree and torus network design tool at ClusterDesign.org. http://clusterdesign.org/fat-trees/.

[90] Konstantin S. Solnushkin. Combinatorial design of computer clusters. In *Proceedings of the International Supercomputing Conference*, ISC'11, June 2011.

[91] Konstantin S. Solnushkin. Computer cluster design automation using web services. In *Proceedings of the International Supercomputing Conference*, ISC'12, June 2012.

[92] Konstantin S. Solnushkin. Fruits of computing: Redefining 'Green' in HPC energy usage. http://clusterdesign.org/2012/08/fruits-of-computing-redefining-green-in-hpc-energy-usage/, August 2012.

[93] Konstantin S. Solnushkin. Automated design of torus networks. http://arxiv.org/abs/1301.6180, January 2013. arXiv:1301.6180 [cs.DC].

[94] Konstantin S. Solnushkin. Automated design of two-layer fat-tree networks. http://arxiv.org/abs/1301.6179, January 2013. arXiv:1301.6179 [cs.DC].

[95] Konstantin S. Solnushkin. SADDLE: A modular design automation framework for cluster supercomputers and data centres. In *Supercomputing*, volume 8488 of *Lecture Notes in Computer Science*, pages 232–244. Springer International Publishing, June 2014. arXiv:1408.1127 [cs.DC].

[96] Dina Spector. This supercomputer is the world's most powerful hurricane prediction machine. http://www.businessinsider.com/climate-data-supercomputer-yellowstone-2012-11, November 2012.

[97] Shawn Strande. Gordon – design and performance of a 3D torus interconnect for data intensive computing. In *Proceedings of HPC Advisory Council Held in Conjunction with the International Supercomputing Conference*, 2012. www.hpcadvisorycouncil.com/events/2012/European-Workshop/Presentations/4_SDSC.pdf.

[98] S.M. Strande, P. Cicotti, R.S. Sinkovits, W.S. Young, R. Wagner, M. Tatineni, E. Hocks, A. Snavely, and M. Norman. Gordon: design, performance, and experiences deploying and supporting a data intensive supercomputer. In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment: Bridging from the eXtreme to the campus and beyond*, page 3. ACM, 2012.

[99] Erich Strohmaier and Hongzhang Shan. Apex-Map: A global data access benchmark to analyze HPC systems and parallel programming paradigms. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 49–49. IEEE, 2005.

[100] H. Subramoni, S. Potluri, K. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A. Moody, and DK Panda. Design of a scalable InfiniBand topology service to enable network-topology-aware placement of processes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 70. IEEE Computer Society Press, 2012.

[101] Zhiqi Tao, Andreas Dilger, Eric Barton, and Bryon Neitzel. Architecting a high performance storage system. http://www.whamcloud.com, January 2012.

[102] Texas Advanced Computing Center (TACC). Stampede. http://www.tacc.utexas.edu/stampede.

[103] The Green Grid. Quantitative analysis of power distribution configurations for data centers. http://www.thegreengrid.org/Global/Content/white-papers/Quantitative-Efficiency-Analysis, December 2008.

[104] My Ton, Brian Fortenbery, and William Tschudi. DC power for improved data center efficiency. Technical report, Lawrence Berkeley National Laboratory, March 2008.

[105] TOP500 List. Jaguar, a supercomputer by Cray Inc. http://top500.org/system/176544, November 2009.

[106] TOP500 List. Lomonosov, a supercomputer by T-Platforms. http://top500.org/system/176549, November 2009.

[107] TOP500 List. Yellowstone, a supercomputer by IBM. http://top500.org/system/177827, November 2012.

[108] Jan Treibig, Georg Hager, and Gerhard Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 207–216. IEEE, 2010.

[109] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. Multi2Sim: A simulation framework for CPU-GPU computing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 335–344. ACM, September 2012.

[110] US Department of Defence High Performance Computing Modernization Program. High performance computing modernization program makes four petaFLOPS upgrade for Department of Defense. http://www.hpcmo.hpc.mil/community/PRESS%20RELEASE/HPCMP-TI%20NewsRelease%20FINAL%20national%205-21%20corrections.pdf, May 2012.

[111] US Energy Information Administration. Electric power monthly. http://www.eia.gov/electricity/monthly/, May 2013.

[112] Nagarajan Venkateswaran, Aravind Vasudevan, Balaji Subramaniam, Ravindhiran Mukundrajan, T. Ramnath Sai Sagar, Madhavan Manivannan, Sriram Murali, and Vinoth Krishnan Elangovan. Towards modeling and integrated design automation of supercomputing clusters (MIDAS). *Computer Science – Research and Development*, 24:1–10, 2009. 10.1007/s00450-009-0085-5.

[113] J. Ward, M. O'Sullivan, T. Shahoumian, J. Wilkes, R. Wu, and D. Beyer. Appia and the HP SAN designer: automatic storage area network fabric design. In *Proceedings of HP Technical Conference*. Hewlett-Packard, 2003.

[114] Andy Woods. Cooling the data center. *Communications of the ACM*, 53(4):36–42, 2010.

[115] Matt T. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 23–34. IEEE, 2007.

[116] Eitan Zahavi. Fat-tree routing and node ordering providing contention free traffic for MPI global collectives. *J. Parallel Distrib. Comput.*, 72(11):1423–1432, November 2012.