TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Rechnertechnik und Rechnerorganisation /
Parallelrechnerarchitektur der Technischen Universität München

Designing High Performance Computing Architectures for Reliable Space Applications

Fisnik Kraja

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Martin Bichler

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Arndt Bode
2. Univ.-Prof. Dr. Xavier Martorell,
   Universitat Politècnica de Catalunya/Spanien

Die Dissertation wurde am 24.09.2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 20.11.2012 angenommen.

Abstract

Future space applications will demand for computing architectures with high performance capabilities. In order to improve the on-board computing power, Commercial Off The Shelf (COTS) multi- and many-core processor technologies have to be introduced in the design process of spacecraft computing platforms. Such technologies will be able to reduce the performance gap between on-board and ground computing platforms without increasing development costs. COTS products can be easily obtained and integrated on computing platforms, but they are susceptible to the radiation environment, which is quite severe to novel circuit technologies based on very small sized transistors.

This dissertation proposes a high performance architecture design for future reliable space applications. To address system reliability, traditional hardware redundancy techniques have to be combined with software implemented fault tolerant ones. The configuration flexibility provided in the new architecture design, enables each platform designer to build customized systems ranging from high reliable to high performance ones.

Different high performance systems are benchmarked to further assist on-board platform designers in architecture specification and implementation decisions. A Synthetic Aperture Radar (SAR) application is developed for this purpose. It is further optimized and parallelized for efficient execution on shared memory, distributed memory, and heterogeneous systems. It turns out that performance of such an application can be really increased up to the desired level on these parallel systems. Heterogeneous systems with shared memory multi-core processors and many-core accelerator modules provide most efficient results in terms of performance per platform size and power consumption.

To Fjoralba,

my beloved wife,

without whom this work would not have been possible.

Acknowledgments

A lot of people contributed to the achievements of this dissertation. First of all, I would like to thank Prof. Dr. Arndt Bode for offering me the possibility to work in a supportive working environment at the Chair of Computer Architecture (LRR - Lehrstuhl fur Rechnertechnik und Rechnerorganisation) at Technische Universität München (TUM). As my first adviser, Prof. Bode encouraged me to work with parallel architectures and programming models and granted me the freedom to research different aspects during these years.

I am particularly thankful to Prof. Dr. Betim Çiço for helping me start my PhD Studies by for creating with Prof. Bode an academic relationship that soon turned into a strong collaboration possibility. I would also like to express my gratitude to my second advisor Assoc. Prof. Xavier Martorell, who supported me especially in the final period of my studies. My gratitude goes also to Dr. Georg Acher, who accompanied and supported me in my work and with whom I have closely collaborated. I am also grateful to all my other colleagues at LRR who were always there when they were needed. A big thank you goes to my family. Without their encouragement and support, these studies would not have been possible.

My PhD studies were partly funded by the Albanian Excellence Fund and partly funded by the Munich Center of Advanced Computing. Albanian Excellence Fund is granted by the Albanian Ministry of Education and Science as a special fund for doctorate and post doctorate studies at best universities all over the world. Munich Center of Advanced Computing is a research entity established at TUM to research various activities related to High Performance Computing and Computational Science and Engineering.

Contents

1 Introduction

This chapter describes the demand for efficient high performance architectures to support the execution of applications in space. It introduces challenges that spacecraft platform designers will have to face in the near future. The scientific contribution is summarized in the methods used to address and face those challenges. This chapter concludes with the structure of the dissertation.

1.1 Motivation

Current space applications play a crucial role in our daily life in security and defense, observing and protecting the environment, in scientific and technical advances and in telecommunication utilities. Traditionally, they have been remote control platforms with all major decisions made by control centers on Earth. In order to deal with high-reliability requirements and other constrains like power and size, on-board computers provided minimal functionality. New space applications demand improved on-board processing capabilities in terms of high processing power and throughput without losing the required reliability. Downlink bandwidth to Earth is very limited for on-board systems.

Taking into consideration that sensor data rates and resolutions are increasing drastically, it is no longer viable to process data remotely on Earth. Indeed, sensor data must be processed where it is captured so that less data gets downlinked to Earth. Even for on-board autonomous processing and control systems it is often not viable to apply remote control from Earth due to propagation delays and bandwidth limitations. On the other side, spacecrafts require their autonomy, which means they might require additional processing power for decision-making.

It is impossible to face such challenges with the current space technology because platform designers have to dedicate resources to techniques dealing with radiation effects coming from the hazardous radiation environment in space. In general, platforms should be designed with limited power, weight, size, and cooling capability in mind. Traditional space technologies are also severely limited, which makes their use impossible for the current purpose.

Other than this, the potential for long time missions with different kinds of needs and requirements brings into existence the demand for powerful adaptive technologies that ensure high levels of reliability and availability. Radiation hardened components have to be combined and supplemented with commercial products to improve performance without increasing costs. Other design goals like modularity, portability, and scalability should be taken into consideration too.

1.2 Design Goals

Meeting above mentioned demands requires a system architecture that can deliver high computational power and memory bandwidth while maintaining correct operation in the space environment. The objective of this work is to propose an architecture that meets requirements for performance, reliability, and low cost by using commercial standards, design techniques, and semiconductor technologies. The following design goals are defined for the architecture:

1. The use of *Commercial Off The Shelf (COTS)* components to reduce costs while increasing performance. Many COTS products provide high performance and can be adopted for space applications in a very short time and with low costs. In order to meet high performance computing requirements, the architecture has to be designed based on multi-core or many-core computing technologies.

2. *Reliability.* The architecture should be reliable by means of hardware and software algorithms and techniques. The architecture should perform required functions under stated conditions for a specific period of time. This means that the rate between performance and reliability will be defined by the system designer, who should decide how many resources are going to be used for the required level of reliability.

3. *Portability.* The architecture should be portable across a wide range of spacecraft platforms and applications and not only available for a specific mission. This implies that the architecture should be general and not customized or tuned to a specific application.

4. *Modularity.* The architecture implementation should allow component change and reuse to reflect updates that might be needed when used for different missions and in different spacecraft platforms. There is a rapid change in the field of computer architectures technology. The architecture should be modular to make use of new emerging technologies, especially in the field of computing technologies. Another advantage of modularity is the fact that benchmarked and tested components of the architecture can be later used in other designs. The architecture should provide interfaces that simplify the integration of the whole system and enable the reuse of commercial components.

5. *Hardware Scalability.* The architecture should be usable and easy upgradable to different kind of applications without many hardware and software changes. Different applications have different requirements in terms of processing power, memory and interconnect bandwidth capacity. The architecture should have a scalable design, in which components can be added and removed according to the needs of the specific application and mission.

6. *Programmability.* The architecture should be easy to program and compatible with different programming environments. This will increase programmer's productivity because each of them will choose the most suited programming model. In heterogeneous architectures specific programming models might provide performance but generic ones provide flexibility and productivity in programming. This has to be studied further for space applications.

7. *Efficiency.* The architecture should perform efficiently in terms of power consumption and size. Applications running on computing architectures should fully exploit computational resources. In order to achieve this on parallel architectures, applications have to be parallelized and tuned to the specific architecture. Benchmarking parallel architectures on Earth with space applications will give an idea on how these kinds of applications can be parallelized to harness efficiently the computational power.

1.3 Scientific Contribution

This dissertation comprises scientific contributions to Aerospace and Computer Science communities. Its main contributions are:

1. The proposal for a high performance computing architecture for future reliable space applications that will require extended on-board processing capabilities.
2. The development of a real Two-Dimensional Spotlight Synthetic Aperture Radar (2DSSAR) application, which can be configured to mimic different computational requirements typical a wide range of space applications.
3. The acquisition of performance evaluations that help in further specifications of the proposed architecture. Results are obtained by benchmarking various High Performance Computing (HPC) platforms on Earth. For benchmarking purposes 2DSSAR application is optimized and parallelized using different programming models.

First, strategies for designing reliable systems are analyzed and most suited ones are selected to be applied in the proposed architecture. Different space applications are analyzed too. Respective processing requirements are extracted and used as design goals for the proposed architecture. In order to reach those goals, computational requirements for on-board processing and interconnect bandwidth capacities are taken into consideration. The proposed architecture combines in a hybrid fault-tolerant system, redundant hardware components with a reliable software architecture. Being focused mainly on the hardware side, further estimations are given for interconnect and processing technologies.

Second, a spotlight Synthetic Aperture Radar (SAR) application that implements a spatial frequency interpolation algorithm for image reconstruction is developed for benchmarking purposes. The proposed architecture targets different space applications, but in this dissertation the SAR family of applications is chosen for benchmarking as one with the highest computational requirements in the range of space applications. Especially the image reconstruction stage of 2DSSAR is used to mimic requirements and challenges that novel SAR applications pose to spacecraft designers. The goal is to create a benchmarking application representative of on-board sensor processing. As such, it has been implemented so that it can be scaled and tuned to match different computing requirements.

Third, various HPC architectures and environments have been considered as possible candidates for the proposed architecture. They have been analyzed, described, benchmarked, and profiled according to their main features. Obtained results reflect hardware architectural features as well as application programmability ones. Application scalability is one of the main objectives on all benchmarked parallel architectures. As part of the parallel architecture environment, different programming models have been analyzed and chosen to be used in application parallelization. In this way, not only processing technologies, but also interconnect and memory ones are benchmarked.

The novelty in this research stays in the proposed architecture and in the parallel implementation of the SAR application used to benchmark HPC platforms in order to identify features that suit the best to SAR processing. The proposed architecture is a combination of radiation hardened components and COTS products that can provide a reliable solution for a required level of performance under limited costs. It is the first time a parallelized SAR application is implemented and benchmarked on emerging HPC architectures and environments, including here shared and distributed memory ones based on multi-core processors and many-core accelerator modules.

1.4 Structure of the Dissertation

This dissertation examines challenges of current and future space applications and proposes a comprehensive approach to deal with them. As possible parts of this solution, different software and hardware technologies are discussed and introduced for the design process of future space computing platform. The content of this dissertation is organized in 8 chapters.

Chapter 2 provides background information on HPC systems, reliable computing, and space applications. It emphasizes challenges in the design process from different points of view related to environment limitations and future space applications. As part of related work, a short history of reliable computers for space is given and novel approaches that deal with recent challenges are discussed and compared towards the approach taken in this dissertation.

This chapter begins with a discussion on shared memory, distributed memory and heterogeneous systems, which is followed by a description of strategies that are used to design reliable systems and ends with a discussion on current hardware and software systems that successfully use those strategies in real spacecraft computing platforms. The concluding part of this chapter provides information on current and future space applications. It describes some of the most challenging ones. To emphasize future space application challenges, this part specifies computational requirements of a novel space application, namely the High Resolution Wide Swath (HRWS) SAR.

Chapter 3 proposes a solution to face challenges in the design process related to environmental limitations and future space applications. It proposes a reliable high performance computing architecture that combines hardware redundancy techniques and fault-tolerant software approaches with COTS high performance computing technologies. It explains methods that can be used to meet all design goals presented in section 1.2 on page 2. The proposal of the architecture is published in [1].

As a general architecture proposal, Chapter 3 does not provide a detailed system specification, but possible alternatives for spacecraft platform designers. These alternatives have to be further studied, analyzed, and compared so that the most suitable one gets selected for a specific application. The best way to estimate computing platforms is by benchmarking.

Chapter 4 discusses the 2DSSAR application that is used to benchmark various HPC systems on Earth. The first two sections describe 2DSSAR from the signal processing point of view by giving basic spotlight SAR formulations and by discussing the spatial frequency interpolation technique. All processing steps composing 2DSSAR application map to specific signal processing steps in synthetic data generation and SAR sensor processing. The latter one applies image reconstruction via spatial frequency interpolation. The third section provides the structure of the application and describes implementation details. The concluding section lists some considerations related to the usage of C programming language for signal processing applications.

Chapter 5, Chapter 6, and Chapter 7 examine benchmarking results on three different systems, namely on shared memory, distributed memory, and heterogeneous systems. Each of these chapters begins with the introduction of each benchmarking platform. Then, steps required to port the 2DSSAR application to these platforms are discussed. Each of these chapters ends with an evaluation section that examines obtained results on each platform and also compares different incremental versions of the same application implementation. A comparison is also applied to results obtained on different systems, in order to profile them according to different performance perspectives related to power consumption and size. Discussions on parallelization techniques for shared and distributed memory platforms are published in [2], whereas evaluations of 2DSSAR on heterogeneous platforms are published in [3] and [4].

Chapter 8 provides final conclusions of the dissertation and gives an outlook on future work.

2 Background and Related Work

This chapter reviews the state of the art relevant to HPC systems and reliable spacecraft designs and applications. It begins with a discussion on common features of shared memory systems. It continues with the definition of distinctive concepts in distributed memory systems followed by an overview of the message passing paradigm, which is well-known for programming such systems. The review of HPC systems concludes with a general description of heterogeneous systems, followed by a detailed description of Graphics Processing Unit (GPU) architectures and programming paradigms.

Main design strategies for reliable computing platforms are analyzed in this chapter too. Specific attention is paid to fault-tolerance techniques. A short history of previous reliable spacecraft computers is followed by emerging approaches in reliable computing for space. This chapter concludes with a short description of current and future space applications that represent performance challenges in terms of computations, storage, and transfers.

2.1 Shared Memory Systems

Shared memory systems consist of multiple processors or processor cores that share main memory. Communication between processors occurs through the shared memory itself. This provides fast communications at the same speed as normal memory accesses occur. Shared memory systems are not always implemented with a single global shared memory, but sometimes with a physically distributed one. In the later implementation each processor has its own local memory that can be accessed by other processors too. In order to obtain very good parallel performance on both implementations, it is very important to be familiar with the memory hierarchy of the system. Shared memory systems are generally an implementation of:

1. Uniform Memory Access (UMA) architecture, in which costs of accessing main memory (access time, latency, and bandwidth) are identical for all processors and memory addresses. This is typical in a system that connects all processors to main memory over one shared bus. In large scale UMA architectures, memory bandwidth can become a bottleneck as it is shared between all processors. UMA architectures are also referred as Symmetric Multiprocessing (SMP) architectures due to the uniform memory access time.

2. Non-Uniform Memory Access (NUMA) architecture, which has been created to overcome memory bandwidth bottlenecks. In such a system there is neither a single global main memory nor a shared bus. Instead, each processor has its own local memory. However, the address space is global and the local memory of each processor can be accessed remotely by all other processors. Remote memory accesses

certainly introduce higher latencies. This means that high bandwidth can be utilized only if memory accesses are scattered over all processors.

2.1.1 Chip Multi Processors (CMPs)

Processor performance in the past has been increased either by increasing switching frequency or by increasing the number of transistors. To ensure stable operation when increasing frequency, the required voltage needs to be increased too. Processor power consumption [5] can be calculated as:

$$p = c \times v^2 \times f \tag{2.1}$$

where c is the capacitive load, v is the voltage, and f is the transistor switching frequency. This means that switching frequency impacts directly the power consumption of a processor. Additional power is needed to dissipate the heat generated from this power. At some point, increasing switching frequency stopped due to the so called *power wall* and manufacturers started to increase the number of transistors they could integrate on die. At first, additional transistors were used to increase caches and to add more functional units to each processor. Bigger caches do not improve processor performance, but increase application performance by hiding latencies to main memory.

Having more functional units in a processor enables Instruction Level Parallelism (ILP). To apply transparent ILP, the processor should be able to apply additional techniques like pipelining, out-of-order execution, multiple-issue, dynamic scheduling and speculative execution. Indeed, these techniques increase the complexity of the processor's microarchitecture. Data level parallelism can also be exploited through redundant functional units that execute the same instruction on multiple inputs. Data level parallelism is usually detected by the compiler and is supported by most superscalar processors that provide Simple Instruction Multiple Data (SIMD) units that operate only on vectors of two, four or eight values and allow for parallelization at a very fine grained level.

However, many applications exhibit coarse grained parallelism with independent parts that can execute in parallel. As the execution can run in separate threads it is known as Thread Level Parallelism (TLP). TLP increases utilization of redundant functional units by interleaving separate instructions from multiple threads into a single stream that is processed in parallel through ILP techniques. This combination of TLP and ILP is called Simultaneous Multithreading (SMT) as it enables each processor to be seen as two logical processors by the operating system, which schedules threads or processes to execute in parallel on a single processor with redundant functional units.

Power limitations have pushed manufacturers to lower processor complexity. To further increase performance they decided to integrate multiple processor cores on a single chip. This introduced chip multi processors or multi-core processors that enable the design of power efficient and high performance computing systems. The main problem is that applications need to be parallelized at thread level in order to exploit the computational power provided by all processor cores in the chip.

2.1.2 The Memory Hierarchy

Performance of DRAM-based main memories is not on the same level with computing capabilities of modern processors. This problem is known by the term *memory wall*, which represents the growing speed gap between processor and memory outside the processor chip. In order to mitigate the impact of the *memory wall*, small caching memories have been introduced in the hierarchy between processor and main memory. There exist several levels of caches as it is very expensive to implement a fast cache memory with high capacity. Usually the cache at the lowest level (nearest to the processor) is small and fast, caches at higher levels are bigger and slower. Processor registers are cached by the Level 1 Data (L1D) cache and micro-instructions by the Level 1 Instruction (L1I) cache. In this way, program code is not evicted from cache due to data reads. Each of them is typically a few kilobytes large and is cached by the Level 2 cache which is usually unified (L2: data and instructions).

Some architectures have also a Level 3 cache (L3) of a few megabytes. Depending on the type of processor architecture, some levels of cache hierarchy are shared by several cores while others are not. For example, in a multi-core processor each core might have its own L1 caches, but shares the L2 cache with another core. In another architecture, each core might have its own L1 and L2 cache but shares the L3 cache with other cores in the chip. Shared caches on multi-core architectures accelerate communication between threads as data does not have to be exchanged through the slow main memory but can be exchanged in a shared cache at much lower latency.

The introduction of caches in the memory hierarchy creates the problem of cache coherency. In a multiprocessor environment, it happens that a processor writes to a memory location of which another processor holds a copy in its cache. The purpose of cache coherence protocols is to guarantee that all processors of a system have the same view over main memory. Snooping protocols are used in small shared memory systems, whereas directory-based protocols are used for better scalability in large systems. With snooping protocols, each processor watches the memory bus for write transactions that affect cache lines for which it holds copies and updates its caches consequently. In larger systems, a global directory keeps track of contents of all caches. Upon a write transaction the directory either updates or invalidates all caches that hold a copy of the respective cache line.

2.1.3 Programming Shared Memory Systems

The main programming constructs in shared memory environments are threads. They are basically provided by the operating system for each specific process. The cost of creating and managing threads is quite low compared to the overhead of creating and managing processes. All threads in a single process share the same virtual address space, on contrary to processes that usually cannot access each other's address space. Individual threads communicate with each other by writing and reading shared variables to and from the shared memory region. Since all threads share the same address space, all application data is accessible from all threads and there is no need to exchange data between threads. This simplifies data level parallelization as threads can read data from shared memory and write back results. However, thread synchronization operations are needed to avoid data race conditions that might impact result's correctness.

POSIX Threads or Pthreads [6] are implementations of the IEEE POSIX 1003.1c standard that specifies an Application Programming Interface (API) for threads in C programming language for UNIX systems. Pthreads API provides subroutines that are responsible for thread management, mutex synchronization, thread communication controlled by condition variables, and thread synchronization by locks and barriers. Working with Pthreads API is a bit complicated as it requires a few programming efforts, but it provides full control over threads.

Open Multi-Processing (OpenMP [7]) is a much simpler API that provides methods to create, manage, synchronize, and destroy threads. OpenMP is composed of a set of compiler directives (pragmas) and library routines that extend C/C++ and Fortran programming languages. In order to use this API, an OpenMP-enabled compiler is needed. The programmer has to tell the compiler where to find program regions that can be executed in parallel. Normal loop constructs can be easily parallelized by specifying the loop body as a parallel region. Loop iterations are then distributed over multiple threads and executed in parallel. This happens in a fork-join paradigm, in which new threads are forked as soon as the parallel region is entered, and joined again at the end of the region. In terms of required programming efforts needed to parallelize an existing sequential application, OpenMP does not require major code changes. It gives the opportunity to decide which part of the code is most computationally intensive so that it can be parallelized in order to achieve good performance in less efforts.

Parallel performance in UMA architectures depends mostly on hardware resources and not so much on programmer's skills because the programmer can easily exploit data level parallelism without communication overhead. The main problem is the bottleneck in memory bandwidth, which can reduce memory performance and complete system performance. Main metrics of memory performance are latency and bandwidth. For applications with regular access patterns, memory latency can be hidden by caches combined with a technique called *pre-fetching.* If the cache controller in each processor detects a pattern in memory accesses, it will load in advance cache lines that will probably be accessed next. Memory bandwidth bound applications have regular memory access patterns, but either their data set does not fit into cache or their data access pattern does not have temporal locality. Such data set can be divided into smaller parts that can fit in cache, by using *blocking* technique. In this way memory bandwidth requirements can be reduced by reusing data residing in cache. This is true only for applications that exhibit temporal locality.

Memory bandwidth problems in NUMA architectures can be avoided by scattering memory accesses. This is where the programmer comes more into play. In order to obtain good parallel performance on NUMA architectures, the programmer should scatter memory accesses and in the same time reduce remote memory accesses. This can be achieved by data partitioning schemes that make sure that most frequent accesses take place in local memory. Some tricks that help achieving such schemes are: *thread pinning technique* and *first touch policy.* Thread pinning technique binds threads to a specific core in order to reduce possible thread migrations that might bring additional remote memory accesses. First touch policy assures that each thread initializing a data object gets the page associated with that data item in the local memory of the processor the thread is executing on. This approach works very well for applications where data elements are updated by the same thread. If the data access pattern is not uniform, first touch policy might not help, but on the contrary it might lower the performance.

When none of the above mentioned techniques helps in keeping most frequently accessed data in local memory, it might be a good idea to apply distributed memory programming models. In such programming models, each process has its own local data that resides on the local memory associated with the core where the respective process runs. Such models are generally used in distributed memory systems, where there is no shared memory and the communication takes place over the network. Distributed memory systems are further discussed in section 2.2. Applications that are parallelized for distributed memory systems can execute on shared memory systems too. The most common distributed memory programming model is the Message Passing Interface (MPI [8]). On shared memory systems, MPI is implemented to perform message passing between processor cores by using the shared memory instead of the network. In distributed memory programming, the programmer has to take care of workload and data distribution. This makes it a more complex process, but that sometimes provides better application scalability.

Shared memory and distributed memory programming paradigms can be combined in some very specific cases. They can be used to build a multiprocess application, in which each process is itself multithreaded. Such an approach tends to benefit the scalability feature of message passing and the parallel efficiency of the multithreaded parallel execution. Another benefit of such an approach is that it simplifies data distribution, which is considered a complex task in distributed memory programming. In multi-socket NUMA systems, hybrid MPI and OpenMP programming assures that each multi-core processor accesses the whole time local memory, and threads in each core share only this memory and not memory associated with another multi-core processor on another socket of the system.

2.2 Distributed Memory Systems

Distributed memory systems are created to support a large number of processors. Since memory is not centralized, but it is physically and logically distributed among compute nodes, the memory hierarchy is capable to fulfill bandwidth requirements of a large number of processors, trying to keep access latency times into acceptable levels. If most of memory accesses occur in local memory, memory bandwidth will scale in a cost-effective way and latency to access local memory will be low. On the other side, the communication between processors in distributed memory systems becomes more complicated. More effort is required from the programmer to exploit the increased bandwidth.

Before going further with other features, it might be better to define some terminology. The term *compute node* refers to a separate computing unit with its own private memory. It can be internally implemented as UMA or NUMA system. As such, it can integrate multiple *processor modules* that share the private memory in a multi-socket design. Each processor module can be composed of one or multiple *cores* that are integrated on the same chip. High speed interconnection technologies like QPI [9] and HyperTransport [10] are used to interface processor modules inside each node. For the interconnection of compute nodes, network interconnection technologies like Infiniband and 10 Gigabit Ethernet are used. Such technologies are discussed in [11]. As memory is physically and logically distributed along compute nodes, the address space is separated in many private address spaces, one for each compute node. In such a system, the private address space cannot be accessed from remote nodes. Hence, communication between different nodes takes place by message passing over the network, whereas the one inside each node takes place over the memory, which is shared among processor cores.

2.2.1 Programming Distributed Memory Systems

The main programming construct in distributed memory systems is the process. It represents an instance of the main program being executed. Each process can be composed of multiple threads that execute concurrently. This is the case when a process is scheduled at the node level and threads are scheduled at the core level of each node. Processes can execute in parallel on different compute nodes, on processor modules inside each node, and even on cores inside each processor. Since each process has its own address space, i.e. its own private data, they have to explicitly communicate with each other. The most common paradigm for inter-process communication is called message passing programming. In such a programming model the programmer has to distribute required data, additionally to the distribution of computations over a set of processes. Data has to be distributed explicitly and in time. If data is not received in time, the receiving process will have to wait, wasting so many CPU cycles.

Porting shared memory parallel applications to message passing paradigm can be difficult and sometimes impossible. Sometimes a hardware resource like main memory in each node limits the number of processes that can be run in one node because the total amount of data does not fit in it. If the size of the data set per each process exceeds the size of the main memory in each node, it is impossible to run this application in parallel. The only way to run such an application on distributed memory systems is to provide another view of the underlying hardware. If the application sees a distributed shared memory system, the single virtual address space over all distributed nodes will enable the execution of shared memory applications without requiring code modification. In such case the application will run, but parallel performance will depend on the performance of cache coherency protocols, which on large scale systems exhibit poor performance due to high complexity.

2.2.2 Communication Characteristics

Since communication between nodes in a distributed memory system takes place over the network, its performance impacts the overall application performance. Indeed, application performance does not depend only on network performance, but also on the application's communication pattern and on the size of messages being exchanged. Network performance is generally defined by bandwidth and delay. The Message Delivery Time (MDT) can be expressed as the sum of Message Transmission Time (MTT) and Delay (D). The delay itself can be a combination of hardware and software delays. Taking into consideration that MTT is obtained by dividing the Message Size (MS) with the Bandwidth (B), the Message Delivery Time can be formulated as:

$$MDT = MTT + D = \frac{MS}{B} + D \tag{2.2}$$

If messages are small, delivery time will mainly depend on delay. However, if messages are large, delivery time will then depend on bandwidth. This means that networking technologies with low delays are suitable for applications that continuously exchange small messages, whereas applications that exchange large data sets need high bandwidth networking technologies. This also concludes that it is very important to adopt the size of messages being exchanged according to the network technology being used.

Another relevant concept related to the communication in distributed memory systems is the network topology. Topology attributes define how compute nodes exchange data and at what cost. Most common topologies are Ring, Star, Tree, Fat-Tree, Mesh, Torus, Hypercube, Butterfly, and Dragonfly. Most important topology attributes are known as *scalability factor* and *node symmetry.* The scalability factor refers to the level of increase in communication complexity while adding nodes in the system. A highly scalable topology does not increase the logic required to implement communication when more nodes are added. A node symmetric topology has no special node. This means that each node has the same view over the rest of the topology. The problem with asymmetric node topologies is that the special node can become a bottleneck for the communication.

2.2.3 Message Passing Interface (MPI)

MPI is a message-passing API based on MPI-Forum specifications. It has become a development standard API for message-passing applications on distributed but also shared memory systems. Starting from 1994, MPI-Forum has approved several specifications that fall into two categories: MPI-1 and MPI-2. MPI-1 comprises functionalities for point-to-point and collective communications in a static runtime environment. New functionalities that cover remote memory operations, parallel Input Output (I/O), and dynamic process management have been added in MPI-2 standard specification. Not all added functionalities are widely used because some are complex, some are not supported on supercomputers, and some are not really needed in applications that have already been implemented with MPI-1. In terms of functionalities, the message passing paradigm is completely covered by MPI-1, and it is also easy to understand and use.

MPI provides environment management functions that initialize and terminate the MPI environment. Initialization has to be applied before calling any other MPI function. The communication context between processes is build based on communicators that facilitate communication. When MPI is initialized, each process becomes part of the global *MPI_COMM_WORLD* communicator, but during runtime other custom communicators can be created. Custom communicators can be used to group processes or to fit them in the network topology. Communicators represent independent communication channels with no message interference. Most of MPI functions take a communicator as an argument. Each process within a communicator has a unique identifier assigned by the system after MPI initialization. This identifier, called *rank,* is used by the programmer to specify message source and destination. The message exchange inside a communicator is ordered, i.e. messages are received in the same order they are sent.

The communication unit between processes in MPI is the message, which can contain simple pre-defined data types or complex user-defined data structures. Most MPI functions need the message length and the starting address of the buffer holding it. The programmer has to specify everything explicitly on both sender and receiver sides and to provide the buffer to store the message on receiver side. Inter-process message exchange is usually carried out through point-to-point or collective operations. Two single processes usually communicate through point-to-point operations, but when a group or all processes have to communicate, the programmer has to apply collective operations. Point-to-point operations use send and receive functions, which can be blocking or non-blocking. After a blocking send function returns it is safe to start processing and modifying the message buffer. When a non-blocking send function returns, the programmer has to check if the message transfer has already finished. Concerning receive functions, blocking ones return after the message has been received and copied to the message buffer. The non-blocking receive function returns immediately. The programmer has to know the time when the received message has to be modified and to explicitly instruct the runtime to wait until the message is completely received.

In collective operations, all processes have to call the same function. They are all blocking functions and usually have three forms: 1-to-all, all-to-1, and all-to-all. *Broadcast* and *scatter* functions apply typical 1-to-all collective operations. The broadcast function sends the same message to all processes in the specified communicator. The scatter function distributes a vector to all processes by sending to each process only one part of the vector. Important all-to-1 functions are *gather* and *reduce.* The gather function, contrary to the scatter one, enables one process to receive data from all other sending processes and to reassemble it into a single vector. Similar to gather, the reduce function collects vectors from multiple senders, but applies a reduction operation on received vectors. This reduction combines respective entries to build a single vector of the same size. In synchronous all-to-all operations, each process is a sender and a receiver at the same time. Gather and reduce functions have their respective all-to-all functions that assemble the result from all processes. The barrier function, as an all-to-all one, synchronizes all processes in one communicator. These synchronous all-to-all functions should not be widely used in order to avoid waiting times among processes.

2.3 Heterogeneous CPU/GPU Systems

As physics laws of are forcing CPU chip makers not to increase switching frequency and instead focus on increasing the number of cores, performance increases following Moore's law are no longer possible. The rapid hardware development has created a set of parallel architectures, such as multi-core CPUs from Intel [12, 13, 14] and Tilera [15], and recent GPU-based accelerators [16] from NVIDIA and AMD. Some interesting heterogeneous CPU/GPU systems on chip are created from AMD [17] and Intel [18]. Despite the development in parallel computing architectures, harnessing this computing power is missing in software due to many difficulties with writing parallel applications. This is not due to the lack of attempts to write programming languages and APIs that exploit parallelism. Some examples are MPI [8], OpenMP [7], OmpSs [19], CUDA [20], and OpenCL [21].

One of the main drawbacks of heterogeneous CPU/GPU computing is the PCI Express (PCIe) interconnection bandwidth, which can be a throughput bottleneck when a significant amount of data is transferred between host (CPU) and device (GPU). Frequent or poorly managed data movements bring bandwidth troubles. Unless the complete working set of data fits into GPU memory, PCIe might be a bottleneck. The programmer has to write algorithms so that he can reduce PCIe transfers as much as possible. Limitations on GPU memory size make this job even more difficult in cases when it is impossible to fit all the data set into it. In order to be able to build applications that execute efficiently on heterogeneous systems, the programmer has to be familiar with CPU and GPU architectures and with programming models that suit the best to such architectures. The rest of this section focuses mainly on GPU architectures and programming paradigms. CPU related aspects are already discussed in previous sections.

2.3.1 GPU Architectural Features

Modern GPUs can be considered as perfect candidates for high performance computing as many of them have now been widely used to build powerful supercomputers. They integrate a large number of processing cores with a high performance memory hierarchy. This and other features have created a large performance gap between general-purpose multi-core CPUs and many-core GPUs. The main reason for this lies in the design philosophy. The CPU is designed for optimized sequential code. It uses sophisticated control logic to increase instruction throughput, and large cache memories to hide memory access latency. Neither control logic nor cache memories increase peak computation speed. On the other side, GPUs are designed as numeric computing engines with maximized chip area dedicated to floating point calculations and very high memory bandwidth (Figure 2.1). As floating-point engines, they will not perform well on tasks optimized for CPU execution. Therefore, most applications will have to use CPUs for sequential or coarse grained parallel code and GPUs for fine grained parallel code.

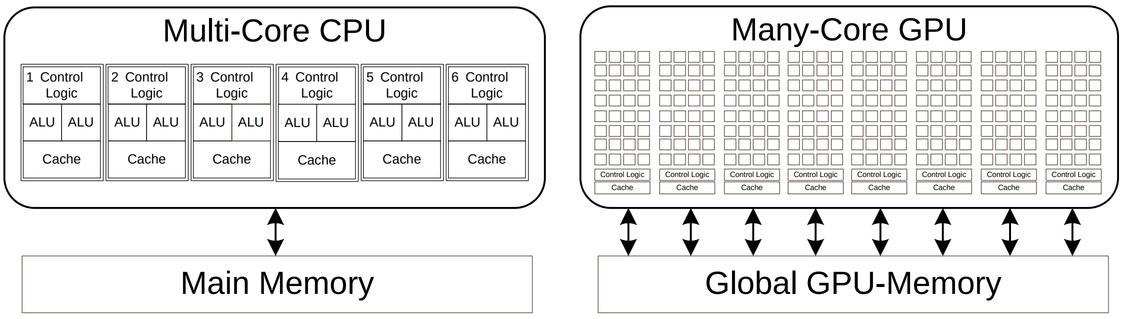


Figure 2.1: CPU and GPU Design Philosophies

Leading GPU manufactures, NVIDIA and AMD have launched many products that are similar in design, but differ in many architectural features related to processor cores and memory hierarchy. The first architectural difference between NVIDIA and AMD GPUs is that AMD ones use Very Long Instruction Word (VLIW) processors for vector processing. It might be difficult for the compiler to find enough independent instructions and to always build compact VLIW instructions. This makes application performance depend on the number of individual instructions that build the VLIW one. On the other side, NVIDIA GPUs use a SIMD approach in a multithreading execution that exploits thread level parallelism to achieve high performance. Applications with significant data dependency that are not suitable for VLIW processing, execute faster on NVIDIA GPUs. In contrast, AMD GPUs are better for applications where sufficient instructions can be found to compact VLIW slots.

The second difference is in the memory hierarchy. Both vendors take a hierarchical approach for the memory organization that consists of the caches (L1 and L2) and global memory. L1 cache on some NVIDIA GPUs is configurable to different sizes and can be disabled or enabled by a compiler flag. The L1 cache on some AMD GPUs is not so flexible and can only be used for images and constants. L2 caches are generally shared among all hardware multiprocessor units on GPUs from both vendors, but on some AMD GPUs only images and constants use the L2, whereas on most NVIDIA GPUs every global memory access goes through L2.

Concerning power consumption and energy efficiency, a modern GPU consumes probably more power than a typical CPU because it integrates a larger amount of transistors on the same chip. This high power consumption is likely to generate heat and increase costs for system cooling. This might mitigate benefits gained from performance improvements. Both NVIDIA and AMD have introduced techniques to improve energy efficiency in their products. For example, AMD PowerPlay technology reduces considerably the GPU idle power. NVIDIA PowerMizer technology is mainly used to reduce power consumption on mobile GPUs.

2.3.2 Programming Models for Heterogeneous Systems

Even programming models used to develop applications for NVIDIA and AMD GPUs are different. NVIDIA GPU developers use CUDA, whereas AMD community is focused on OpenCL. In a first comparison, one should state that CUDA can only be used to develop applications for heterogeneous systems based on NVIDIA GPUs, whereas OpenCL is a framework for developing applications that can run on different heterogeneous systems consisting of different CPUs, GPUs, and other processing modules like DSP processors and FPGAs. This makes OpenCL a very flexible programming model since OpenCL applications can be easily ported to different heterogeneous systems, but from the performance point of view CUDA is much better on NVIDIA GPUs.

Compute Unified Device Architecture (CUDA [20]) is actually an architecture which is used to implement GPUs enabled to perform both traditional graphics rendering tasks and general-purpose tasks. Computing resources were partitioned into vertex and pixel shaders in previews NVIDIA GPU generations. CUDA GPUs include a unified shader pipeline that allows each ALU to be used by an application that performs general-purpose computations. System drivers enable communication between the application and the CUDA-enabled GPU. Applications for CUDA GPUs can be programmed by CUDA C, which is essentially a handful of C extensions to allow programming of massively parallel machines like NVIDIA GPUs. CUDA provides access to the virtual instruction set and memory of the parallel computing elements in NVIDIA GPUs so that they can become available for general purpose computations.

Open Computing Language (OpenCL [21]) is a specification of a parallel computing framework for heterogeneous systems. It includes a group of C parallel extensions that support both data- and task-based parallel programming models. It enables support for multi-core CPUs, many-core GPUs, Cell/B.E., and DSP processors. OpenCL includes the language for writing kernels and the API to define and then control different heterogeneous platforms. In GPU-based heterogeneous systems, OpenCL enables the application to run non-graphical computations on GPU, thus extending GPU usage beyond graphics processing.

OpenMP Superscalar (OmpSs [19]) is another framework that uses task level parallelism to step over the programmability wall of heterogeneous systems. OmpSs uses OpenMP pragmas, a source-to-source translator and a runtime system that schedules tasks while detecting dependencies between them. Different target architectures can be specified in OmpSs, such as SMPs, GPUs, heterogeneous CPU/GPU platforms, and cluster environments. The work that is to be computed in each task has to be implemented in multiple versions, one for each target architecture. For GPU-based heterogeneous systems CUDA and OpenCL kernel implementations can be easily integrated in the OmpSs version of the application. OmpSs takes care of data movements between the CPU and the GPU, based on the data direction hints and dependency detections at runtime.

2.3.2.1 Introduction to CUDA C

CUDA C is an extension to C programming language, which provides the possibility to combine serial and parallel CPU code with parallel GPU kernels. CUDA kernels, when called are executed by parallel CUDA threads. A group of threads forms a thread block. Thread blocks are organized into a grid. Threads within the same thread block can synchronize execution and share access to the local scratchpad memory. Essential for performance is to keep in mind that shared memory is shared among all threads in a block, since they all run on the same Streaming Multiprocessor (SM).

On each CUDA device there is an array of SMs. Each SM is an instruction-fetch-execution engine. CUDA thread blocks get mapped to SMs, which have thread processors, private registers, shared memory, etc. Each SM executes a pool of warps (group of 32 threads), with a separate instruction pointer for each warp. Warps ready for execution are selected by the warp scheduler and issued to the SIMD pipelines in a loose round robin fashion that skips non-ready warps. This means that each warp instruction operates on up to 32 data items, in the absence of branches (warp divergence). CUDA programmers write code for each thread as if it runs independently, but actually instructions are broadcast to 32 threads in parallel. The programmer has to consider this in order to obtain maximal performance.

Programming in CUDA is far from being an easy task. Despite the fact that NVIDIA's terminology includes the thread concept, CUDA programming is different from programming a shared memory machine with x86 CPUs using Pthreads or OpenMP. Indeed, a first look at CUDA can make one think that it contains only a subset of OpenMP functionality. As opposed to OpenMP, which includes a multitude of advanced synchronization options, CUDA allows only for barriers within blocks of threads and atomic operations. Porting an x86 application to CUDA might be a complex task as it requires fitting the application into a completely different programming model. Performance programmers should not ignore the inherent SIMD nature of GPUs. Threads are executed in warps that receive the same instruction from the control unit. Therefore, the number of branches has to be minimal in order to avoid branch divergence that brings inefficient use of SIMD units.

To be able to exploit the computational power coming from plenty SIMD units on the GPU, locality has to be exploited properly in algorithms. This is equivalent to saying that the programmer has to be aware of the memory hierarchy and the properties of each memory in the hierarchy. Furthermore, GPUs pose an additional challenge due to their limited amount of memory; currently around 6 GB. This memory is fast, but using it implies transferring data over a slow PCIe link. It is essential to know that performance is not guaranteed in spite of the peak GPU performance, which is approximately two orders of magnitude higher than a single-core x86 CPU. Dense linear algebra applications are suitable for GPU computing. Other applications might require serious code transformations, i.e. new data structures and algorithms, to make it execute on the GPU. These are success stories of using GPUs for numerical applications. But the situation is completely different with applications not easy to express in terms of vector operations. Indirect memory accesses can also generate problems. A warp should access a contiguous memory space for optimal performance.

2.3.2.2 Introduction to OmpSs

OmpSs is a programming model developed at Barcelona Supercomputing Center (BSC) that intends to step over the programmability wall especially for heterogeneous systems. In order to achieve this, OmpSs uses task level parallelism. This kind of parallelism is implemented by OpenMP pragmas, a source-to-source translator, and a runtime system that schedules tasks while detecting dependencies between them. Knowing dependencies between tasks enables each task to execute as soon as input data is available, without having to wait for all other tasks. This overlaps computation and communication providing much better load balancing in the system. Automatic data movement between subsystems is carried out by the runtime system that detects data dependencies among tasks based on data direction hints given by the programmer.

OmpSs combines OpenMP (parallelism, loop scheduling and tasking) with StarSs [22] features to support computation/communication overlapping and heterogeneity. Even though OmpSs is based on OpenMP, it includes some differences like different execution model with heterogeneity support and extended memory and task synchronization model. In the OmpSs execution model, OpenMP parallel directives are ignored. All threads are created at startup, but only one of them starts executing the main function. All threads in the team can get work from the task pool. In the memory model of OmpSs, a single naming space exists from the programmer point of view. From the runtime point of view different possibilities exist. In an SMP environment there is a single address space. In distributed and heterogeneous environments multiple address spaces exist. In such environments, versions of the same data may exist in different address spaces.

The OmpSs unit of computation is the task. Tasks can be defined by inline pragmas or by attaching pragmas to a function definition. These pragmas contain clauses (*input, output, inout*) that express data directions. Taking into consideration these clauses, data dependencies are calculated at runtime. The *copy_deps* clause instructs the runtime system to transfer associated input/output data in and out. The *target* directive is used to specify device specific information for heterogeneity support. The *device* clause specifies the type of the environment (SMPs, GPUs, heterogeneous SMP/GPU and cluster). With the *implements* clause different implementations can be specified for different kind of architectures. This allows for different task implementations for different heterogeneous systems or even for different subsystems within the same one.

OpenMP *taskwait* directive is used for synchronization purposes. Such a directive suspends the current task until all children tasks are completed. By default in a heterogeneous system, all data in the device is synchronized also with the data in the host. This behavior can be relaxed by using the *taskwait on (data)* directive that will synchronize only the specified data or *taskwait noflush* that will not copy or move any date between host and device.

OmpSs environment is composed of the Mercurium compiler and the Nanos runtime. Mercurium compiler takes the above constructs and transforms them into calls to runtime, while doing code restructuring for different target devices. Nanos runtime supports different programming models like OpenMP, OmpSs, and StarSs. It includes independent components for thread, task, and dependency management. After tasks are generated, dependencies are analyzed before they are scheduled for execution.

2.4 Designing Reliable Spacecraft Systems

Spacecraft systems have to operate reliably for long periods with little or no maintenance. The faults rate in modern electronic systems is impacted from factors like high complexity, small transistor sizes, high switching frequencies and low voltage levels [23]. This problem becomes even more obvious, when taking into consideration radiation effects on Integrated Circuits (IC) in the rough space environment. The radiation environment in space is composed of various particles generated by sun activity [24, 25], which could be charged particles like electrons, protons and heavy ions. Another form is the electromagnetic radiation, caused by self-propagating waves in vacuum or matter like x-ray, gamma-ray and ultraviolet light. Some of the most disturbing radiation effects [26, 27] on electronic equipments are:

1. *Single Event Upset (SEU)* is a bit-flip in a memory element caused by a change in the state of the transistor when hit by an energetic particle. This becomes even more dangerous when it is transformed to *Multiple Bit Upset (MBU).*
2. *Single Event Transient (SET)* are transient current pulses generated when a charged particle hits a combinatorial block. If this pulse propagates fast enough it can become SEU.

3. *Single Event Latch-up (SEL)* is the activation of new paths between transistors caused by a current spike. This turns the new circuit fully-ON causing a short connection and after that the current might burn the device.

4. *Total Ionizing Dose (TID)* is the cumulative long term ionizing damage due to protons and electrons. Since Earth observation satellites fly on low orbit there is not so much radiation to be accumulated due to planetary and Earth's magnetic field that forms radiation belts with trapped electrons and protons (no heavy ions).

Reliable operation in space is possible when systems are designed to either avoid failures by using highly reliable parts (shielding or radiation hardened components), or tolerate failures so that the system can continue operation untroubled in their presence. It is not so easy to decide which strategy or combination of strategies to use in a particular design. To simplify this decision making process, different fault-avoidance and fault-tolerance techniques have to be studied and analyzed.

First, let's define some terms concerning fault-tolerant computing [28]. A *defect* is a physical anomaly in a device that can or cannot cause a failure. A *failure* is the divergence of a device from specified characteristics. A *fault* reflects the effect of failure on logical signals. An *error* is the exposure of a fault within a program or data structure. *Transient* errors occur temporarily in the system and are usually caused by interference. *Permanent* errors occur when a part fails completely and needs to be replaced. *Fault-tolerant computing* is defined as the correct execution of a specified algorithm in the presence of failures [29]. Errors caused by these failures can be overcome by using *redundancy* [30], which can either be *temporal* (repeated in time) or *physical* (replicated hardware or software). The redundant information produced in each method can be used in detection and possible correction of errors in the system output. An error can be detected, when an inconsistency is discovered among outputs. With additional redundancy, errors can be corrected, the system can be reconfigured or errors can be masked so that correct operation can continue. Nevertheless, redundancy increases costs, size, weight and power consumption.

2.4.1 Designing Reliable Systems

There are different strategies used in the design of highly reliable systems (Figure 2.2 [31]). Main ones can be classified as follows:

1. *Fault-avoidance strategy* is used to reduce the probability that a fault occurs by using conventional design practices, such as the integration of highly reliable or radiation-hardened components, the application of shielding or the selection of special fabrication techniques.

2. *Error detection strategy* is used to detect errors so that they do not propagate to the output of the system. If an error is detected a correction procedure can be started.

3. *Fault-tolerance strategy* is used to add enough redundancy so that the system continues operating correctly. It either obscures faults and ignores their occurrence (*Masking* [32]), or reconfigures itself to circumvent faulty parts (*Dynamic* [33]).

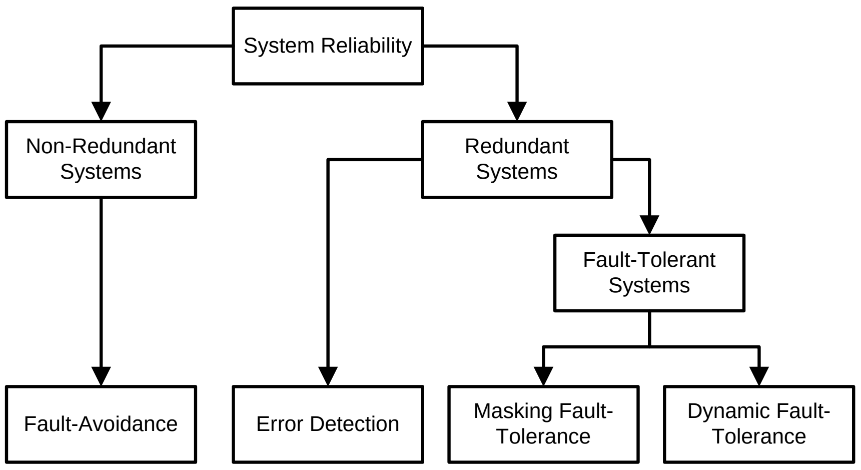


Figure 2.2: Taxonomy of System Reliability

2.4.1.1 Fault-Avoidance Strategy

Fault-avoidance techniques can be considered as the most expensive and the most time consuming ones, since custom design techniques have to be applied to avoid as much faults as possible. Different approaches can be taken in the implementation of these kind of techniques. They can be mainly categorized depending on production process (design, manufacture, and test) in which they are applied. *Radiation hardening* increases system reliability by adopting conservative design practices such as the use of high reliable components, the exclusion of radiation sensitive circuits and the provision of functional margins in circuit designs [34]. It is very difficult to find radiation-hardened parts that meet power, performance, and low-cost demands of modern systems. The design of a radiation-hardened IC should take into consideration the following points:

1. The selection of technology and manufacturing process that are not sensitive to the application environment.
2. The selection of design techniques that minimize radiation effects on IC response.
3. Computer simulations should be performed throughout the whole design process to see circuit response in the specific environment.

Shielding is another fault-avoidance technique, which reduces the environment severity [35]. Shielding works the best for alpha particles that have limited energy and range. However, it is not so effective in reducing the severity of energetic particles such as atmospheric neutrons, solar protons, and galactic cosmic ray heavy ions. This can increase some types of failures, as such particles can have more physical impact than before. When applied, shielding increases the overall system weight and volume.

Another possibility to avoid faults exists in *IC fabrication process.* The fabrication itself is a sequence of steps followed to create electronic circuits out of semiconducting material. Other than silicon, various compound semiconductors can be used for specific applications. Depending on the materials used and on the way how they are combined, different manufacturing processes provide different features concerning fault-avoidance in radiation environments. The most suited manufacturing process is Silicon-On-Insulator (SOI [36]), which is a semiconductor process that produces lower power, higher performing devices than traditional silicon processes. SOI works by placing a thin insulating layer, which could be silicon oxide or glass, between a thin layer of silicon and the silicon substrate. This process helps in reducing the amount of electrical charge that the transistor has to move during a switching operation, thus making it faster and allowing it to switch using less energy. SOI chips can be as much as 15 percent faster and use 20 percent less power than today's Complementary Metal Oxide Semiconductor (CMOS) based chips. SOI processes are generally tolerant to radiation [37] and have been developed for radiation hardened and space applications. The use of this technology is motivated by the full dielectric isolation of individual transistors that prevents latch-up effects.

2.4.1.2 Error-Detection Strategy

Error detection is crucial for meeting required system reliability, but it is a source of time overhead. In order to reduce this overhead, one possible approach is the hardware implementation of error detection mechanisms, which on the other side increases overall system costs. Error detection techniques that can be applied at hardware level[38, 39] are *Error Detection and Correction (EDAC) codes* for memories, *parity bits* for data buses, and *residue codes* for Arithmetic and Logic Units (ALUs). At system level, error detection techniques like *watchdog timers and processors* [40] and *N-version programming* [41] can be applied.

The general approach for system level error detection is to collect necessary information about the program at compilation time and to compare it with the information collected during runtime. Information that can be checked includes: program control flow, unit control signals, memory access behavior and results correctness. This can be achieved either by implementing self-checking programs with a separate checking task, or by integrating a watchdog processor, which minimizes performance overhead by passively monitoring bus transactions of the main processor. Errors are detected by monitoring the general system behavior. According to the location where the system-level error-detection techniques can be applied, they are classified as follows:

1. Control-flow error detection techniques check the sequence in which instructions are executed.
2. Address and execution time techniques check address information and time needed to execute a piece of code.

Pure software techniques are used in cases when hardware is fixed and cannot be changed. Some of the most used ones are:

1. *Block signature self-checking* technique checks the control flow between program blocks [42].
2. *Error capturing instructions* technique inserts trap instructions in the program, data and, free memory areas [42]. Such instructions should not be run during error-free execution.
3. *Executable assertions* are logic statements in the code that give the possibility to test some assumptions about the program [43].

4. *Procedure duplicated execution* applies temporal redundancy by executing a group of instructions more than once [44].

2.4.1.3 Fault-Tolerance Strategy

A fault-tolerant system can tolerate different types of faults like transient or permanent hardware faults, software and hardware design errors or induced radiation effects. Most fault-tolerance techniques can be categorized into hardware and software ones. A hardware fault-tolerant design is able to recover from random faults occurring in hardware components. This is achieved by partitioning the system into fault containment modules. Each module is replicated so that, in case of failure, others can take over. Some additional mechanisms are needed to detect errors and recover from faults. Hardware fault-recovery can be implemented in two general ways:

1. *Masking fault-tolerance* implements a structural redundancy technique by completely masking faults in a set of redundant modules [45]. Same functions are executed on a number of identical modules and their outputs are voted to detect and remove errors created by a faulty module. *Triple Modular Redundancy (TMR)* is a form of fault masking, in which modules are triplicated and voted [46]. To avoid individual voter failures, voter modules can also be triplicated. When two modules in a TMR system create errors, the vote is no longer valid and the TMR system fails. Such systems require at least three times as much hardware as non-redundant systems in order to continue operation even in presence of faults.

2. *Dynamic fault-tolerance* involves automated self-repair, but it is used only when one computation instance is running at a time [33]. Similarly to fault masking, system is partitioned into modules, which in this case are backed up by spares as protective redundancy. Special mechanisms are used to detect faulty modules and switch to a spare one, and to investigate software actions (rollback, initialization, retry, and restart) needed to restore state and continue computation. In single-processor systems, special hardware is required, while in multiprocessor systems the function is generally distributed among processors. Compared to voted systems, dynamic fault-tolerance is more hardware-efficient and is preferred in systems with limited resources.

Fault-tolerance can be implemented also on software by static and dynamic redundancy approaches, as those used for hardware fault-tolerance. One such approach that uses static redundancy, is *N-Version Programming* (NVP [41]). Redundancy is implemented by independent programs or versions of the same program that perform same functions. At special checkpoints, outputs are voted to find the faulty result. Once a consistent value is determined, all good versions can use this value for further processing. The dynamic approach might be based on *recovery blocks* that are parts of the same program. After the execution of each block, acceptance tests take place. In cases when acceptance tests fail, a redundant block is scheduled for execution.

Another approach to fault-tolerance with low cost is to tailor the scheme to the algorithm to be performed. This technique is very useful for high performance systems where high throughput is provided by the use of multiprocessors. The *Algorithm Based Fault-*

Tolerance (ABFT [47]) scheme needs only a few additional processing resources to tolerate failures. Incorporating ABFT comes with some costs. The fault-tolerance policy should be incorporated by programmers explicitly in the application code, making the code more complicated, hard to read and to maintain. *Software Implemented Fault-Tolerance* (SWIFT [48]) is another approach that efficiently manages redundancy by recycling unused instruction-level resources, which are available while most programs execute. SWIFT provides high-level protection and performance with an extended control-flow checking mechanism. As a compiler-based transformation, SWIFT duplicates instructions in the program and inserts comparison instructions at important points in the code generation. Values are computed twice and compared for equivalence during execution and before any differences due to transient faults can affect the output of the program.

There is another approach called *design diversity*, which combines software and hardware fault-tolerance. It implements a fault-tolerant computer system using different software and hardware redundant modules. Each module is supposed to provide same functions. A specific method is needed to find out if one module deviates unacceptably from others. In this way the system can tolerate software and hardware faults. Even though this is a very expensive technique, it is used in aircraft control applications and other critical systems.

2.5 History of Reliable Computers for Space

Space applications require processing platforms that operate for a long time without maintenance. The typical requirement is to provide a probability of 95 % that the processing platform will function correctly from five to ten years. Such platforms must use efficiently hardware resources since they are constrained to low power, size, and weight. It is true that National Aeronautics and Space Administration (NASA) sponsored from early times research in the field of fault-tolerance computing. The Orbiting Astronomical Observatory (OAO) was in 1960 the first fault-tolerant on-board computer that was developed and flown to space. OAO used the fault masking technique at component level, which at that time was the transistor [49].

The next fault-tolerant on-board computer was the Self-Testing-and-Repairing (STAR) computer, developed by Jet Propulsion Laboratory (JPL) in the late 1960 for a ten year mission [50]. This computer used dynamic recovery in design. Modules were implemented in such a way that they could detect faults and give this information to a repair processor that was responsible for reconfiguration and fault-recovery. Another successful space application that has operated in space for 20 years now, is the JPL Voyager [51]. Voyager's on-board computing system used dynamic redundancy by having redundant computer pairs checking each other by message exchanges. In the case when one of them failed, his partner could take over.

2.5.1 Reliable Real-Time Computers

Systems for aircraft and spacecraft control cannot tolerate an error or a delay. This is the reason designers invest a lot in implementing redundant hardware and software and in testing their implementations. The first machine of this type was Saturn V [52], which was developed in the 1960s. A TMR redundant processor was combined with duplicated memories that used error detection. Processor errors were masked by the voter, whereas memory errors were avoided by reading information from the other one. Another similar machine was the Space Shuttle computer [53]. It used four computers executing the same program and a voter for the results. When software errors were encountered, another computer was needed to recover from them.

During the 1970s, NASA developed two other fault-tolerant machines by applying hybrid redundancy. The first one was developed by SRI International and used software implemented fault-tolerance to achieve voting and reconfiguration of off-the-shelf computers [54]. The second one was the Fault-Tolerant Multiprocessor (FTMP) that used special hardware to achieve error and fault recovery [55]. Later, the FTMP evolved into the Fault-Tolerant Processor (FTP) and the Fault-Tolerant Parallel Processor (FTPP) [56]. FTP was a TMR system for process-control applications, whereas FTPP was a parallel processor system that ran processes in groups of two, three or four processors.

2.5.2 Microprocessors Used in Reliable Space Computers

In 1989, on-board of the Galileo[1] spacecraft [57], the RCA 1802 microprocessor was used for the command and data system and the Sandia Labs Rad-hard 2901 was used for attitude control computers. The 1802 microprocessors used in space were build using silicon-on-sapphire process, which was a more stable manufacturing process in the radiation environment. Galileo spacecraft had six such microprocessors, two for high-level modules and four for low-level ones. The 2901 microprocessors were configured as 16-bit processors (4 x 4-bit 2901s) and duplicated for redundancy for a total of eight.

In 1990, on board of Hubble Space Telescope [58] the DF-224 (8-bit) microprocessor was initially used. On the first service mission, an Intel 386 coprocessor was added. Pathfinder spacecraft had on-board in 1996 a BAE RAD6000 microprocessor [59], which is a radiation hardened IBM CPU produced by British Aerospace Electronics (BAE). Again in 1996, the on-board embedded computer of Sojourner rover was based on 100 KHz Intel 80C85 CPU with 512 Kilobytes of RAM and 176 Kilobytes of flash memory solid-state storage. The International Space Station (ISS) is a research facility in Low Earth Orbit (LEO) since 1998. It has several computers on-board. The most important ones are command computers that now use Intel i386 microprocessors. Since 2004, Spirit & Opportunity Rovers search for and characterize rocks and soils that are related to the past water activity on Mars. They are mainly based on the radiation-hardened RAD6000 single board computer, based on the IBM RISC design [59].

2.6 Novel Approaches to Reliable Computing for Space

The research in reliable computing for space applications is currently related to different topics and directions, such as radiation-hardened general-purpose processors, radiation-tolerant Field Programmable Gate Arrays (FPGAs), reconfigurable fault-tolerant systems,

[1]Galileo is a satellite navigation system built by ESA.

dependable multiprocessor designs and recent radiation-hardened by design many-core processors. In order to provide reliability, most of these directions are still oriented towards protection by-hard or by-design, but there is also an effort to include COTS products in the design of such computing platforms.

2.6.1 Radiation-Hardened General-Purpose Processors

The current generation of radiation-hardened general-purpose processors is the result of some work aiming to build radiation-hardened processors compatible with commercial products in terms of performance. This compatibility has to be achieved through adoptions in architecture and technology. One such processor is the RAD750 microprocessor [60], which is a compatible version with the commercial PowerPC 750 in terms of functions, architecture, instruction set and pinning layout. In the new RAD750 processor, new circuit designs replace pin-by-pin the original design component database. This is achieved by combining custom circuits with standard circuit design techniques. The main problem in the development of RAD750 was to harden the deep sub-micron technology. It was expected to have increased susceptibility to SEUs. Many new technologies had to be developed and implemented in the RAD750 microprocessor.

2.6.2 Radiation-Tolerant Processors

European Space Agency (ESA) in collaboration with Gaisler Research designed two radiation tolerant processors called ERC32 and LEON. ERC32 is a 32-bit RISC radiation-tolerant processor developed for space applications following SPARC V7 specifications. Two versions of ERC32 have been manufactured, ERC32 Chip Set [61] and ERC32 Single Chip [62]. ERC32 Single-Chip includes an Integer Unit (IU), a Floating Point Unit (FPU), a memory controller with an additional DMA arbiter. For real-time applications, it provides security watchdogs, timers, interrupt controllers, parallel and serial interfaces. Fault-tolerance is achieved by using parity on internal and external buses and by applying EDAC codes on external data buses.

LEON is a synthesizable VHDL core that can be implemented in programmable FPGAs or can be manufactured as ASIC. There is a series of incremental versions of the LEON Intellectual Property (IP) core [63]. LEON2 provides basic functions of a pipelined in-order processor. It has a five-stage pipeline, but does not support SMP. LEON2-FT is the SEU-tolerant version of LEON2. TMR is used in the internal design and all memories are protected by EDAC and parity bits. LEON3 is configurable and suitable for system-on-chip designs. The difference from LEON2 is that LEON3 has a seven-stage pipeline and supports SMP. LEON3-FT is the fault-tolerant version that provides SEU error-correction in the register file, error correction in cache memory, autonomous transparent error-handling and minimal timing impacts due to error detection and correction.

LEON2-FT has been proposed to be integrated in the next generation of Global Navigation Satellite System (GNSS) space receivers that are compatible with improved GNSS signals from Galileo and modernized GPS systems [64]. Advanced GPS/Galileo ASIC

(AGGA4) chip integrates LEON2-FT and a full FPU. Multi-DSP/micro-Processor Architecture (MDPA) is a system-on-chip design that again integrates a LEON2-FT with a micro FPU.

LEON3-FT was considered for the Solar Probe Plus program that required a low power microprocessor with resistance to Single Event Effects (SEE = SEU + SET) in a high proton flux environment [65]. The authors in [66] define SCOC3, which is an integrated system-on-chip for space applications. At the core of SCOC3, the LEON3-FT processor is combined with a FPU to improve performance of applications with floating-point operations. The latest LEON4 is an updated version of LEON3 with performance improvements as the result of wider internal buses, support for L2 cache and modified pipeline. Precisely, there is a 64-bit or 128-bit path to AMBA AHB interface and static branch prediction is added to the pipeline.

2.6.3 Reconfigurable Computing in Fault-Tolerant Systems

Reconfigurable computing in fault-tolerant systems takes advantage of reconfigurable features of FPGAs. Such features can provide dynamic and adaptive fault-tolerance. Static fault-tolerance limits system flexibility. On the other side, reconfigurable fault-tolerance provides the possibility to tune performance and reliability according to respective needs. This is interpreted as the key to achieve power savings or performance improvements without losing system reliability.

Since FPGAs reside at the center of such systems, they need to be further analyzed concerning their application in the harsh space environment. When a SEU happens in the FPGA logic fabric, no permanent damage is caused and the fault can be recovered by a simple reset [67]. This is generally called a soft-fault. But when the SEU happens in the configuration memory of an active part of the FPGA, a simple reset cannot restore the initial circuit state [68]. In this case a Single Event Functional Interrupt (SEFI) has happened. Even this failure does not damage permanently the FPGA, but traditional recovery techniques cannot be used because the physical circuitry in the fabric has been changed.

FPGAs that have a fuse-based one-time configuration memory can be used without problems in high-radiation environments. This protects them from SEFIs, but limits design flexibility. One-time programmable FPGAs cannot reconfigure in the field, which makes them unsuitable for reconfigurable computing. In SRAM-based FPGAs, a scrubber module is used to detect and correct SEFIs in configuration memory. The scrubber continuously compares data in configuration SRAM memory to original configuration data in a non-volatile off-chip device [69]. When an error is detected, the scrubber overwrites configuration memory with the original values. Scrubbing is not dependent on system hardware and it does not have to be integrated in the main system. The drawback of scrubbing is that scrubbers do not know in which location of the configuration memory errors have occurred. They scan the whole memory content, which leads to additional latencies.

Permanent damage is caused to SRAM-based FPGAs from the TID radiation effect. In order to deal with this problem a tile-based soft processor computing system is proposed in [70]. This approach divides the FPGA into tiles and in each tile the Xilinx picoBlaze [71] soft processor is implemented. Each soft processor is programmed using partial reconfiguration [72]. Three processors are connected in TMR and the rest of them are considered spares. The number of spare processors depends on available FPGA resources. When a fault is detected, the recovery process resets, reinitializes, and resynchronizes the damaged tile. This process reduces SEUs in FPGA fabric. If recovery is not successful, a spare processor substitutes the faulty one. When TMR is again in operation, partial reconfiguration is used to recover the faulty tile. If successful, this step reduces SEUs in FPGA configuration memory. When the recovered tile shows again problematic behavior, it might be damaged from TID and cannot be used anymore. Leaving this tile aside allows the system to continue operation in presence of TID faults.

A runtime reconfigurable processing technology called eXtreme Processing Platform (XPP) is proposed in [73]. This solution provides the flexibility of a general-purpose processor and the performance of an ASIC. XPP is composed of an array of arithmetic units, embedded memories, high bandwidth I/O, and a packet-oriented internal network. As a flexible high-performance architecture it can be easily integrated in a system-on-chip design. The distinct feature of this architecture is the fact that the instruction flow is replaced with a configuration flow and single basic operations are replaced with complex operations performed on whole data streams. This allows to calculate in one cycle a complete algorithm composed of many basic machine operations such as addition or multiplication. In such an architecture, multiple data-flow applications can run in parallel after mapping algorithms to processing elements of the scalable array.

2.6.4 Dependable Multiprocessor Designs for Space

In 2005, NASA's new millennium program office authorized the development of a new technology for payload and robotic missions in response to the need to apply COTS products in space. This new technology, called Dependable Multiprocessor (DM) [74], is a high-performance fault-tolerant cluster of COTS processors. The main feature of DM architecture is the employment of reconfigurable FPGA co-processors to achieve high-performance and efficiency. DM provides a parallel environment for scientific codes including an application development and a runtime environment that are familiar to scientific application developers. This reduces costs and time needed to port applications to DM platform. In order to provide the required reliability, DM includes algorithms for fault-tolerance that enable the system to dynamically manage resources depending on environment and application conditions.

DM can be considered as a reconfigurable cluster with central control. It contains a radiation-hardened system controller, a cluster of reconfigurable processors, redundant packet-switched networks, and a radiation-hardened data storage. The software architecture is composed of mission layer, middleware layer and platform layer. Fault-tolerance techniques are implemented in middleware layer. DM technology was developed as part of the Space Technology 8 (ST8) project. In 2006, the DM successfully passed some key ST8 project milestones. After passing the Technology Readiness Level 5 (TRL5) milestone, DM qualified for flight system development [75]. In 2007, NASA removed from ST8 project the TRL7 technology validation experiments, including DM. Since then, DM project has been looking for an alternative ride to space to achieve TRL7 validation for DM technology. The authors in [76] describe implementation details on DM technology for CubeSat applications. A CubeSat is a type of tiny satellite (10 cm cube with up to 1.33 kg mass) used for research with COTS products.

2.6.5 Multi-Core and Many-Core Processors for Space Applications

The fact that commercial processors integrate many cores on a single die can be seen as an opportunity for high performance space computing. Besides, the use of such processors in space can be also seen as a challenge, taking into consideration the radiation environment. The next generation of multi-core processors will increase the number of cores, building so many-core processors. Such processors are the 64-core TILE64 from Tilera [15] and the 80-core tera-scale processor from Intel [77]. By using similar designs for critical applications and on-board science computing, high performance can be obtained by reducing mass and volume.

Another novel and interesting architecture is the Plural Architecture [78], which is defined as a shared memory many-core system with hardware scheduling. It consists of many small processor cores, each containing a small private memory. A fast network on chip interconnects cores with shared memory, while another network on chip is used to connect them to the scheduler. A 64-core chip with 16 FPUs operating at 400MHz consumes only 1 Watt. The Plural many-core is an accelerator single-chip module that runs a single parallel program at a time. This makes it suitable only for solving a specific given problem, but also suitable for space applications, taking into consideration the low-power consumption.

NASA is considering the Maestro many-core processor [79] for reliable space applications. Maestro is a radiation-hardened by-design processor based on the TILE64 processor by Tilera with additional FPUs on each core. Redundancy and fault-recovery techniques have been applied for error detection and permanent recovery. Maestro integrates a grid of 7-by-7 general purpose processing cores (executing at 300 MHz), each composed of a multilevel cache hierarchy with an 8 kilobyte L1 cache and a 64 kilobyte L2 cache. The virtual L3 cache consists of the L2 cache of each tile. This means that it is distributed over all of 49 tiles of Maestro. A two-dimensional mesh network known as the iMesh [80] interconnects the cores within Maestro. iMesh is responsible for inter-core communication and also for routing data from main memory to individual tiles and I/O interfaces.

The authors in [81] propose new fault-tolerance strategies for the Maestro many-core space-enabled processor because it is still possible to have software and hardware failures, despite the fact that Maestro is radiation-hardened by-design. To minimize the number of such failures, common fault-tolerance strategies used in high performance computing and embedded systems have been applied to Maestro. The available cores and memory controllers provide enough resources for replication, checkpoint and rollback recovery and for application specific fault-tolerance on top of existing radiation-tolerance of Maestro. The purpose of such techniques is to allow the developer to detect and recover from faults within the processor and not to prevent them from occurring. In this way, the computation can continue even in presence of failures, if they do not propagate to the application-level.

Maestro's architecture is very flexible concerning programming models that can be implemented [82]. This flexibility is inherited from the TILE64 processor. Maestro can run an

SMP Linux Kernel that supports process and thread parallelism. It also supports the iLib [80] task-based parallel programming model from Tilera. Shared memory programming paradigms such as Pthreads and OpenMP can be used for parallelization in Linux. MPI and Tilera's iLib provide message passing abstractions for the distributed memory environment. Commercial Tilera programming models and tools had to be modified mainly to add support for the floating point module included in each of Maestro cores. Even though Maestro is a customized many-core processor based on the commercial TILE64 processor from Tilera, it cannot be considered as a pure COTS product since it was modified in hardware to provide a radiation-hardened by-design processor. There is not yet a final product because Maestro is still in development as part of NASA's OPERA program.

2.7 Current and Future Space Applications

There has been a lot of interest in building reliable computers for spacecrafts over the years, but in the latest ones researchers have started to work towards designing high-performance reliable platforms. Such platform do not only manage and control the correct functioning of the spacecraft, but can take over some processing tasks needed by the application. This requirement has been dictated by the trend in space applications, which require more and more resources for on-board processing. Current and future planned space applications have to be taken into consideration in order to provide a consistent proposal for the reliable high-performance architecture. In a long time, space applications and missions have been used for both commercial and military purposes. Navigation, communication, reconnaissance, and weather satellites provide advantages to military personnel. Besides, commercial space applications are very important to prosperity, economic success, and overall business climate in society. They enhance such things as: telecommunications, television broadcast, navigation, and computer network timing. Some of newborn space applications that pose high performance requirements are:

1. Cosmic Ray Elimination (CRE),
2. Hyper-spectral Imaging (HSI),
3. Synthetic Aperture Radar (SAR), and
4. High Resolution Wide Swath (HRWS) SAR.

Other applications related to aerospace are: Space-Time Adaptive Processing (STAP [83]), Ground Moving Target Indicator (GMTI [84]), Airborne Light Detection and Ranging (LIDAR [85]), Digital Down Conversion (DDC [86]) and Probability Distribution Function Estimation (PDFE [87]). STAP is a technique used to cut off clutter returns in airborne phased-array radar systems. It applies a multidimensional adaptive filter to the input data, which enables target separation from clutter by using relationships between angular location and clutter's Doppler frequency. GMTI radar is used to detect, track, classify and locate ground moving targets in all kinds of weather, day or night, and cluttered conditions. This makes it suitable for commercial and military applications including surveillance, airport traffic control, and threat assessment in war situations.

LIDAR technology, based on scanning lasers combined with GPS and inertial technologies creates fast and accurate terrain models for various applications. LIDAR can be considered an optical remote sensing technology that can measure target distance and other properties by illuminating it with light or even by using pulses from a laser. The purpose of DDC is to reduce analog components due to drift and noise. It converts a digital real signal centered at an intermediate frequency to a baseband complex signal centered at zero frequency. This is called down conversion and is usually followed by decimate to a lower sampling rate enabling further processing by slower processors. PDFE is used to quantify signal characteristics in many signal processing applications, such as image segmentation, restoration and texture synthesis. In a parametric estimation a known model is fitted into data and then the probability distribution function is expressed by parameters of that model. In a non-parametric estimation, the distribution function is estimated by measuring the original signal.

2.7.1 Cosmic Ray Elimination (CRE)

CRE application uses image processing methods to remove effects caused by cosmic rays. The main computation of CRE is the median filtering, which runs through image pixels replacing each of them with the median of neighboring entries. The median is calculated by sorting all signal values from the neighborhood into numerical order and then replacing the value of the signal being considered with the middle signal value. Image processing in CRE is easily parallelizable with minimal communication overhead. For the median filtering, there is already a fault-tolerant implementation available [88]. Other portions of the algorithm have to be replicated by hand in order to provide the required level of reliability.

The CRE Processing Framework [89] is an image analysis application that eliminates cosmic-rays in Charge Coupled Device (CCD) images using the embarrassingly parallel L.A.COSMIC algorithm. The framework is written in C and parallelized using MPI. For image partitioning, it uses a two-dimensional algorithm that partitions the input image into N rectangular sub-images with nearly the same area. Each sub-image includes sufficient additional pixels along the common image partition edges in order to eliminate the need for communication between different processes. This is the reason that it is considered an embarrassingly parallel image-analysis framework. L.A.COSMIC algorithm is based on a variation of the Laplacian edge detection. It uses edge sharpness to identify cosmic rays of arbitrary shapes and sizes and can reliably distinguish between poorly under-sampled source points and cosmic rays.

2.7.2 Hyper-Spectral Imaging (HSI)

Earth observation applications can be categorized according to the used sensor technology, which can be optical and radar. Hyper-spectral imaging that falls into optical category and SAR imaging that falls into radar category, represent the most demanding class of instruments in which large data flows put limitations on payload performance. HSI applications use traditional beam-forming methods to perform coarse-grained classification on hyper-spectral images [90]. HSI divides the spectrum into much more bands than the human eye can see. The information collected by sensors builds a set of images, each one representing a range of the electromagnetic spectrum. HSI processes and analyzes a three-dimensional data set, which is a combination of those images. The three-dimensional data set is reduced through the course of the application, most stages of which can be executed in parallel [91]. Hyper-spectral images can be beneficial in remote sensing applications such as monitoring growth status of vegetation, the atmosphere, and underwater ecological environment [92].

Each HSI sensor receives reflected light from below and transforms it to electrical signals. Objects on the ground are illuminated by sunlight. As an aircraft or a satellite flies over the ground surface, the reflected light is collected by HSI sensors. HSI technology uses the reflected light to detect objects or to distinguish between them. Reflected light is dispersed into its spectra and then intensities of certain spectral bands are measured. Measured intensity values are characteristic of reflecting objects. HSI applications use spectral signature matching and anomaly detection to detect objects. Spectral signature matching detects distinct objects or even substances. The processor loads spectral signatures for objects or substances. When a reflected light matches a spectral signature, then an object is highlighted on display. Anomaly detection discovers objects that do not belong in the area where they are located. Spectral intensities are combined while reflected light is being obtained. When detected spectral intensities diverge significantly from the calculated model, the object is classified as an anomaly and is highlighted on display.

One of the instruments that uses HSI is the infrared sounder. The primary objective of infrared sounder is to support numerical weather prediction at regional and global levels [93]. The on-board processing consists of spike detection and correction, non-linearity correction using Look Up Tables (LUTs), interferogram re-sampling, numerical filtering, decimation, compression, and CCSDS[2] data formatting. Non-linearity correction and re-sampling algorithms require intermediate memory for buffering purposes. LUTs used by the non-linearity correction algorithm need to be updated on-board during commissioning phase. During calibration and commissioning phases, specific algorithms within the on-board processing chain need to be bypassed.

2.7.3 Synthetic Aperture Radar (SAR)

Earth observations using SAR have a wide range of practical applications such as ship detection, ice monitoring, and enhancement of geological or geomorphological features. In the aftermath of a flood, the ability of SAR to penetrate clouds is extremely useful. Here, SAR data can help in optimizing response initiatives and damage assessments [94, 95]. In general, SAR applications are used to form high resolution images of the Earth's surface from platforms moving in space. Processing is based on patches that have significant overlap among each other. SAR applications can be parallelized on different levels of granularity and sometimes even without the need for inter-processor communication. The data set is two-dimensional and can range in size from some megabytes to gigabytes and is not reduced through the application processing stages. As suggested in [91], SAR applications are very suitable for ABFT techniques. They were considered as a candidate application for the Dependable Multiprocessor [74]. Since the SAR family of applications is chosen to benchmark different HPC systems, it will be further analyzed in this section.

[2]Consultative Committee on Space Data System develops recommendations for data- and information-systems standards to promote interoperability and cross support among space agencies.

The selection is made because SAR applications pose significant challenges considering very large computation and data storage requirements.

SAR is a side-looking radar system [96] that uses the flight path to build a large antenna aperture (Figure 2.3). That enables SAR to generate high resolution images. Data is stored when each transmit/receive cycle is finished. Pulse Repetition Frequency (PRF) represents the number of pulses transmitted per second. SAR processing uses magnitude and phase of the received signals from different elements of the aperture. After some cycles, data is recombined to build the high resolution image of the terrain. This combination is needed because SAR uses one antenna in time-multiplex mode, instead of using many parallel antennas. A side-looking radar travels in the flight direction while transmitting the microwave beam at right angles to the flight direction. The cross-track dimension perpendicular to the flight direction is referred as *range*, while the along-track dimension parallel to the flight direction is referred as *azimuth*. The strip of the scanned Earth surface is called *swath* and the width in range dimension is called *swath width*.

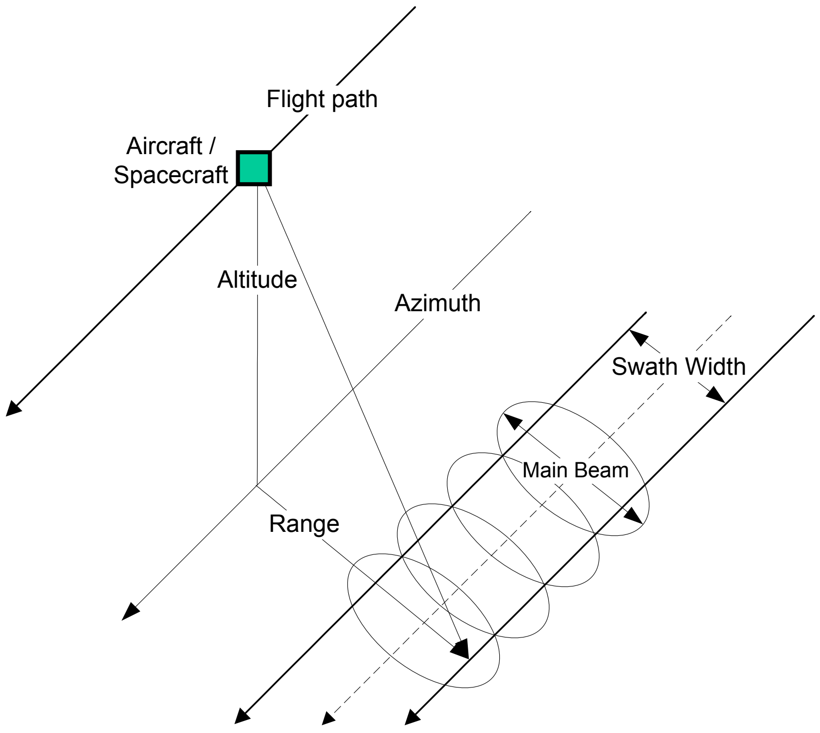


Figure 2.3: Side-Looking Radar Systems

As shown in Figure 2.4, there are three different operation modes for SAR [97]:

1. In *stripmap* mode a fixed pointing direction is assumed. Image width is equal to SAR swath and length is equal to the distance flown by the platform.
2. When SAR operates in *spotlight* mode it provides high resolution while keeping targets within the beam for as long as possible, and thus forming a longer synthetic aperture. Azimuth resolution increases with the number of transmitted pulses. This is the reason that each target is kept in spotlight illumination of the radar for a longer time.

3. In *scan* mode, SAR is able to illuminate several sub-swaths by moving its antenna into different positions.

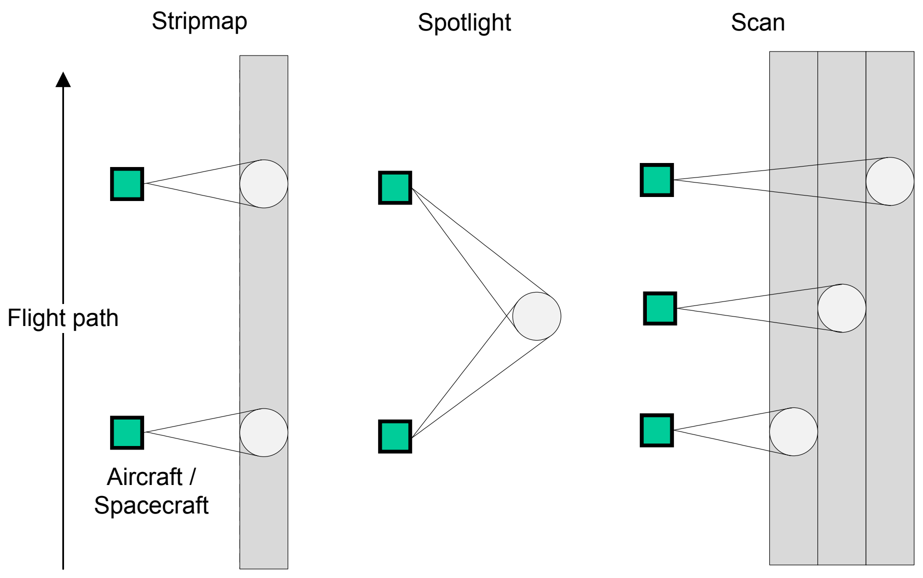


Figure 2.4: SAR Operation Modes

2.7.4 High Resolution Wide Swath (HRWS) SAR

Current state of the art SAR systems typically use antennas with analog networks and one final phase center. In such systems, two top-level system parameters, *swath-width* and the *azimuth resolution,* are related and cannot be improved at the same time. The novel HRWS concept is expected to overcome these constraints by combining a high azimuth resolution with an improved swath-width and a continuous coverage [98]. Further details on HRWS SAR technology and baseline instruments can be found in [99]. The goal of HRWS SAR is to help saving communication bandwidth by compressing generated images. Its processing steps and corresponding computational requirements are analyzed in this section, as it is considered a candidate application for future space missions. HRWS SAR challenges are referred as requirements that have to be fulfilled by the proposed reliable high performance architecture.

The HRWS SAR system architecture requires two separate apertures for transmit and receive. The receive aperture is split into multiple sub-apertures in azimuth and range called panels and tiles. In this way, signals from panels are not combined as a single signal as in the case of a conventional phased-array SAR application. The radar echoed signal is sampled at different panels simultaneously, obtaining higher PRF than usual. The receive antenna of a reference instrument consists of 7 panels with 12 tiles each. The signal of each panel is processed separately by the SAR processor because each panel represents an independent phase center. The radio frequency signal received by each tile is down-converted and then digitized at a sampling rate of 1 GHz with a resolution of 8 bits. Each tile therefore yields an instantaneous raw output data rate $\rho = 1\,GHz \times 8\,bits = 8\,Gbit/s$. PRF depends on the imaged swath. The maximum PRF is 1795 Hz. The largest echo window size occurring is $T = 500\,\mu s$, so the maximum average raw output data rate per tile is $T \times PRF \times \rho = 7.18\,Gbit/s$. For the whole aperture with 84 tiles, an average data rate of 603.1 Gbit/s is expected.

Figure 2.5 depicts the HRWS SAR processing steps at the tile and panel level, whereas Table 2.1 lists processing power requirements per tile, per panel, and at complete instrument level. Each panel is subjected to an identical digital processing sequence. Each tile is the source of an analog signal. After ADC sampling, data from each tile is identically subjected to scan-on-receive channel processing. Processed data from each tile is then summed into a single data stream. This data stream is finally subjected to decimation and compression before it is sent to the on-board memory for storage and subsequent down-linking.

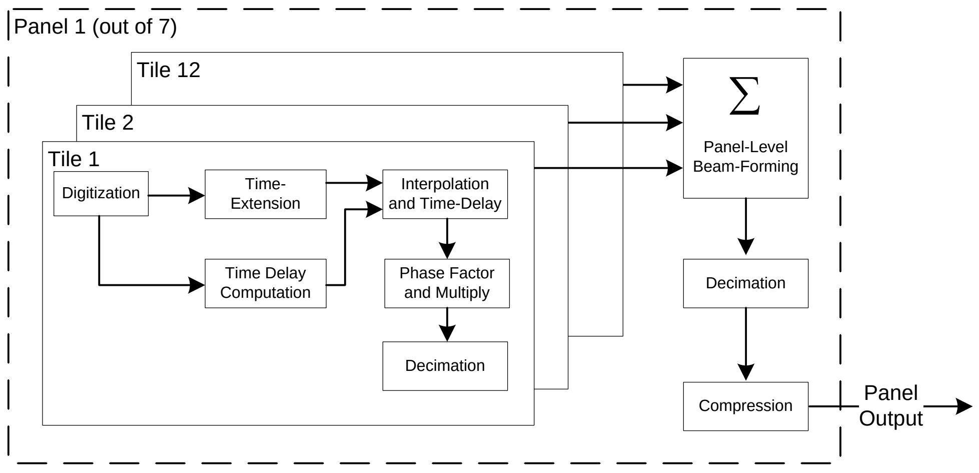


Figure 2.5: HRWS SAR Processing Steps

Table 2.1: HRWS SAR Processing Power Requirements

Processing Level	**Processing Power Requirements**			
	Number of Complex Operations	Input Word Length (bits)	Data Rate (Giga Bits/ Second)	Number of 16 bit fixed point operations per second
Tile	25	16	7.18	8.7×10^9
Panel	106	24	86.16	145×10^9
Instrument	742	24	603.1	1.02×10^{12} (~1 Tera)

3 The Proposed Reliable High Performance Architecture

This chapter proposes a high performance computing architecture for future reliable space applications. All design goals discussed in section 1.2 on page 2 are taken into consideration for the proposal. The distinguished feature of the architecture critical to achieving these goals will be the combination of radiation-hardened and fault-tolerant components with high performance multi- and many-core computing technologies, which have lately evolved to overcome computational and memory constraints. Accelerator technologies based on GPUs are also considered in the proposal. The purpose of integrating such components in the architecture is to provide better performance and bandwidth capacity for specific space applications that can benefit from their architectural features.

3.1 Achieving Design Goals

In order to build a high performance computing platform, multi-core and many-core computing technologies have to be introduced in the design process. Symmetric and distributed shared memory processor architectures combine a wide range of features to deliver, in a cost effective way, high processing power and reduced bandwidth demands on memories [100]. The main feature is the shared memory communication model, which simplifies programming of complex and dynamic communication patterns. Other benefits include lower communication overhead and reduced remote communication by data caching. The presence of large multilevel coherent caches for both shared and private data, results in reduced access latency and memory bandwidth.

To further relieve the pressure on the memory, some of the processors have integrated memory controllers on chip, which can be coupled with large high performance caches to lower overall latency. Having multiple synchronized cores gives the possibility to exploit thread-level parallelism, where each core fetches its own instructions and operates on its own data. Cache use is even more effective when multiple threads run on the same core and use the same data. Except from thread-level parallelism and multithreading, the upcoming multi-core processors will also be capable of vector and graphics processing. Since many similar tasks can be processed in a fixed amount of time, multi-core processors provide higher performance per power consumption and per unit of area.

In the recent years, GPUs have increased their peak performance and bandwidth capacity faster than CPUs. Modern GPUs are massively parallel computing devices that support thousands of concurrent threads. They are special purpose processors that for specific applications perform certain computations many times faster than CPUs, and what is more important, at the same power consumption. Not only there is more potential performance and power efficiency in a GPU, but they have also an additional advantage. GPUs have always been designed as data-throughput processors and provide a favorable Floating-Point Operations per Second (FLOPS) to bandwidth ratio. The authors in [101] state that for workloads with abundant parallelism, GPUs deliver higher peak computational throughput than latency-oriented CPUs. CPU memory tends to be more bandwidth constrained, compared to GPU memory. This makes it more difficult to extract theoretical FLOPS from the CPU. This is one of the main reasons that performance of data-intensive applications almost never scales linearly on multi-core CPUs.

Flexibility can be assured from the fact that COTS components integrated in the computing hardware architecture can be programmed using available application development and runtime environments. In shared memory processors, transparent sharing is supported by coherent cache memory hierarchies and data can be easily protected by software locks, semaphores, and critical sections. The most familiar thread programming tools are POSIX-threads API [6] and OpenMP pragmas [7]. SIMD vectorization [102] is usually carried out by compiler technologies that support efficient data gather, scatter, and conversion utilities. Further improvements can be obtained by using intrinsics to vectorize the code manually, but this might ask for additional development efforts. When it comes to programming GPUs, different approaches can be followed depending on the manufacturer. CUDA [20] and OpenCL [21] are the most well known programming models for GPUs. OpenCL has recently become suitable for both CPU and GPU programming.

In general, the application development environment should provide:

1. Techniques for communication between the fault-tolerant host module and the computing modules,
2. Paradigms for parallelization at different application levels,
3. Software scheduling algorithms for dynamic load-balancing purposes, and
4. Irregular data structure support.

Other requirements are related to high throughput I/O and high speed interconnection/interfacing techniques. To deal with high throughput I/O requirements, different approaches might be taken. One such approach relies upon the use of high throughput system bus interfaces. Another approach might be to integrate a COTS packet switched fabric to interconnect modules in the proposed computing architecture. Best candidates for interconnection and interfacing standards like PCIe [103], Serial RapidIO [104], and Ethernet [11] are discussed later for the proposed architecture. A specific interface can be used to isolate the computing subsystem from other spacecraft subsystems making it easily portable to different missions and application. This will also reduce costs and time associated with porting of applications to the spacecraft computing architecture.

When building reliable systems, spacecraft system designers do not trust new technologies. However, it is not always feasible to build systems based on customized hardware considering that COTS products are cheaper and more flexible in use. In most previous architectures, designers trusted only customized components, like radiation-hardened processors (RAD6000 and RAD750 [60, 59]) and radiation-tolerant processors (LEON-3FT [65]). This method turns out to be not so feasible when designing spacecraft systems, as the process of designing and manufacturing such components becomes too expensive for such a small market. To provide a reliable architecture, one might take into consideration the adoption of techniques in fabrication or design process. Even though fabrication-process based techniques have high costs for such small markets, some control and management components of the system architecture have to be radiation-hardened to reduce the likelihood of experiencing major system faults.

On the other hand, design based techniques can be easily implemented on a higher level and what is more important they are less expensive. In this case, the architecture has to be supplemented with special hardware and software design techniques, which improve system reliability and use resources efficiently to obtain required performance. Most preferred means would be redundancy [105] and software implemented fault-tolerance techniques [48].

The primary indicator for the provided hardware scalability of the architecture will be the number of computing modules interconnected in the system. However, scalability also depends on the approach taken to interconnect them in the system because one has to be careful to avoid bandwidth bottleneck scenarios. System components have to be separable in order to be recombined to accommodate upgrades for future COTS parts. Aside from the number of advantages mentioned above, there are three main problems that have to be addressed when using COTS components in space:

1. Radiation effects like TID, SEL, SET and especially SEU/MBUs in the COTS products.
2. Thermal issues associated with the nature of the COTS products.
3. High power efficiency in terms of performance per power consumption.

3.2 The Reliable Hardware Architecture

As shown in Figure 3.1, the proposed reliable high performance computing architecture is composed of *N* Parallel Processing Nodes (PPNs) and one Radiation Hardened Management Unit (RHMU). The dataflow is from instruments to PPNs via I/O and digitizer components and from there through the backplane/interconnection and finally to ground via the satellite communication subsystem. The number of PPNs depends on computing needs and hardware scalability. In space, it is preferred to have small processing components distributed all over the area, so that radiation effects do not impact many of them at the same time and heat can be dissipated much more efficiently. However, network communications and application scalability is impacted by having many small distributed PPNs.

The distribution of PPNs implies that memory is physically distributed, but this does not mean that this architecture can be used to implement only distributed memory systems. For applications with low and intermediate computing requirements it is more flexible and efficient to implement a distributed shared memory system with a few PPNs. In such systems memory bandwidth scales if most accesses occur on local memory. Compared to symmetric shared memory approaches, this provides low latency scattered access to memory. Multiple physically distributed memories (one in each PPN) can be addressed as

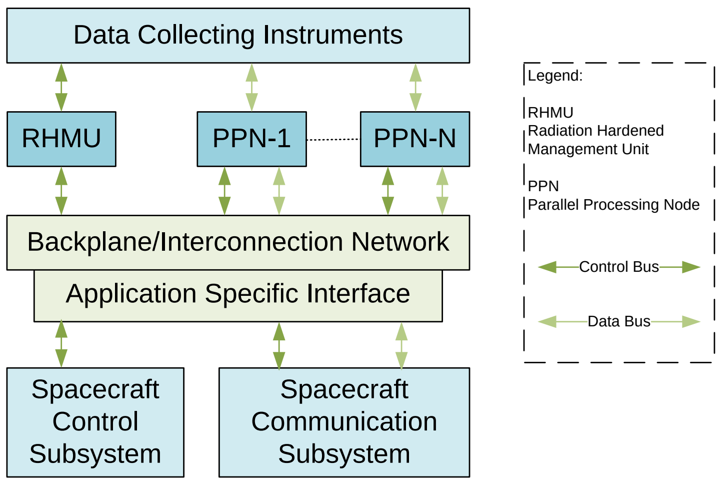

Data Collecting Instruments
RHMU
PPN-1
PPN-N
Backplane/Interconnection Network
Application Specific Interface
Spacecraft Control Subsystem
Spacecraft Communication Subsystem
Legend:
RHMU
Radiation Hardened Management Unit
PPN
Parallel Processing Node
Control Bus
Data Bus

Figure 3.1: The Reliable High Performance Computing Architecture

one logically shared address space, which in this case will serve also as a communication environment for the distributed multi-core processors in all PPNs.

For applications with very high computing requirements, the distributed shared memory system should be extended with accelerator modules on each PPN. This builds a heterogeneous CPU/GPU system. Such a heterogeneous system integrates two memory hierarchies that communicate with each other through the PCIe interconnect. Additionally to the main shared memory, there is also a high speed GPU memory. Communications between these two memory hierarchies have to be specified explicitly by the programmer. GPU-based accelerator modules provide efficient parallel performance for applications that can exploit fine grain parallelism.

For other applications, it might be better to consider implementing a distributed memory system with many CPU-based PPNs. Physically and logically distributed memory systems can improve application performance only if optimal communication patterns are used to exchange data between processes with separate address spaces executing on each PPN. Contrary to shared memory systems where this is done from communication protocols, in distribute memory systems the programmer has to explicitly specify needed communications.

Figure 3.2 illustrates the CPU-based PPN composed of M I/O components, two Multi/ Many-Core Processors (MCPs), one memory module associated with each MCP, the local bus, and the bus interfacing component. The number of MCPs in each PPN depends on computing needs, size of PPN, and generated heat. Increasing the number of PPNs might be a better option than increasing the number of MCPs in each PPN when considering the generated heat.

Figure 3.3 illustrates the heterogeneous PPN composed of M I/O components, one MCP and its memory, one GPU card with integrated graphics memory, one local bus, and one bus interfacing component. The MCP and its memory is needed to manage execution contexts on the GPU card as well as for processing algorithms that are not suitable for GPU computing. There can be more than one MCP and more than one GPU card on one

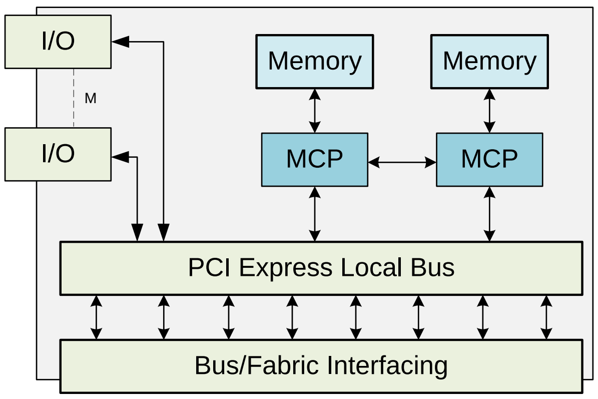


Figure 3.2: The Parallel Processing Node

PPN, but power consumption and generated heat will increase a lot.

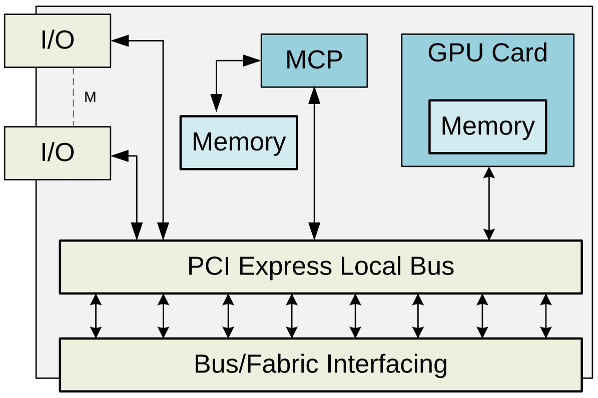


Figure 3.3: The Heterogeneous Parallel Processing Node

Each PPN has to be integrated in a separate card to provide modularity and scalability. This will also help in reducing thermal issues by avoiding concentrated heat. I/O components have to be used only to input and output application data and not for control data. A local PCIe bus has to be used especially for the communication between I/O components, MCPs, and the GPU Card. A separate module has to be used to interface the local PCIe bus to a backplane bus or to a packet switched fabric. Memory components with high bandwidth and sufficient capacity are needed to store buffered temporal data.

An Application Specific Interface (ASI) is needed to connect the computing subsystem with other subsystems of the spacecraft. By making the ASI an independent component, one can reduce the impact of porting the computing architecture to new communication systems and also to complete new missions. If needed, a low bandwidth and low latency control bus can be used to connect the RHMU with PPNs and with other satellite subsystems.

To protect the architecture, especially from upsets, it is proposed to apply a combination of hardware and software fault-tolerance techniques. Some components of the architecture, such as the RHMU and the ASI have to be radiation-hardened, to make sure that critical parts stay reliable. This approach will minimize the likelihood that data will be corrupted by faults, ensuring reliable input and output of data from the computing subsystem. The internal construct of the RHMU is similar to the PPN but some of the elements composing this unit have to be customized. As shown in Figure 3.4, at the center of the RHMU stays a Radiation Hardened Processing Unit (RHPU) that communicates with main memory through a voter. Memory in the RHMU should be triplicated for TMR. The voter is responsible for data integrity in memory as it selects the correct output by voting on three inputs coming from memory modules.

To address latchup effects, a latchup protection technology is proposed in [106]. This technology is designed to protect and recover susceptible ICs by providing limited current to the device. A shutdown of the device is applied when the current threshold is exceeded. The device is put on hold for some time before it is returned to its original operating level. Various manufacturing processes (like SOI) are susceptible to SELs, producing ICs suitable for such applications.

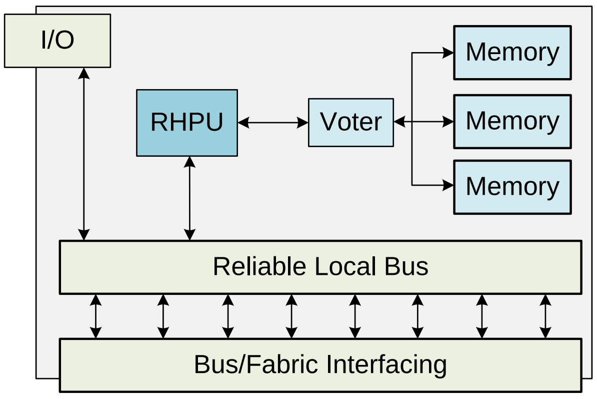


Figure 3.4: The Radiation Hardened Management Unit

3.3 The Reliable Software Architecture

Software techniques provide more flexible alternatives with lower-costs. The best way to improve reliability through software techniques is to integrate a fault-tolerant middleware between operating system and application. This middleware is supposed to coordinate with fault-tolerance functions used by the application programmer. An overview of the software architecture stack, its partitioning and mapping onto hardware is shown in Figure 3.5.

This solution will mainly provide SEE (SEU+SET) tolerance through software implementations. Hosted on the RHMU, the Fault-Tolerance Manager (FTM) is responsible not only for management activities related to fault recovery but also for other management tasks related to mission and application tasks and services. On the other hand, the Fault-Tolerance Layer (FTL) hosted on every PPN, is responsible for fault detection and report, but it also includes local management services. Both of these middleware layers should be isolated by APIs. This will make FTM and FTL services available for future missions

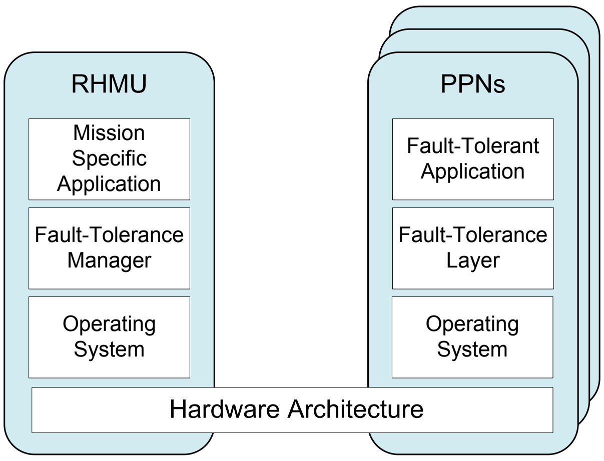


Figure 3.5: Software Architecture Stack

and applications improving so system portability. In coordination with each other, these layers should provide the following critical functions:

1. *Health monitoring.* FTM is supposed to monitor the health state of all system resources like applications, operating system, I/O interfaces, processing nodes, networks and other peripherals. A heartbeat technique can be implemented to monitor system health. At the hardware level of monitoring a watchdog timer [40] is a good solution for monitoring PPNs.

2. *Fault detection and report.* Any change in the health state that is detected by any respective resource's FTL should be reported to FTM. Algorithm based fault-tolerance [47] and check-pointing techniques [107] can be used to detect and in some cases correct data faults. Libraries of functions enhanced with these techniques should be used by application developers as fault detection mechanisms.

3. *Fault diagnosis.* FTM has to search for the origin of the problem starting with the FTL from where the report for a fault in the system came.

4. *Fault recovery* should eliminate the effect of the fault. Some recovery actions have to be taken based on runtime conditions and system history. These actions should depend on the type of the resource being affected by the error.

5. *System reconfiguration* should be possible to replace or isolate the faulty component if needed. FTM should be responsible for application scheduling, resource allocation, process dispatching and application recovery based on recovery policies. In a multi-core processor parallel environment some configuration options might be:

    a) To use other cores inside the same processor for the workload coming from a faulty process.

    b) Non-faulty processes continue execution by dividing the rest of the workload among them.

    c) Recover faulty components in order to have the system as before.

Replication and comparison is a well known method to detect and correct errors in a system. If two or more results agree in TMR, that result is taken as correct, otherwise an uncorrectable fault has been observed and additional action is needed. TMR technique provides an easier way to detect errors, but it makes no efficient use of resources. This technique was traditionally used at hardware level by replicating hardware resources, but this proved to have high costs. Since TMR is recommended only for critical system components, it is proposed to make use of it in the RHMU and in the ASI. Replicated results are submitted to a voter which determines the correct one. Voters tend to be single points of failure for most fault-tolerance techniques, so they should be designed and developed to be highly reliable, effective, and efficient. This is the reason it recommended to implement redundant voters in the radiation hardened modules of the system.

Process-level replication is a software technique, in which multiple identical processes can be instantiated on same or different computing resources and their results have to be compared for consistency. Some other replication techniques are diversity oriented, i.e. hardware and software elements are not copied reproducing redundant errors, but are independently designed to fulfill the same function through implementations based on different technologies. Time redundancy via diversity can be used to detect and recover from transient errors and spatial redundancy via diversity can be used to detect and recover from permanent errors.

One proposed diversity technique is the “N-Version Software”, in which each module is made up of N-different implementations that accomplish the same task in a different way. Multi-core processors provide multithreading capabilities, allowing these versions of the program to execute at the same time. Replication techniques can be implemented at different levels beginning from hardware components and ending at complete programs. It is recommended to use process level replication or N-version programming because multi-core processor based PPNs provide enough resources to implement them without losing too much performance. In the proposed architecture, redundancy can be implemented by threading into the shared memory multi-core processor at the PPN-level or by creating replicated processes at the distributed memory system level.

Some redundant threading alternatives for the PPN-level are evaluated in [108]. Chip-level Redundant Threading with Recovery (CRTR) combines the fault coverage of lock-stepping with the efficiency of the Simultaneously and Redundantly Threaded (SRT) design. Another mechanism to further improve system reliability would be to distribute asynchronous redundant threads or processes at the system level [109]. This approach will present communication and synchronization overhead since data and threads will be distributed dynamically over PPNs. In general, one needs to triplicate for recovery, but the above mentioned approaches are supplemented with techniques that enable recoverability even in double checking designs.

Another solution to the tradeoff between performance and reliability might be to implement a rotated consistency checking, in which only some threads get replicated and results checked for consistency at a time, but over a longer period all of them get verified. Figure 3.6 illustrates a combination of rotated redundancy and SRT technique. For an initial single threaded process, respective pointers to input and output data are given to the thread. At different moments in time, double threading can be applied in order to provide reliability or performance.

In Figure 3.6 the single threaded process is first double-threaded for reliability and then, at a later moment it is double-threaded for performance. This process continues in cycles that can be different from each other, meaning that in the next cycle double threading for performance takes place earlier than double-threading for reliability. The application programmer is the one that has to choose how often to apply each of the above mentioned threading, and which level of replication to choose if double-threading is not enough. In a distributed environment, each process can be subject of such a multithreading scheme. In order to balance system reliability and performance, the programmer has to make sure that each process gets checked only once in a cycle, while other processes are executing in parallel for better performance.

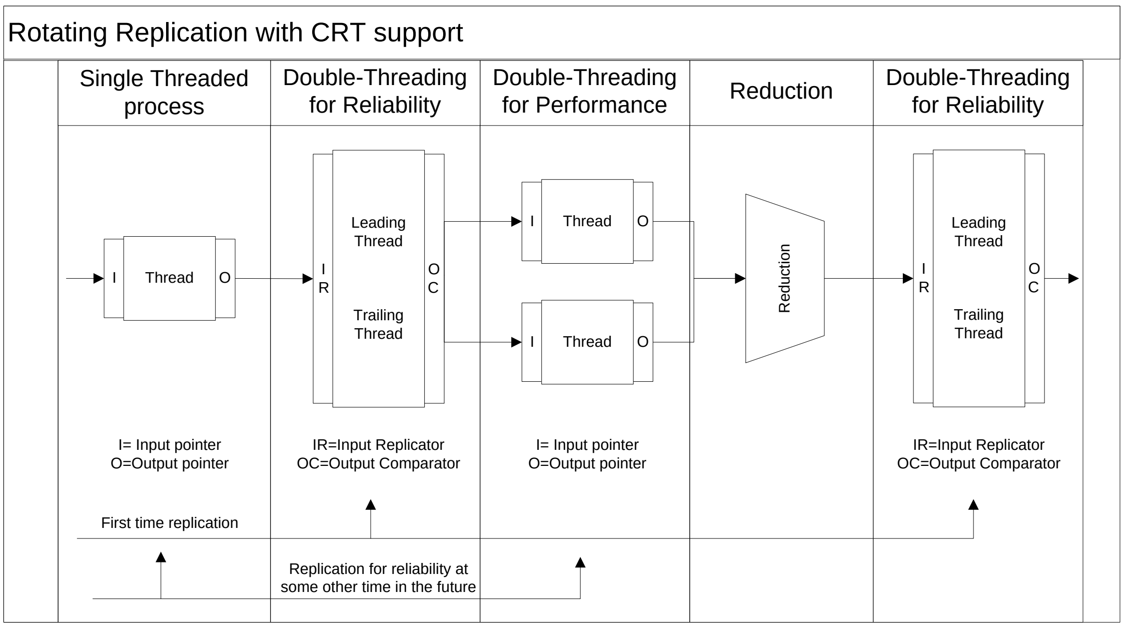


Figure 3.6: Rotating Redundant Threading

3.4 Hardware Technologies for the Proposed Computing Architecture

This section references computational and bandwidth requirements of HRWS SAR application to estimate the suitability of various hardware technologies for the proposed architecture. As already described in subsection 2.7.4 on page 34, HRWS SAR asks for 603.1 Gbit/s data rate and 1 Tera 16-bit complex multiply-and-accumulate operations per second. Taking into consideration the fact that each panel in HRWS processing is subjected to an identical digital processing sequence, it is recommended to partition the system at the panel level. It would be feasible to think that a separate PPN is needed in the computing architecture for each panel in the HRWS aperture. In the case of the reference instrument that consists of 7 panels, each of 7 independent PPNs will have to deal with an average raw data rate of 86.2 Gbit/s and a minimum processing power of 145

Giga 16-bit complex operations per second.

3.4.1 PPN Interconnection and Interfaces

Interfaces needed inside each PPN are described in this subsection. In addition to the description, some hints are given on the availability of such interfaces on the COTS market. The central interconnection medium is a PCIe switch interfacing multi-core processors with I/O cards and with the rest of the system via the backplane (Figure 3.7). The PCIe switch might be integrated in a chipset. Both microprocessor designers and vendors, Intel and AMD provide high performance chipsets to interconnect their processors with the rest of the system. ExpressLane PCIe Switch Family from PLX Technology [110] includes high performance, low latency and low power, multipurpose, highly flexible and highly configurable devices. Depending on the existing backplane or on the chosen interconnect, the PCIe switch can be interfaced to:

1. An Ethernet switch via an Ethernet controller [111]. Solutions for embedded designs using Gigabit Ethernet are provided by Altera [112], Vitesse [113] and Micrel [114] suppliers.
2. A RapidIO switch via a PCIe-to-Serial RapidIO Bridge. In [115] a PCIe-to-RapidIO bridge board is illustrated. It uses Infiniband cables to get to the RapidIO. Altera provides also solutions for RapidIO interfaces and RapidIO IP Cores. Another supplier is the Integrated Device Technology Company [116] that offers RapidIO switches for different applications.
3. A CompactPCI Express (CPCIe [117]) backplane via the PCIe switch. CPCIe is an industrial backplane standard exclusively based on PCIe.

The latest PCIe 3.0 standard [103] specifies a bandwidth of 32 GB/s (256 Gbit/s). It provides a serial point-to-point connection with dedicated bandwidth, enough to accommodate the transfer rate of two HRWS panels. Reliability in PCIe has been enhanced by error detection and signal integrity. PCIe supports Error Correcting Codes (ECC) for both Data Link Layer and Transaction Layer errors. Error reporting has been expanded as well. Signal integrity is improved by differential pairs used for signal lines with greater noise immunity than a fast speed parallel bus. The 8b/10b decoding scheme embeds the clock in the data signal which minimizes timing issues.

PCIe has fewer trace lines than prior PCI buses and therefore has fewer possible points of failure. By implementing fewer, yet more robust signal lines, PCIe is a reliable high speed bus. It supports a feature called *non-transparent bridging* that can be used to assist failover, as it provides a mechanism to isolate primary and standby host processors. Another way to improve reliability might be to build a fully redundant system by implementing a dual-star topology. Other worth mentioning PCIe features include:

1. End-to-end Cyclic Redundancy Codes (CRC) used to detect system-wide errors,
2. Credit-based flow-control for reliable transactions, and
3. Differential services by mapping virtual channels to different traffic classes.

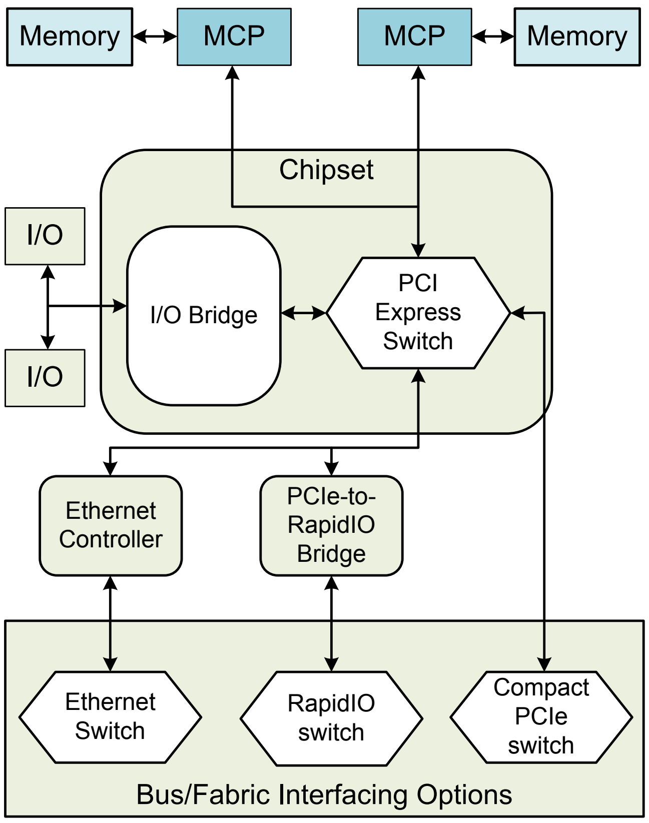


Figure 3.7: Interfacing inside the PPN

3.4.2 The Radiation Hardened Management Module

As already discussed, at the center of the RHMU stays RHPU that can be a radiation hardened processor like the RAD750 [60], a radiation-tolerant processor like the LEON-3FT processor [65], or a radiation hardened FPGA. To provide reliability for the data stored in the memory, the RHMU will integrate memory modules in TMR. Compared to the PPN, the local PCIe interface is replaced by a reliable SpaceWire interface (Figure 3.8). SpaceWire is a real-time communication network used on-board of spacecrafts. The purpose of this interface is to connect different on-board sub-systems of a spacecraft into a high-performance network. Several radiation-tolerant chips have been developed to support the use of SpaceWire. SpaceWire provides serial high performance links with up to 200 Mbit/s bandwidth. They are full-duplex point-to-point data connections between SpaceWire equipments.

3.4.3 System Interconnection Technologies

In subsection 3.4.1 on the preceding page, three possible interconnect technologies have been proposed for the system backplane. This backplane interconnects all individual subsystems of the spacecraft electronic system. CPCIe [117] supports up to 16 bidirectional high-speed communication lanes, with a total of 32 Gbit/s each way, which is even faster

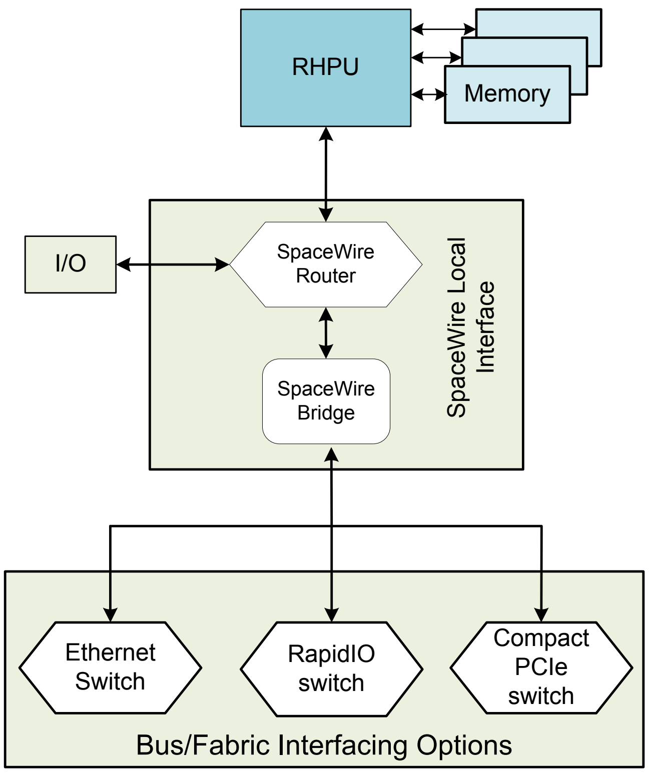


Figure 3.8: Interconnection in RHMU

than a 10 Gbit/s Ethernet link. CPCIe supports point-to-point connections over the lanes, providing direct connections within the system. There are already COTS products available to interface PCIe with CPCIe, but since CPCIe does not provide the required bandwidth, it is not a good solution for the proposed architecture.

Another technology that can be easily interfaced with PCIe is Advanced Switching. It is a packet-based transaction layer protocol that operates over PCIe physical and data link layers [118]. It provides all properties needed to build redundant systems in a variety of topologies like dual-star and meshes. Advanced Switching provides enhanced features such as sophisticated packet routing, congestion management, multicast traffic support, fabric redundancy, and fail-over mechanisms to support high performance, highly utilized and high-availability system environments. Its protocol encapsulation provides an efficient way of tunneling mixed protocols within the universal switch architecture with minimal processing overhead. This standard provides many useful features but it is actually difficult to find available products on the COTS market. As another packet-switched fabric, the 100 Gbit/s Ethernet IEEE 802.3ba Standard will be able to deal with the bandwidth required for data exchange with and within the architecture. This standard is more oriented toward LAN/WAN networks and it is not suitable for a fabric interconnect.

The packet-switched fabric might also be based on the RapidIO commercial industry standard [104]. RapidIO implements a packet-switched point-to-point interconnect allowing multiple full-bandwidth links to be simultaneously established between nodes in the network. The non-blocking nature of RapidIO allows concurrent routing of multiple packets.

By using multiple switches in the system, topologies consisting of hundreds or thousands of nodes can be designed. RapidIO interfaces are based on Low Voltage Differential Signaling (LVDS) technology and can achieve bandwidths of up to 60 Gbit/s for each active link. A 16-bit RapidIO system with two active point-to-point links is capable of 120 Gbit/s.

A notable benefit of the RapidIO protocol is its extensive error detection and recovery mechanism. By combining retry protocols with CRC and single/multiple error detection, RapidIO handles all network errors without application intervention. This inherent error handling and recovery capability proves ideal for space applications that require a highly reliable interconnection. Latest specifications of RapidIO technology state that it supports message passing and globally shared distributed memory programming model.

Taking into consideration the above mentioned features of RapidIO, it is recommended to use it as a system interconnect. Figure 3.9 summarizes proposals concerning interconnect technologies at system level and inside PPNs. It also includes information on how data is transmitted along different interconnect standards. This information shows that at the respective network layer, a bridge is needed to convert packets from one standard to the other. PCIe switches and the SpaceWire router are responsible for packet routing from and to different modules in the system.

3.4.4 Processor Technologies

Multi/many-core processor technologies are considered as candidates for the proposed architecture. Such technologies have to deal with computing requirements in the range of Teraflops. The selection of the processor should be based mainly on costs, performance (processing power and memory bandwidth) and other features like manufacturing process (for example SOI), power consumption, and size. Many new processors provide integrated error detection and correction techniques on the cache hierarchy and direct connection to memory via integrated interfaces. Additional benefits will be gained if processors integrate memory controllers on die. HRWS SAR application is a memory bound application. Shared memory architectures with coherent cache hierarchies are a good help for the programmer to face this challenge.

In the HRWS application, different data collected from different parts of the instrument are distributed and processed in 7 parallel panel processing stages and within each of them data is further distributed into 12 parallel tile processing stages that perform the same processing steps. This means that there can be 84 instances of the same processing steps executing in parallel. At the panel level each PPN will be processing its own data set, accessing most of the time its local memory. Having a good data locality, the HRWS SAR application will also benefit from vector and graphics processing capabilities of latest processor designs. Thread level parallelism can be easily exploited inside each tile processing of HRWS SAR, since some parallel processing steps work on the same data set.

In the latest years, a lot of research has been done from different processor vendors in the scope of many-core processors. At Intel, the Single Chip Cloud Computer (SCC) is an interesting many-core processor architecture [13]. One research chip contains 24 tiles with two x86 cores per tile, a 24-router mesh network with 256 GB/s bisection bandwidth, 4 integrated DDR3 memory controllers and hardware support for message-passing. Another

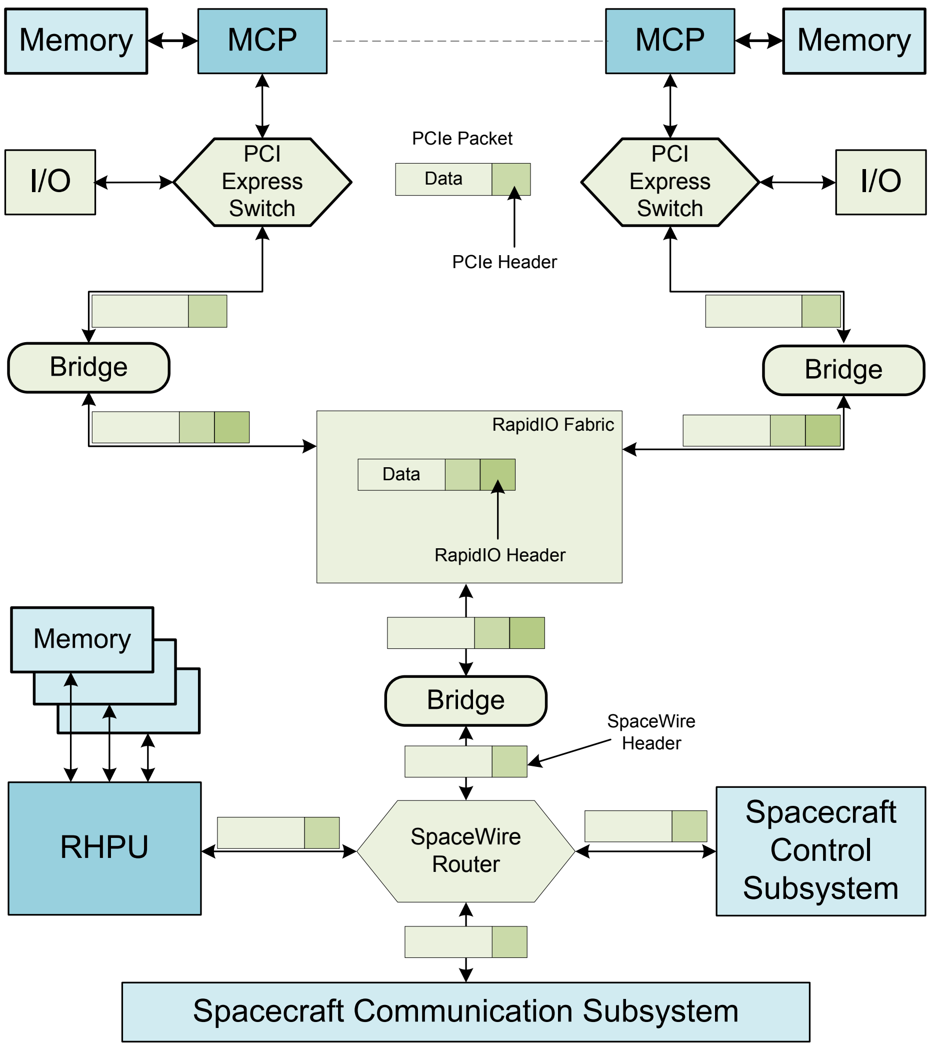


Figure 3.9: System Interfaces in the Reliable High Performance Architecture

many-core technology from Intel is the Teraflops Research Chip with 80 cores [77]. Each core contains two floating-point engines for accurate calculations in graphics, financial and scientific applications. Intel is also working on the upcoming accelerator Many Integrated Core (MIC) architecture [14] based on previous many-core research. The first 22 nm Xeon Phi coprocessor based on Intel MIC technology will be released soon. It will integrate many small cores with wider vector units and more hardware threads.

Another vendor, the Tilera corporation has launched different many-core processors based on the intelligent mesh (iMesh [80]) architecture. The first generation is the TILE64 family of processors that includes 64 processor cores (called tiles) interconnected with iMesh [119]. The second generation, the TILEPro family introduces the dynamic distributed cache technology that improves performance of the coherent cache. This is achieved by having two different sets of interconnects, one for cache and another one for memory and communication operations. The third and the last generation is the TILE-GX family that can integrate 36, 64, and 100 cores on a single device. This family of processors integrates PCIe controllers, memory controllers, and 1 Gbit/10 Gbit Ethernet ports, making it a very good solution for multi-chip designs.

On the other hand, both Intel and AMD have promoted processors with integrated graphics cores. These approaches tend to increase performance of the computing system in a power efficient way. The AMD Fusion technology combines multi-core CPU technologies with many-core GPU ones. The first released Accelerator Processing Unit (APU) codenamed Llano contains 2, 3 or 4 general purpose Huskey cores and a Sumo graphics core [17]. Each of the cores has a 1 MB L2 cache. The microarchitecture of the Huskey cores is based on Athlon II and the one of the Sumo graphics cores is based on the Radeon HD-5000 series. The successive APU codenamed Trinity is based on the Bulldoser core and AMD's second mainstream APU. Its GPU is significantly more powerful than Llano's and its CPU incorporates a number of improvements over Bulldozer.

Intel has recently promoted the Sandy Bridge graphics architecture which integrates GPU functionalities in the same die as the CPU [18]. In order to achieve power and area efficiency fixed functional hardware has been implemented. It can integrate on the same die up to 4 x86 cores, a GPU optimized for power or performance, memory, and PCIe controllers. An 8MB level-3 cache memory is shared among CPUs and the GPU. The data flow is improved by the Ring interconnect fabric that connects all components on die. The successive graphics architectures from Intel will be the 22 nm Ivy Bridge [120].

One of the most important vendors of pure GPU solutions is NVIDIA. The latest computing architecture from NVIDIA, codenamed FERMI, features 512 CUDA cores, NVIDIA parallel DataCache technology, NVIDIA GigaThread Engine and ECC support. One of the key features impacting performance directly is the memory hierarchy that includes local memory, L2 cache, and global memory. Local memory can be divided between L1 cache and shared memory of each SM. The next generation CUDA architecture codenamed Kepler integrates the next generation SMX with 192 CUDA cores, 32 SFU, and 32 load/store units. Other architectural features provided by Kepler are Dynamic Parallelism and Hyper-Q technology. More information is available in [121]. Table 3.1 summarizes main features of the discussed technologies.

For spacecraft designers, it is very important to know the Thermal Design Power (TDP) and the manufacturing process of components in the architecture. On the other hand, information on the number of cores operating at a certain frequency and the theoretical peak performance should be considered in order to select the most performing and power efficient processing module for the architecture. Multi-core and heterogeneous (multi-core with integrated graphics) solutions from Intel, AMD, and Tilera can be easily put inside each PPN of the architecture in the position of the MCP (Figure 3.2). Taking into consideration the above mentioned processor technologies and the overall proposed system architecture, spacecraft system designers will have to choose which approach to take and which technology to use. This decision depends also on the application and on how the application can exploit the parallelism in the architecture. A typical SAR application is built and used to benchmark various high performance systems. Next chapter describes the algorithm and the implemented application used for benchmarking purposes.

Table 3.1: Processor Technology Features

	Maximal TDP	No. of Cores at Frequency	Manufacturing Process	Peak Performance
Intel SCC	125 W	48 x 1 GHz	45 nm CMOS	12 Giga FLOPS
Intel TRC	62 W	80 x 3.2 GHz	65 nm CMOS	1 Tera FLOPS
Tilera TILEGX (100)	55 W	100 x 1.5 GHz	40 nm CMOS	1.2 Tera-Ops/sec
AMD Llano APU	100 W	4 x86 cores x 2.9 GHz + 400 Stream. Cores x 600 MHz	32 nm SOI	500 Giga FLOPS
Intel Sandy Bridge	95 W	4 x86 cores x 3.3 GHz + 850 MHz Processor Graphics	32 nm CMOS	375.04 Giga FLOPS
NVIDIA Fermi (Tesla)	250 W	448 CUDA Cores x 1.15 GHz	40nm CMOS	515 Giga FLOPS

3.5 Expected Problems in the Proposed Architecture

The proposal in this chapter combines various technologies to provide a solution for future space applications. Nevertheless, it is not possible to address all probable issues in the same solution. Being very flexible in design, the architecture can be configured according to various requirement levels in terms of performance and reliability. Main problems that might arise are related to power consumption, dissipated heat, and size.

The integration of many high performance computing components in the same system increases power consumption. Other solutions should be found on how to obtain that amount of power on-board of the spacecraft. If the system is used efficiently without wasting resources, the tradeoff between performance and power consumption might be acceptable. Applications have to be tailored to specific architectural features to increase efficiency.

Probably new technologies have to be developed for heat dissipation in the lack of air. Emerging hot and cold water cooling systems being implemented on Earth might have to be considered for spacecrafts too. The distribution of processing elements helps in reducing thermal issues, but increases interconnect latencies and complicates communication patterns if many such elements have to be integrated.

It might be hard to apply all proposed solutions in a real system architecture as new problems might arise when trying to implement it. Combining different technologies might add some additional question marks that are difficult or impossible to predict at the moment.

4 The Two-Dimensional Spotlight SAR (2DSSAR) Application

In order to mimic computational requirements represented from current and future space applications, a spotlight SAR benchmarking application is implemented. The purpose of this application is to benchmark various high performance computing systems. This application is selected from the SAR family of applications because such applications pose nowadays computing challenges. The increase of image resolution in SAR applications increases the size of data to be processed or downloaded to ground stations. Two approaches can be taken depending on the type of application. In bandwidth critical applications, a data compression is needed to reduce the amount of data that has to be downloaded to Earth.

For real-time Earth observation scenarios, on-board image preprocessing or complete image reconstruction is needed to fulfill the requirements. As the second approach poses more computational challenges, a benchmarking application that applies a complete SAR image reconstruction is implemented. Before going into more detail on the application implementation, some background information is given on spotlight SAR systems and image reconstruction algorithms focusing mainly on spatial frequency interpolation. This background information extracted from [122] is formulated and structured so that the reader finds it easy to understand the basis behind SAR synthetic data generation and digital reconstruction.

4.1 Basic Spotlight SAR Formulations

In spotlight mode, SAR images are typically taken from a fixed area on the ground (swath) that is typically on the side of the radar. Since range is the size of the swath in the direction the radar is looking into, swath's range resolution is determined by frequency (bandwidth) variations of the transmitted waveform. That means that it is determined by the width of transmitted pulse. Narrower pulses define finer range resolution [123].

Recall that cross-range (also known as azimuth) is the dimension perpendicular to range. Swath's cross-range resolution is determined by variations in the synthetic antenna aperture. To obtain a fine cross-range resolution, a physically large antenna aperture is needed to focus the transmitted and received energy into a sharp beam. The sharpness of this beam defines the cross-range resolution [123].

In order to easily understand digital processing algorithms, swath can be cut into tiles that are indexed by a pair of coordinates *x(h)* and *y(h)*. Each value of h indexes a particular tile on the swath (Figure 4.1). Each point on the swath operates on the received signal in a different way. A point further away from the radar has a longer round trip delay than a closer point. Different points on the swath reflect the signal at different intensities. The returns strength depends on the reflectivity of points on swath.

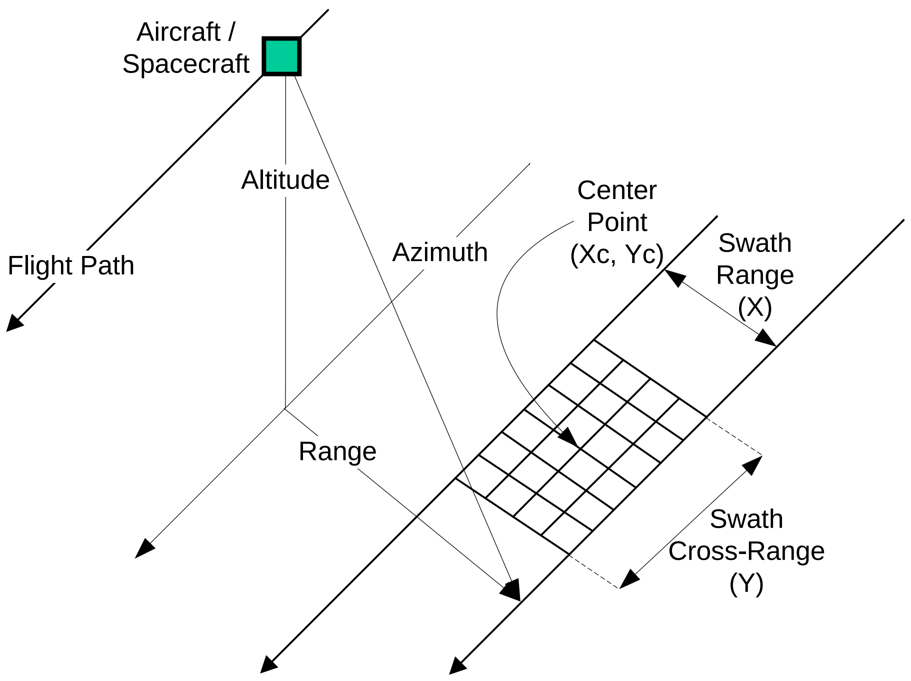


Figure 4.1: Illuminated Swath in Spotlight SAR

Typical spotlight SAR systems keep two different time dimensions for captured returns. Fast-time dimension (denoted by t) indexes through analog-to-digital samples, whereas slow-time dimension (denoted by u) indexes through transmitted pulses. In classic spotlight SAR systems, the transmitted signal is a time-varying linear frequency modulated chirp pulse train, as depicted in Figure 4.2 and defined by Equation 4.1:

$$p(t) = a(t)e^{(j\beta t + j\alpha t^2)}\ , \tag{4.1}$$

where:

$$a(t) = \begin{cases} 1 & for\ 0 \le t \le T_p \\ 0 & otherwise \end{cases},$$

T_p is the duration of the transmitted pulse, $j = \sqrt{-1}$ is the imaginary unit, α is the chirp rate, and β is the minimum spectral support band of the chirp. Frequency in a chirp signal increases or decreases with time. A linear chirp is a sinusoidal wave that increases in frequency linearly over time.

The returned signal is the sum of the originally transmitted signal and the attenuated time delayed echoes of the transmitted waveform itself (Figure 4.3). The time delay (ξ) is the time that the signal takes to complete a round trip from the radar to the swath tile and

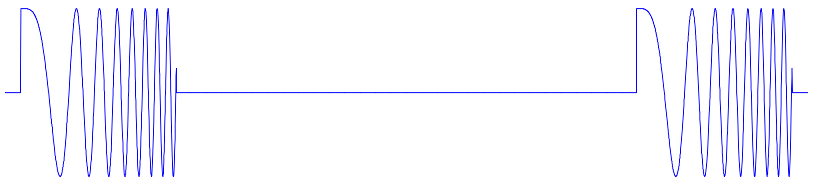


Figure 4.2: Transmitted linear chirp pulse train

back. The attenuated amplitude is due to the reflectivity constant (σ) at the particular tile where the signal is retransmitted back. Both ξ and σ might vary across different tiles on the swath. In an analytical way, the SAR return is expressed from Equation 4.2:

$$s(t,u) = \sum^{n} \sigma(n) p(t - \xi(n)) \,, \tag{4.2}$$

where $s(t,u)$ is the returned signal, σ is the reflectivity constant at the point of impact in the respective swath tile, p is the originally transmitted signal, ξ is the round trip time delay of the transmitted SAR signal, and n is the index through tiles on the swath.

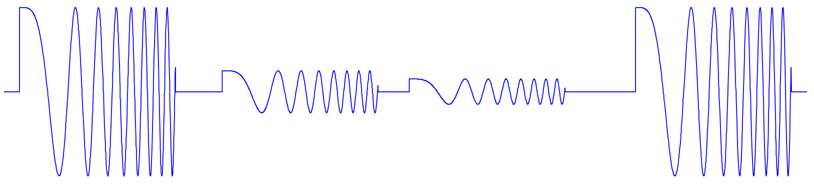


Figure 4.3: The linear FM chirp pulse train with two non-overlapping echoes returned from different points on the swath

Substituting the expression for the transmitted signal in Equation 4.1 into the SAR return Equation 4.2 gives Equation 4.3:

$$s(t,u) = \sum^{n} \sigma(n) a(t - \xi(n)) e^{(j\beta(t-\xi(n)) + j\alpha(t-\xi(n))^2)}. \tag{4.3}$$

Later, the linear FM chirp signal is de-ramped in slow time (compressed in frequency). This is done to ease processing in later steps. The round trip time delay (ξ) varies along range and cross-range as shown in Equation 4.4.

$$\xi(n) = \frac{2\sqrt{x^2 + (y-u)^2}}{c} \,, \tag{4.4}$$

where ξ is the round trip time delay from a given point on the swath, x is the range on the swath, y is the cross range on the swath, u is the slow time index, and c is the propagation speed constant ($3e^8$meters/second). This happens because of the circular radiation pattern of the transmitted waveform and the curvature of Earth at the location where the waveform hits the swath. The chirp rate measures the rate of change in the frequency of a waveform. Usually it is set to:

$$\alpha = \frac{jw_0}{T_p} = \frac{j2\pi f_0}{T_p} \,, \tag{4.5}$$

where α is the transmit linear FM chirp rate, $j = \sqrt{-1}$ is the imaginary unit, w_0 is the angular frequency of the signal (half-bandwidth, the radar signal baseband bandwidth is $2w_0$), f_0 is signal frequency measured in Hertz, and T_p is the duration (in seconds) of the transmitted chirp pulse. Usually a carrier is needed to transmit information as an electromagnetic wave through space. The carrier can be a sinusoidal waveform modulated with the input signal. It is used for signal transportation purposes. That is why the chirp is frequency modulated. Similar to other waveforms, the carrier can be disassembled into its spectral components. The minimum spectral component of the carrier band is set to:

$$\beta = w_c - w_0 = 2\pi(f_c - f_0) \,, \tag{4.6}$$

where β is the minimum spectral support band of the transmitted signal, w_c is the center angular frequency (in radians) of the carrier signal, w_0 is the signal angular frequency, f_c is the carrier signal frequency, and f_0 is the signals frequency measured in Hertz. The delayed fast time is $t_d = t - \xi$ and the range position on the swath is calculated as an offset to the swath center point:

$$x = X_c + x(h) \,, \tag{4.7}$$

where X_c is the range dimension of the point in the center of the swath, and $x(h)$ is the range to a particular tile on the swath. Substituting these implications in Equation 4.3 and Equation 4.4, the returned signal for each point on the swath (for $0 \le t \le T_p$) is:

$$s(t,u) = \sum^{n} e^{(j2\pi(f_c - f_0)t_d + \frac{f_0}{T_p} t_d^2)} \,, \tag{4.8}$$

where:

$t_d = t - \frac{2\sqrt{(X_c + x(n))^2 + (y(n) - u_c)^2}}{c}$,

u_c is the compressed slow time index ($|u_c| \le L$), $2L$ is the length of the synthetic aperture, and T_p is the ending time of the processing interval. Synthetic SAR return data is two-dimensional and consists of the number of pulses in the aperture (slow-time) by the complex number of Analog-to-Digital Converter (ADC) samples per pulse (fast-time).

4.2 Spatial Frequency Interpolation (SFI)

SFI is widely used in digital reconstruction, which converts raw SAR data into a perceptible SAR image. Initially the echoed SAR signal is transformed into frequency domain by applying Fourier transforms. Then, matched filtering is achieved by simply multiplying the echoed SAR signal with a baseband reference signal, which is also previously Fourier transformed into frequency domain. The signal is then interpolated from a wedge to a rectangular area. Finally, a two-dimensional inverse Fourier transform is applied to transform the signal into visible spatial domain. The whole digital reconstruction process is illustrated in Figure 4.4.

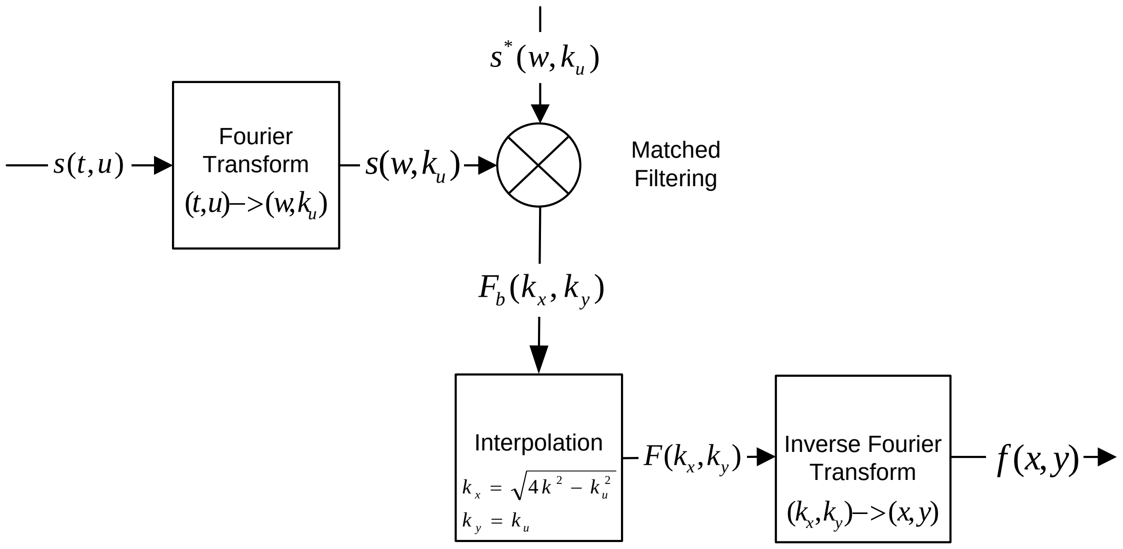


Figure 4.4: Spotlight SAR Digital Reconstruction Algorithm

The input $s(t,u)$ is the baseband echoed SAR signal, where t is the index of the ADC sampling in fast-time and u is the index of the pulse number in slow time. After applying two-dimensional Fourier transform, $s(w,k_u)$ is generated in frequency domain. Here w indexes the fast-time angular frequency and k_u indexes the slow-time frequency. The complex conjugate of $s(w,k_u)$ is denoted as $s^*(w,k_u)$. The output of the interpolation step is $F(k_x,k_y)$, where k_x indexes thought the interpolated angular frequency in fast-time and k_y indexes through the interpolated frequency in slow-time, which in this case is equal to k_u. In spatial domain, the resulting function $f(x,y)$ represents the spotlight SAR image, where x is the range dimension index of the SAR image and y is the cross-range (azimuth) dimension index of the SAR image. The first Fourier transform is applied to transform the SAR returns signal $s(t,u)$ into frequency domain to facilitate matched filtering, which takes place later. When substituting Equation 4.4 in Equation 4.2, the measured echoed signal is formulated as:

$$s(t,u) = \sum^{n} \sigma_n \, p \left[t - \frac{2\sqrt{x_n^2 + (y_n - u)^2}}{c} \right]. \tag{4.9}$$

The Fourier transform of $s(t,u)$ with respect to fast-time index t is:

$$s(w,u) = P(w)\sum^{n} \sigma_n e^{-j2k\sqrt{x_n^2+(y_n-u)^2}} , \tag{4.10}$$

where $k = w/c$ represents the wavenumber and $P(w)$ is the Fourier transform of the known transmitted SAR signal. The SAR signal in the (w,u) domain is composed of a combination of spherical pulse modulated signals, i.e. $e^{-j2k\sqrt{x_n^2+(y_n-u)^2}}$. Fourier properties of such signals have been examined in [122]. Further detailed analyzes concerning these spherical signal exceeds the purpose of this dissertation. In order to define Fourier transform with respect to slow-time t domain, Fourier transform of the spherical PM signal is formulated as:

$$F_u\left[e^{-j2k\sqrt{x_n^2+(y_n-u)^2}}\right] = e^{-j\sqrt{4k^2-k_u}x_n - jk_u y_n} , \tag{4.11}$$

for $k_u \in [-2k,\, 2k]$. k_u is the slow time frequency domain, which is often referred in SAR imaging systems as the slow time Doppler domain. Taking into consideration the Fourier transform (Equation 4.11) of the spherical PM signal, the Fourier transform of $s(w,u)$ in Equation 4.10 with respect to slow-time u is:

$$s(w,k_u) = P(w)\sum^{n} \sigma_n e^{-j\sqrt{4k^2-k_u}x_n - jk_u y_n}. \tag{4.12}$$

This means that indexes k_x and k_y are defined by the following functions:

$$k_x(w,k_u) = \sqrt{4k^2 - k_u^2} \tag{4.13}$$

$$k_y(w,k_u) = k_u \tag{4.14}$$

After applying Fourier transforms in slow time and fast-time, the baseband target function is reconstructed in spatial frequency domain. This is achieved by a two-dimensional matched filtering of the measured SAR signal with the reference signal (Equation 4.15).

$$F_b\left[k_x(w,k_u), k_y(w,k_u)\right] = s(w,k_u) \otimes s^*(w,k_u) \tag{4.15}$$

After matched filtering, the baseband target function is interpolated with a tapered *sinc* function. The tapered function is selected to reduce computational costs. The interpolation equation is:

$$F(k_x, k_{ymn}) \approx \sum^{|k_x - n\Delta_{k_x}| \leq N_s \Delta_{k_x}} F(k_{xn}, k_{ymn}) h(k_x - n\Delta_{k_x}) \,, \tag{4.16}$$

where:

$$h_w(k_x) = sinc\left(\frac{k_x}{\Delta_{k_x}}\right) = \begin{cases} h(k_x)w(k_x) & for\ |k_x \leq N_s \Delta_{k_x}| \\ 0 & otherwise \end{cases} \,, \tag{4.17}$$

N_s is a constant parameter that determines the support region (size) of the window ($[-N_s\Delta_{k_x}, N_s\Delta_{k_x}]$). It denotes the half number of *sinc* side lobes used for interpolation, $\Delta_{k_x}$is the Nyquist criterion in range spatial frequency domain for evenly spaced data, i.e. $k_{xn} = n\Delta_{k_x}$, $w(k_x)$ signal is referred to as the Hamming window [124] and it has the following specific formulation:

$$w(k_x) = \begin{cases} 0.54 + 0.46cos\left(\frac{\pi k_x}{N_s \Delta_{k_x}}\right) & for\ |k_x| \leq N_s \Delta_{k_x} \\ 0 & otherwise \end{cases} \,. \tag{4.18}$$

The time-domain target function $f(x, y)$ is obtained by applying an inverse Fourier transform to the interpolated baseband target function:

$$f(x, y) = \sum_{k_y} \sum_{k_x} F(k_x, k_y) e^{(jk_x x + jk_y y)} . \tag{4.19}$$

4.3 2DSSAR Application Structure

2DSSAR is based on formulations stated in previous sections. Its structure is adopted from the HPCS Scalable Synthetic Application Number 3 (SSCA#3 [125]). 2DSSAR defines a general front-end sensor processing chain that can be used in development experiments, system specification and performance testing. As such, it features computations, communications and data I/O requirements that are present in many types of sensor processing applications. 2DSSAR cannot be considered as a solution to an existing sensor processing problem but as a processing chain that consists of computation and data I/O stages applied to synthetic scalable data with verifiable results. It is completely coded in C programming language, so that it can be easily optimized and parallelized for different high performance computing architectures.

Problem size is scalable since 2DSSAR is able to synthetically generate its own sensor data based on configurable parameters. It can be scaled to address the widest range of potential SAR systems ranging from normal to wide swath. The principal performance metric is throughput, or in other words, the rate at which sensor data is generated and the SAR image is reconstructed.

2DSSAR can be run in one of the following three modes:

1. Compute Mode, which maximizes the amount of needed computations but minimizes data I/O operations.

2. Data I/O Mode, which maximizes data I/O requirements and reduces the amount of computations.

3. System Mode, which combines both Compute and Data I/O Modes.

As shown in Figure 4.5, 2DSSAR is composed of two processing stages, the Synthetic Data Generation (SDG) and the SAR Sensor Processing (SSP). In SDG, SAR returns are generated from a uniform grid of synthetic point reflectors (discussed in section 4.1 on page 53). Sensor data returns are stored to disk before entering the second stage. This makes it possible to execute multiple times SSP stage after generating only once the synthetic data. SSP reads sensor data from disk, applies digital reconstruction via spatial frequency interpolation (discussed in section 4.2 on page 57), and writes the resulting reconstructed image to disk for verification purposes.

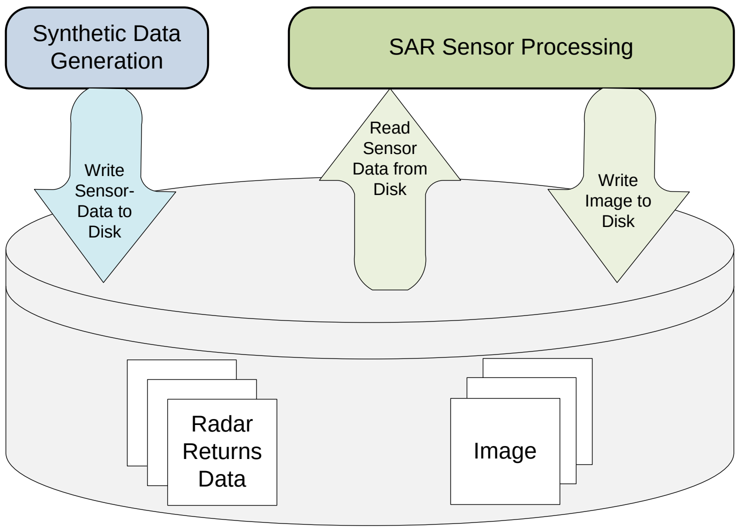

Synthetic Data Generation
SAR Sensor Processing
Write Sensor-Data to Disk
Read Sensor Data from Disk
Write Image to Disk
Radar Returns Data
Image

Figure 4.5: Two-Dimensional Spotlight SAR

In Compute Mode only one set of sensor data is generated and stored by SDG and only one image is reconstructed in SSP, whereas in System Mode multiple sets of sensor date are generated from SDG and a grid of images is reconstructed in SSP. Compute Mode is used when running the application for verification purposes. System Mode is used when multiple identical runs are needed for performance measurements. Data I/O Mode is never used for benchmarking purposes because data I/O transactions (especially with hard disks) are not relevant for on-board space applications.

4.3.1 Synthetic Data Generation (SDG)

SDG is the first stage of 2DSSAR that is used to produce raw SAR data approximates, which are similar to what would be obtained from a real SAR system. Pulse trains are transmitted as a spacecraft flies adjacent to its terrain of interest or *swath.* Echo returns are scaled to mimic different reflection coefficients at various points on the swath, time delayed to mimic different times at which echoes are returned from different points on the swath, and finally summed together. The size of the SAR synthetic aperture is determined by the distance that the sensor flies while the radar is capturing returns from ground. This realizes a challengingly long antenna aperture length.

SAR simplifications are assumed in order to reduce repetitive code complexity. First, SAR processing is fixed to broadside. This means that the swath is fixed to the spacecraft's broadside to avoid squint angle processing considerations. A squint angle is an offset from the normal transmission angle of the antenna. In order to achieve this, the center of the swath (X_c,Y_c) is set to $Y_c = 0$. This means that the swath is at 90^o from the flight path (Figure 4.6). The synthetic aperture ($2L$) is set equal to the swath cross-range ($2Y_0$). In other words, $L = Y_0$. The last simplification is that all reflector magnitudes have been set to maximum value $'1'$. The simulated swath is of size $2X_0$ by $2Y_0$ meters or m_c by n time samples.

SDG is composed of multiple processing steps that make it possible to distinguish different parts in the algorithm implementation. In the first part, some SAR related constants and parameters are defined. They can be categorized into temporal (frequency) and spatial (geometric) parameters. Then some SAR data handling arrays are allocated. Main parameters are defined in Table 4.1.

Table 4.1: SAR Related Parameters

Parameter Description	**Parameter Definition**
The synthetic aperture half length	$L = 100 \times Scale$
The ratio of swath range to cross-range	$aspectRatio = 0.4$
Half length of the swath cross-range	$Y_0 = L$
Half length of the swath range	$X_0 = aspectRatio \times Y_0$
Number of slow-time samples in the compressed synthetic aperture	$n = 2 \times \left\lceil 80 \times Scale \right\rceil$
Number of fast-time samples in swath range	$m_c = 2 \times \left\lceil 158.5 \times Scale + 60 \right\rceil$

Taking into consideration these definitions, for a $Scale = 60$, the simulated size of the swath is 12000 by 4800 meters or 19140 by 9600 time samples. These numbers come from the following calculations:

$2 \times X_0 = 2 \times aspectRatio \times Y_0 = 2 \times 0.4 \times L = 2 \times 0.4 \times 100 \times 60 = 4800$

$2 \times Y_0 = 2 \times L = 2 \times 100 \times 60 = 12000$

$m_c = 2 \times \left\lceil 158.5 \times Scale + 60 \right\rceil = 19140$

$n = 2 \times \left\lceil 80 \times Scale \right\rceil = 6900$

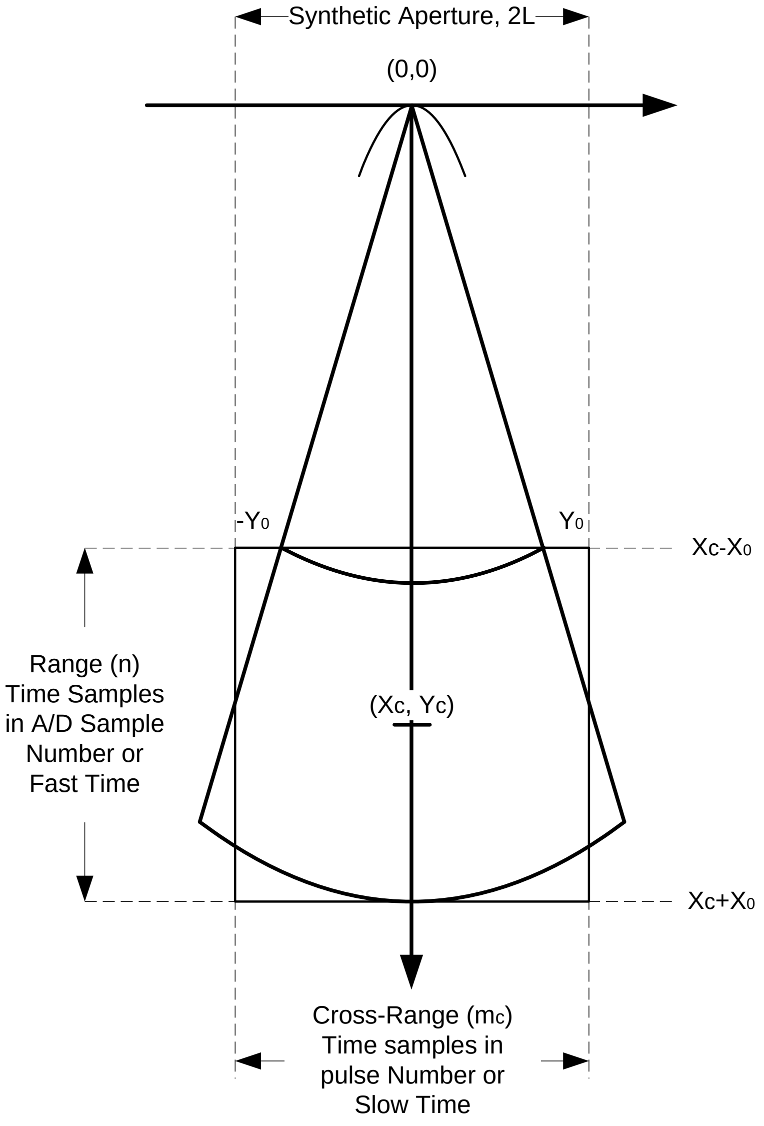

Synthetic Aperture, 2L
(0,0)
-Y0
Y0
Xc-X0
Range (n) Time Samples in A/D Sample Number or Fast Time
(Xc, Yc)
Xc+X0
Cross-Range (mc) Time samples in pulse Number or Slow Time

Figure 4.6: Geometry of Broadside SAR

Synthetic radar returns are created by summing simulated individual returns from a mesh of point reflectors. In order to achieve this, a uniform rectangular grid of point reflectors is generated on the swath area. Each point corresponds to a distinct location on the swath, where a simulated point reflector is located. Synthetic SAR returns are the summed-up of echo return signals from each reflector on the swath. Each echoed pulse chirp signal is the originally transmitted radar signal that is uniquely time-delayed. The echoed signal is then converted to a baseband lowpass signal for later processing. A baseband signal is a low frequency signal, which simplifies Fourier analysis. Then, the fast-time reference signal is constructed from a unit reflector at the center of the swath. A fast-time filter is needed for digital spotlighting and bandwidth expansion along slow-time. This filter is obtained by transforming the baseband reference signal into the Doppler (spatial frequency) domain through a Fast Fourier Transform (FFT).

Algorithm 4.1 illustrates the pseudo code of the most computationally intensive part in SDG. It computes the hyperbolic time delay to each reflector placement and integrates synthetic echoes from the transmitted waveform after ADC. Regarding the type of computations, the calculation of the delay in fast time t_d is a computation applied on 32-bit floating-point variables, whereas the accumulation in Equation 4.8 is computed on complex 32-bit floating-point variables.

Algorithm 4.1 Intensive Computations in SDG

```
for $h$ = $0$ to $nReflectors$ do
  for $i$ = $0$ to $n$ do
    for $j$ = $0$ to $mc$ do
      $t_d(i,j) = t(i) - \frac{2\sqrt{(X_c+x_n(h))^2+(y_n(h)-u_c(j))^2}}{c}$
      if $0 \leq t_d(i,j) \leq T_p$ do
        $s(t,u) = \sum^n e^{(j2\pi(f_c-f_0)t_d+\frac{f_0}{T_p}t_d^2)}$ (Equation 4.8 on page 56)
      end if
    end for
  end for
end for
```

The number of iterations is specified by the number of samples in the compressed synthetic aperture defined by index mc (slow-time), and the number of fast-time samples defined by index n. The simulated swath area results of size m_c by n. Locations on the swath where point reflectors are to be placed are referenced by the index $nReflectors$. All of the three index values (n, mc, and $nReflectors$) are configured in 2DSSAR by the *Scale* parameter. This dependence of indexes on the *Scale* parameter is shown in Table 4.2, starting from small scales to large ones. For a scale of 60, more than 25 trillion ($19140 \times 9600 \times 146132 = 26850878208000$) iterations have to be executed. This means that it takes some time to generate SAR sensor return data for the 2DSSAR application. In order to reduce the generation time, SDG is optimized and parallelized for shared memory multi-core environments (see section 5.2 on page 74).

Table 4.2: Index Values Depending on Scale Parameter

Scale	**m_c**	**n**	**nReflectors**
5	1706	800	1040
10	3290	1600	4108
20	6460	3200	16377
30	9630	4800	36652
60	19140	9600	146132

4.3.2 SAR Sensor Processing (SSP)

SDG stage is followed by a front-end sensor processing stage that consists of raw data retrieval, Image Reconstruction (IR), and image storage for further verification. SSP startup routines retrieve the SDG's SAR coefficients, support parameters, and SAR sensor generated data. Then, SAR sensor data is interpolated in spatial frequency to build the reconstructed image, which is later stored to disk for verification. Depending on the operation mode, different execution scenarios exist. In Data I/O mode, a large random matrix is generated, instead of reconstructing a SAR image. In Compute mode, the generated SAR raw data is retrieved and a single image is reconstructed. In System mode, multiple iterations of the complete 2DSSAR are run. In each iteration, random SAR sensor data is retrieved and processed. This random retrieval of SAR sensor data is intended to mimic the random order in which data is acquired in a typical system. At the end of each iteration, one reconstructed SAR image is stored in a random location within an external image grid. The number of iterations and the size of the grid can be easily configured before compiling 2DSSAR.

After the random SAR sensor data has been retrieved, the echoed signal is filtered along fast-time by correlating the synthetic baseband echoed signal with the fast time filter, which is earlier created in SDG. The filtered signal is then digitally spotlighted and bandwidth expanded using slow-time compression and decompression in spatial frequency (Doppler) domain. The spatial frequency domain compressed signal (m_c by n) is then zero padded to produce an array of m by n complex numbers. The spotlight SAR image is reconstructed, so that it can fit a polar mesh. Finally, it is reshaped to fit a rectangular mesh. This interpolation is done in spatial frequency domain and is referred to as digital reconstruction via SFI. Figure 4.7 illustrates the transformation of raw data to reconstructed image focusing on interpolation mapping. The upper left side shows generated SAR sensor data before entering SSP. The upper right side shows the reconstructed image after SSP. The lower left side shows the logical representation of the raw SAR data in polar coordinates, whereas the lower right side shows the logical representation of the reconstructed image in rectangular coordinates.

Image reconstruction with SFI is discussed in section 4.2 on page 57 and the algorithm is depicted in Figure 4.4 on page 57. In this substage, raw data is converted from temporal to spatial domain and interpolated from polar to rectangular swath to form the desired image. SFI consists of:

1. An FFT into the frequency domain,
2. Pulse compression or matched filtering to remove spectral components of the transmitted waveform,
3. Spatial frequency domain interpolation, to change the representation of the swath from polar to rectangular coordinates, and
4. A two-dimensional inverse FFT.

First, the size of the resulting reconstructed image has to be determined, which is m by n_x pixels. The scaling relationships are defined in Equation 4.20 (85 % compression scale) and Equation 4.21. In Equation 4.21, kx_{max} and kx_{min} are respectively the maximum and the minimum wavenumber. They are obtained by calculating maximum and minimum

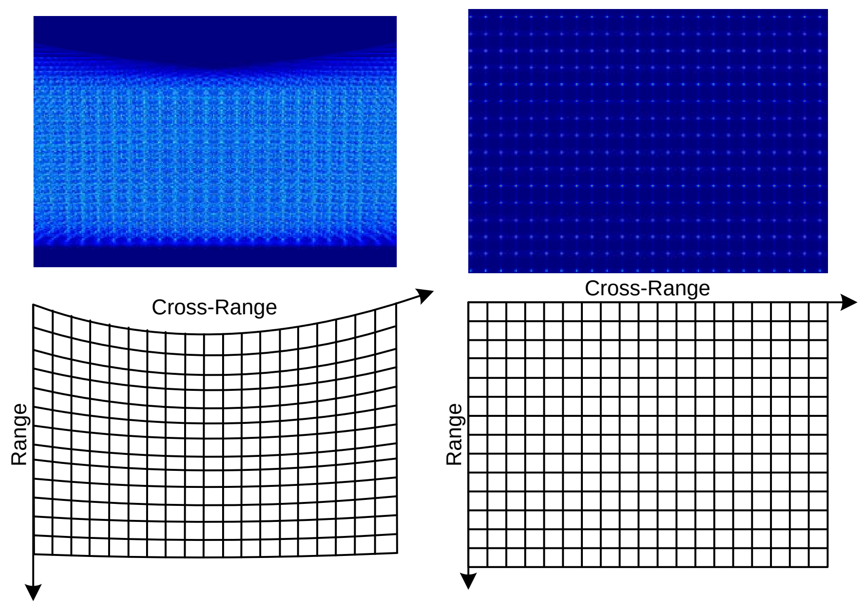


Figure 4.7: Reconstruction and Interpolation (Polar to Rectangular Coordinates)

values of the spatial frequency (Doppler domain) in range dimension. Table 4.3 shows for different scales, the size of the raw SAR sensor data and of the reconstructed image in each dimension.

$$m = \left\lceil \frac{m_c}{0.85} \right\rceil \tag{4.20}$$

$$n_x = 2 \times \left\lceil 20 \times Scale \times \left(\frac{kx_{max} - kx_{min}}{\pi} \right) \right\rceil + 20 \tag{4.21}$$

Table 4.3: Size of SAR Sensor Data and Reconstructed Image

	SAR Sensor Data		**Reconstructed Image**	
Scale	m_c	n	m	n_x
10	3290	1600	3808	2474
30	9630	4800	11422	7380
60	19140	9600	22844	14738

In order to remove spectral components of the reference signal, the spotlighted signal is match-filtered with the complex conjugate reference signal along fast-time and slow-time. First, the conjugate complex reference signal is generated, which is the Doppler domain representation of the reference signal. Then, the Doppler domain two-dimensional matched filtering in polar format takes place. In terms of processing steps, the fast-time filtered signal is compressed along slow-time and then narrow-bandwidth processed in polar format. This reconstruction along compressed slow-time picks up each row along slow-time, FFTs it, and puts it back in its corresponding slow-time location.

Then, the compressed signal (of size: m_c by n) is zero-padded along slow-time producing an m by n array of complex numbers. In order to transform the zero-padded spatial spectrum back to temporal domain, each row is picked up, inverse FFT-ed, and put back in its corresponding location. At this point a decompression takes place in slow-time. Compression and decompression processing steps are applied element-by-element, row-by-row in two-dimensional complex arrays. Finally, digital spotlighting in slow-time involves a row-by-row FFT of the SAR signal spectrum array and then an *fftshift* operation on both dimensions.

A two-dimensional *fftshift* operation moves the lower left corner (lowest frequency component) to the center of the image to prevent edge processing effects that might be introduced in later processing steps. As shown in Figure 4.8, *fftshift* swaps the first (A) quadrant with the third (C) and the second (B) quadrant with the fourth (D). For one-dimensional data in cross-range (slow-time) a one-dimensional *fftshift* is needed to swap the first and second halves of the data vector.

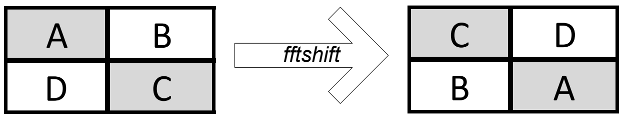


Figure 4.8: The Notion of *fftshift* Operation

When matched filtering is finished, some parameters have to be configured before entering the interpolation loop. A tapered window is used to reduce computation costs of interpolation. Due to the non-linear nature of the two-dimensional mapping from temporal to spatial domain conversion, the resulting data is unevenly spaced as shown in Figure 4.9.

Algorithm 4.2 describes the interpolation process, which involves upsampling and interpolating in order to increase the range resolution of the reconstructed image. Interpolation maps each row in the frequency-domain match-filtered input image to several rows in the output image. The number of output rows that each input row is mapped to, is determined by the number of sidelobes in the selected interpolation function. A *sinc* function is used for the interpolation window and 8 interpolation sidelobes are defined for unevenly spaced data. In other words, each row in the input image is mapped to 8 rows in the interpolated output image. This 1-to-8 mapping adds to the application non-deterministic features when thinking about parallelization and data distribution, because overlapped regions arise among mappings of different lines. The fact that this mapping is internally implemented by indirect addressing obscures a lot of information to both compiler and programmer, which in turn should take care of data integrity. Parallelization techniques and data integrity solutions are discussed in more detail in later chapters.

The final step of the image reconstruction substage uses two-dimensional *fftshift* operation to transform back the image to its original shape and a two-dimensional inverse FFT to transform the image into viewable spatial domain coordinates. The two-dimensional inverse FFT is implemented by two batches of one-dimensional inverse FFTs, one along rows and one along columns. The resulting viewable image is obtained by getting the magnitude of each element representing pixel intensities in the spatial image. In other words,

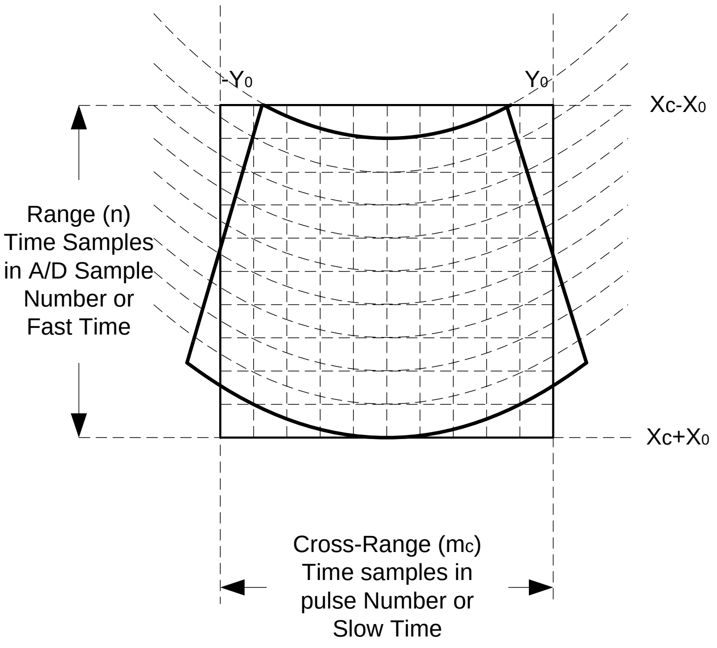

-Y0
Y0
Xc-X0
Xc+X0
Range (n) Time Samples in A/D Sample Number or Fast Time
Cross-Range (mc) Time samples in pulse Number or Slow Time

Figure 4.9: SAR Spatial Frequency Mapping

Algorithm 4.2 Interpolation Algorithm

```
for i = 0 to n do    //in sample number, slow time or range
  for j=0 to m do    //in pulse number, fast-time or cross-range
    Calculate the index of the closest cross-range sliver
    for h=0 to S do    //'S' is the interpolation cross-range sliver size
      Calculate slice indexes (ikxRow, ikxCol)
          that includes the cross-range sliver at its center
      Calculate the interpolating sinc() and Hamming windows
      InterpolatedSignal[ikxRow][ikxCol] +=
                MatchedFilteredSignal[i][j] * (sinc * ham);
    end for
  end for
end for
```

the absolute value of each complex number has to be calculated. Table 4.4 summarizes all SSP processing steps, their type of computation, and the data size in each step. Table 4.5 lists the amount of floating point operations needed for image reconstruction of different scales. More than one Tera operations are needed for large scale image reconstruction, which is mainly used for benchmarking.

Once image reconstruction substage has ended, the reconstructed image is stored to disk for verification purposes. The verification process is defined as optional and can be enabled or disabled in source files, before compilation, at the same time parameter *Scale* is also set. The file-system location where to store the image can be also specified at this point in time. For the verification, visualization functions from Octave [126] are used to get image

Table 4.4: SSP Processing Steps

	Processing Step	**Type of Computation**	**Data Size**
1	Read raw SAR sensor data from disk	Input/Output (complex float)	m_c by n
2	Filter the echoed signal in fast-time	Column-Wise 1-D Forward FFT, Element-Wise Multiplication	m_c by n
3	Compress the filtered signal along slow-time	Element-Wise Complex Exponential (CEXP) and Multiply-Acumulate (MAC)	m_c by n
4	Narrow-bandwidth polar format reconstruction along slow-time	Row-Wise 1-D Forward FFT	m_c by n
5	Zero pad the spatial frequency domain's compressed signal	Padding	m_c by n-> m by n
6	Decompress the zero-padded signal in slow time	Element-Wise CEXP and MAC	m by n
7	Digitally spotlight the SAR signal spectrum	Row-Wise 1-D Forward FFT	m by n
8	Generate the reference signal's complex conjugate	Element-Wise CEXP and MAC	m by n
9	Circumvent edge processing effects	2-D *fftshift*	m by n
10	Interpolation Loop	Element-Wise MAC, Sine, Cosine	m by n -> m by n_x
11	Transform signal from Doppler domain to spatial domain	Row-Wise 1-D Backward FFT,Column-Wise 1-D Backward FFT	m by n_x
12	Transform the spatial domain signal into a viewable image	Calculate the absolute value of each element (CABS: complex float -> float)	m by n_x
13	Write image to disk	Input/Output (float)	m by n_x

Table 4.5: Required Floating Point Operations for IR

Scale	**Floating Point Operations**
10	29. 54 Giga
30	115.03 Giga
60	1.302 Tera

data from disk and to display it on the screen.

4.4 C Programming Language Considerations

Usually, most of SAR algorithms are coded in Matlab as it is considered a programming language for engineering and science. Matlab libraries and functions tend to simplify coding of different scientific algorithms, but it might not be the best approach to take when performance estimations have to be done. This is the reason that 2DSSAR application is coded in C programming language. Another advantage is that it is easier to parallelize C code and the programmer has full control over parallelization techniques applied in C. Most of computationally intensive functions in Matlab are internally parallelized to efficiently execute on parallel architectures, but they are optimized for general approaches and do not take into consideration specific hardware resources and features. In order to get the best performance out of a parallel architecture, it is generally needed to parallelize complete steps or stages, and not only single functions.

Another advantage of using C over Matlab is the memory management system. C programmers can manage statically or dynamically data allocations and layouts. Taking into consideration the fact that 2DSSAR has to be configurable to process different sized images, all two-dimensional arrays representing raw data and reconstructed image have to be dynamically allocated. Some one-dimensional arrays, supposed to become large for large scales, are also being allocated dynamically. Only small data sets that do not depend on the scale factor are allocated statically.

A strong difference between Matlab and C is that Matlab stores multidimensional arrays in a column-major order in the linear memory, whereas C programming language uses a row-major order. The layout is very important in order to obtain performance when accessing multidimensional arrays because accessing contiguous array elements is usually faster than accessing non-contiguous ones. This is mainly due to data caching in multiple levels of modern computing architectures.

A lot of signal processing libraries are also available in C. This simplifies the transition from Matlab to C for many processing steps in SDG and SSP stages. FFTW library [127] is used for FFT transforms, taking place in both SDG and SSP stages. FFTW is a C subroutine library for computing the discrete Fourier transform in one or more dimensions, of arbitrary input size, and of both real and complex data, as well as of even/odd data, i.e. the discrete cosine/sine transforms. Column-wise one-dimensional FFT transforms (step 2 and 11 in Table 4.4) are implemented by a transposition of the two-dimensional data followed by a row-wise one-dimensional FFT. After analyzing all steps in SDG and SSP, it turns out that there are some consecutive steps processing the same data set. Data locality is improved by merging some of these steps together. In this way, data stays longer in cache memories, minimizing so the number of cache misses per cycle.

5 Benchmarking Shared Memory Systems

In order to further define components and logical interactions between them in the proposed on-board computing architecture, high performance computing systems have to be benchmarked with a real space application. Benchmarking results can also provide information on how the application exploits efficiently the available computational power. SSP stage of 2DSSAR application is of very interest for on-board computing. The main substage of SSP is IR.

This chapter focuses on shared memory systems and their benchmarking with SSP. It starts with a description of benchmarked shared memory platforms. For each of the benchmarked platforms, different parallelization techniques and optimizations are described, discussed, and applied to SSP and IR. Performance results obtained from the parallelization of SSP on such systems are discussed in conjunction with respective parallelization techniques and optimizations. Different programming models are used for the parallelization of SSP. In order to create an effective conclusion, a set of results is evaluated and interpreted.

5.1 Benchmarked Shared Memory Computing Platforms

2DSSAR application is run on two different shared memory platforms, one NUMA and one UMA SMP platform. On the NUMA platform, latency problems related to possible remote memory accesses are investigated among other performance estimations. On the UMA SMP platform, memory bandwidth related problems are addressed. This section describes features and logic representations of the underlying hardware of each platform.

5.1.1 The NUMA Shared Memory Platform

The first benchmarked platform is a dual-socket ccNUMA[1] platform. Figure 5.1 shows that it is composed of 2 Nehalem [128] chip-multi processors (Intel Xeon X5670) working at 2.93 GHz, each with 6 cores and with direct access to 18 GB of main memory via one integrated memory controller. This results in a total of 36 GB of shared memory. The multilevel cache hierarchy includes 64 KB L1 cache (32 KB L1 Data + 32 KB L1 Instruction private cache for each core), 256 KB unified L2 cache also private for each core, and 12 MB L3 shared cache that allows each core to use the complete cache memory space. This enables very efficient data sharing between threads running on different cores. Simultaneous multithreading (Hyper-Threading according to Intel terminology) is enabled to allow instructions from different instruction streams or threads to be executed simultaneously or interleaved on functional units inside a processor core. As a combination of

[1]Cache Coherent NUMA

ILP and TLP, SMT is supported by speculative execution, which is based on second-level branch prediction and pre-fetching.

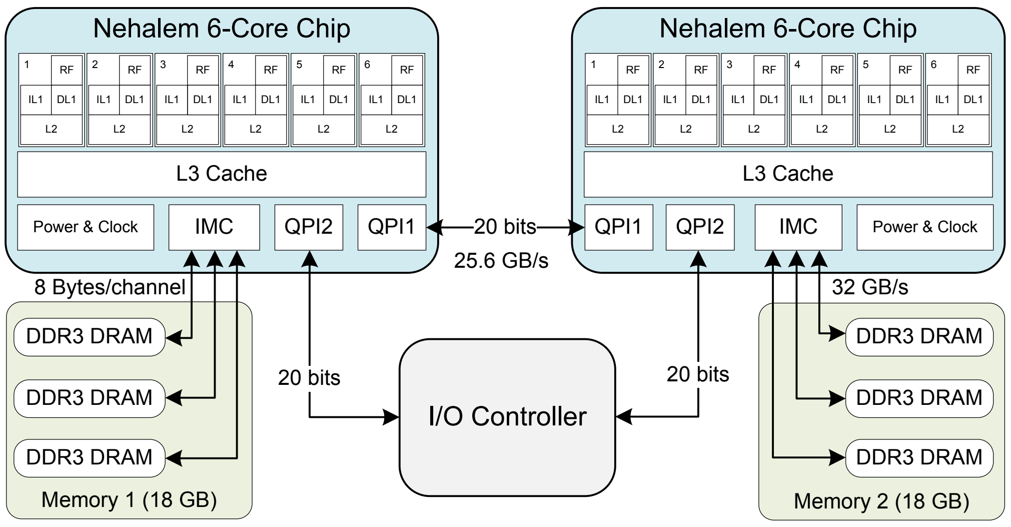


Figure 5.1: The ccNUMA Shared Memory Platform

Each Nehalem processor chip integrates two QuickPath Interconnect (QPI [9]) ports, one memory controller, and complementary circuits for cache coherence, power control, system management, and performance monitoring logic. QPI in high-end Intel models replaces the legacy Front Side Bus (FSB). It provides point-to-point high-speed full-duplex links with an aggregate bandwidth of up to 25.6 GB/s for CPU-to-CPU and CPU-to-I/O communications. In such a NUMA architecture, QuickPath technology increases performance by improving accesses to remote memories.

The Integrated Memory Controller (IMC) supports three 8-byte memory channels of DDR3 SDRAM operating at 1066 MHz. The total theoretical bandwidth between DRAM and IMC is around 32 GB/s. The integration of the memory controller on chip makes it easy to implement NUMA systems when multiple chips are used in the architecture. The combination of QPI and IMC accelerates data movements in the ccNUMA platform and avoids memory bottlenecks by having each multi-core processor interfaced to a local memory. QPI and IMC provide a big support in maintaining memory consistency between caches that reside on different chips by assisting the cache coherency protocol.

Each Nehalem CPU is manufactured in 45 nm process and has a maximal TDP of 95 Watt. TDP refers to the maximum power that has to be dissipated by the cooling system. When CPUs consume electrical energy they dissipate this energy in transistor work, but some part of this energy gets lost in form of heat because of the presence of impedance in electronic circuits. Different processor vendors define TDP as the maximum power consumption when running worst-case workloads for thermally significant periods.

The ccNUMA platform runs on Ubuntu 11.04 Linux Operating System. Linux programming environment [129] is composed of tools for compilation, profiling, debugging, and testing. C/C++ programs can be compiled with GNU C/C++ compilers, *GCC* and *G++*. The GNU profiler *GPROF* [130] is a tool that is used to collect and arrange statistics on the application. For debugging, the GNU Debugger *GDB* is used to see what is going on inside the application while it is executing, or what the application is doing at the moment it crashed. Eclipse programming platform is used on top of Linux programming environment.

Eclipse C/C++ Development Tooling [131] provides a functional development environment with support for project creation and configured build for various toolchains. Projects created with Eclipse can be moved, compiled, and run also on platforms that do not have Eclipse installed because everything is based on makefiles that use tools from the Linux programming environment. Periscope tool is used for parallel performance analysis. Periscope [132] is a distributed automatic on-line performance analysis system developed at the department of Computer Architecture of TU Munich (LRR-TUM). It is simple and straightforward to use as it can be integrated with Eclipse [133]. Table 5.1 lists software environment details of the ccNUMA platform.

Table 5.1: Software Environment on the ccNUMA Platform

	Name	**Version**
Operating System	Ubuntu Linux 11.10	Linux 3.0.0-24-generic x86_64
Compiler	GNU C Compiler (GCC)	4.6.1
Debugger	GNU Debugger (GDB)	7.3-2011.08
Profiler	GNU Profiler (GPROF)	2.21.53.20110810
FFT Library	FFTW	3.2.2
MPI Library	OpenMPI	1.5.1

5.1.2 The UMA SMP Platform

The other benchmarked platform is a UMA SMP shared memory platform. It is an IBM x3850 M2 system with 24 cores and 48 GB main memory. The x3850 M2 is a standalone server with four processor sockets and up to 32 dual in-line memory module sockets. It uses the fourth generation of IBM XA-64e chipsets. This platform (Figure 5.2) consists of four Intel Xeon E7450 processors (6 cores each), one Hurricane 4 memory and I/O controller, eight high-speed memory buffers, one PCIe bridge, and one south bridge.

The distinguished feature of such platform is the centralized memory controller, which is a separate module that provides uniform memory access time to each processor and processor core. Compared to NUMA platforms, UMA ones can be considered an older architecture as it has the drawbacks discussed in section 2.1 on page 7. Nevertheless, applications on such architectures are much easy to parallelize because the programmer should not try to reduce accesses to remote memories as such ones do not exist in this case. One way to obtain good parallel performance on UMA SMPs is by not exceeding the bandwidth between processor and memory modules. In order to achieve this, the memory hierarchy has to be used efficiently. Using as much data from cache memories keeps the used memory bandwidth low. The memory controller routes all traffic from 8 memory ports, 4 processor ports, and 2 bridge ports. Since there are 8 memory ports, spreading installed memory modules across all of them can improve performance. Memory modules are installed in two-way interleaving matched pairs, to ensure that the memory port is

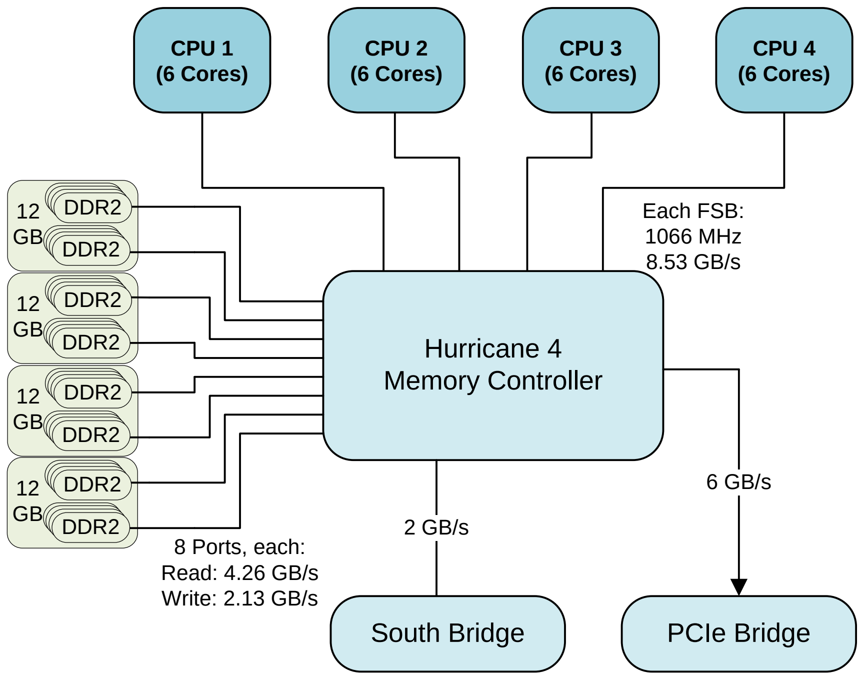

CPU 1 (6 Cores)
CPU 2 (6 Cores)
CPU 3 (6 Cores)
CPU 4 (6 Cores)
Each FSB: 1066 MHz 8.53 GB/s
12 GB
DDR2
Hurricane 4 Memory Controller
6 GB/s
2 GB/s
8 Ports, each: Read: 4.26 GB/s Write: 2.13 GB/s
South Bridge
PCIe Bridge

Figure 5.2: The UMA SMP Shared Memory Platform

fully utilized. With four memory cards installed, and eight memory modules in each card, peak read memory bandwidth is 34.1 GB/s and peak write bandwidth is 17.1 GB/s.

Each Intel Xeon E7450 is a Dunnigton processor, which comes from an older generation than Nehalem processors. The main difference is that Dunnington processors lack Hyper-Threading technology and integrated on chip QPI and memory controller modules. Manufactured on 45 nm process they have a maximal TDP of 90 Watt. Each of the 6 cores, in each processor chip operates at 2.4 GHz and has a 64-bit Instruction Set based on Intel 64 technology.

Another difference is that in Dunnigton processors the L2 cache is shared between a pair of cores and not separated for each core. Each pair shares a 3 MB L2 cache (quite larger than 256 KB L2 cache/core in Nehalem) and all cores share a 16 MB L3 cache (larger than 12 MB L3 cache in Nehalem). This means that a good exploitation of the cache hierarchy can be used to avoid bottlenecks in the memory link. This UMA SMP platform runs on Linux Operating System too. It provides the same programming environment with compilers, profilers and debuggers as the ccNUMA platform. Paraver [134] tool is used for visual and performance analysis. Table 5.2 lists software environment details of the UMA SMP platform.

5.2 Speeding up Synthetic Data Generation

In order to reduce the time needed for synthetic SAR sensor data generation, SDG processing steps are parallelized. SDG is not interesting for SAR on-board processing, but it

Table 5.2: Software Environment on the UMA SMP Platform

	Name	**Version**
Operating System	SUSE Linux Enterprise Server 11	Linux 2.6.32.12-0.7-default x86_64
Compiler (OpenMP)	GNU C Compiler (GCC)	4.3.4
Compiler (OmpSs)	Mercurium Compiler (MCC)	1.3.5.7
Debugger	GNU Debugger (GDB)	7.0-0.4.16
Profiler	GNU Profiler (GPROF)	2.20.0.20100122-0.7.9
FFT Library	FFTW	3.2.2
Performance Monitoring Tool	Paraver	3.99

is parallelized and optimized to reduce execution times and to collect experiences related to the behavior of such processing steps in modern computer and processor architectures. Similar processing steps applying FFTs, complex exponentials, additions and multiplications to two-dimensional data sets can be found also in the SSP stage of 2DSSAR. SDG is optimized because of the fact that it has to run on parallel platforms at least once in order to obtain raw sensor data that is input data for SSP.

5.2.1 SDG Parallelization

In order to speed-up SDG, multithreading for shared memory environments is applied. The work in different processing steps of SDG is distributed over multiple OpenMP threads, which are then scheduled for execution on different cores. The most time consuming steps of SDG are first identified by profiling SDG with $gprof$ [130] profiling tool. It turns out that the step computing the hyperbolic time delay of each reflector placement and integrating synthetic echoes from the transmitted waveform is the most time consuming step in SDG. Around 90 % of the total execution time in SDG is spent in this processing step, which is called the *loop over the reflectors* and is already described in subsection 4.3.1 by Algorithm 4.1 on page 63. 5 % of the time is spent in baseband lowpass signal compression for later processing suitability. 3 % of the time is spent in baseband reference signal transformation into Doppler (spatial frequency) domain. This conversion generates the fast time filter, which is later used in IR. The rest of the time is spent on parameter preparation and configuration.

As the most time-consuming step, the *loop over the reflectors* is the first step to be parallelized by distributing iterations of the first *for* loop over OpenMP threads. This is a straight-forward task to do because iterations are completely independent, i.e. there is no data dependency between them. Variables containing loop counters and the number of iterations are specified as private for each thread. This means that each thread will have a private copy of these variables allocated on the stack, which in cache coherent platforms is supposed to be in a memory local to the corresponding thread. This reduces multiple remote accesses present in the original scenario. Signal compression to a baseband lowpass signal involves one compression nested loop with independent iterations in range (n) and cross-range (m_c), one *fftshift* operation on two-dimensional data, and two compression loops with independent iterations in range (n).

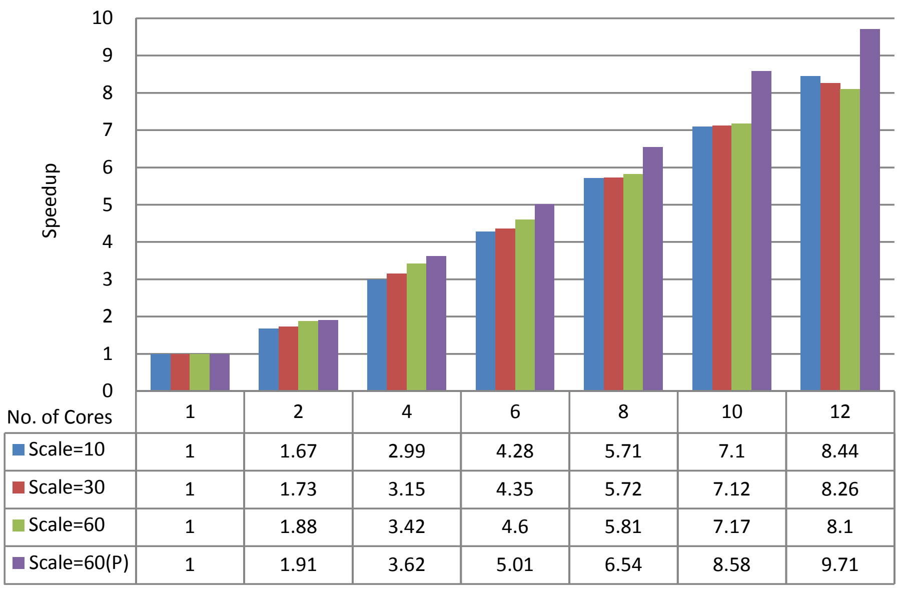
Similar to the *loop over the reflectors*, for the parallelization of compression loops, loop iterations are distributed over OpenMP threads with specified private variables. The two-dimensional *fftshift* interchanges entries in quadrants 1 and 3, and 2 and 4. In the sequential version, *2D-fftshift* is implemented by nested *for* loops that iterate till the half of range (n) and cross-range (m_c). In each iteration, an entry from quadrant 1 is exchanged with an entry from quadrant 3 and an entry from quadrant 2 is exchanged with an entry from quadrant 4. This is achieved by using two temporal variables that store exchange values. In the parallelized version, each thread executes some iterations in range dimension and all nested iterations in cross-range dimension. It is very important to make sure that temporal variables are private to each thread in order to provide a correct parallelized *2D-fftshift* operation. Other variables are specified private as in the previous loop parallelization.

The last processing step that is parallelized in SDG is the transformation of the baseband reference signal into spatial frequency domain, which is implemented by a batch of row-wise one-dimensional FFTs. In the parallel version, multithreaded FFTW routines for shared memory threads are used. In order to use such routines, the FFTW library should be installed with enabled threading and FFTW threads should be initialized before calling any multithreaded routine. The number of threads used by these routines is configurable by the programmer, but on the other side is also tunable from the library itself. This means that the library functions will always auto-tune the number of threads depending on the number of points in each transform.

Each FFTW transform is implemented in three stages called plan creation, plan execution, and plan destruction. Plan creation and destruction are not thread-safe routines, i.e. such routines cannot be called from multiple threads. The main restriction of FFTW is that it is not safe to create plans in parallel because plan-creation functions share trigonometric tables. All plans have to be created from a single thread. Only the plan execution can take place in parallel. In the current implementation of FFT in SDG, only one complex-to-complex one-dimensional plan is created in sequential. The execution is also called in sequential for each individual row, but the internal execution of the plan for each row takes place in parallel.

5.2.2 Obtained SDG Speedup

Figure 5.3 illustrates the speedup obtained when running SDG on the ccNUMA platform. Speedup is measured for different data scales, from a small scale data set (Scale = 10) to a large scale one (Scale = 60). Obtained speedup is almost linear for small and large scale data generation. There is a difference in SDG speedup obtained for different data scales depending on the number of threads used. When using less than 10 threads, large scale data generation speedup is higher, but when using more than 10 threads small scale data generation speedup gets higher. This is due to data transfers, the number of which increases if the data generation scale increases. Since there are also some application parts that cannot be parallelized, like parts of code for reading and writing from and to I/Os, there is a difference between the speedup measured from the whole SDG (Scale=60) and the speedup measured when monitoring only the parallel region in the computational intense part of the application (Scale=60(P)).



No. of Cores	1	2	4	6	8	10	12
🟦 Scale=10	1	1.67	2.99	4.28	5.71	7.1	8.44
🟥 Scale=30	1	1.73	3.15	4.35	5.72	7.12	8.26
🟩 Scale=60	1	1.88	3.42	4.6	5.81	7.17	8.1
🟪 Scale=60(P)	1	1.91	3.62	5.01	6.54	8.58	9.71

Figure 5.3: SDG Speedup on the ccNUMA Platform

5.3 SAR Sensor Processing on the ccNUMA Platform

Just to recall, SSP is the second stage of 2DSSAR following the data generation stage. SSP initially reads generated raw data from disk and then applies an image reconstruction algorithm to form the viewable image. The resulting image is then written to disk for further verification purposes. This section starts with some profiling results of SSP that help in identifying the most computational intensive and time consuming parts of this stage. Then, parallelization techniques and respective performance results are discussed.

5.3.1 Profiling SSP Stage

The GNU profiler *GPROF* tool is used to profile SSP. Table 5.3 shows profiling results in terms of elapsed execution time and the respective percentage compared to the total execution time. These results are obtained when running the application for large scale image reconstruction (Scale=60), which is the largest one used in benchmarking. The *SSP Sequential Profiling* column shows results of the complete SSP stage by identifying 3 main substages. As shown, the IR substage takes 94 % of the total execution time and the rest (6 %) of the time is spent on reading and writing to disk. The interesting point is that I/O operations cannot be parallelized. As a consequence, parallelization techniques are applied only to IR. This is the reason that IR is also profiled separately and results are shown in the *IR Sequential Profiling* column.

Profiling is very important because it identifies application parts that take more time. These parts should be the first to be considered for parallelization. It also helps in defining the effort needed to parallelize each individual step. The most time consuming step of IR is step 10 (Interpolation Loop) that takes 69 % of the total execution time. This means that it is very critical to efficiently parallelize this step.

In terms of execution time, Interpolation Loop is followed by step 11 that applies a two-dimensional FFT. This step is originally implemented by two batches of one-dimensional FFTs, one row-wise and one column-wise. Because of poor performance on the second batch of one-dimensional column-wise FFTs, it is then implemented with an intermediate transposition stage between batches. This results in much better performance since both FFT batches take place row-wise. The next time consuming steps are step 6 and step 8 that apply element-wise complex exponential, multiplication and accumulation operations to a large two-dimensional data set. The rest of the steps all together take around 10 % of the execution time. In order to obtain very good parallel performance, these steps have to be parallelized too.

5.3.2 Expected SSP Speedup

SSP contains some processing steps that are suitable for parallelization and others that are not. By using multiple processors, time spent in parallelized parts can be reduced, but time spent on sequential parts remains the same. Eventually, execution time is impacted by the time taken to compute the sequential portion, which puts an upper limit on the theoretical expected speedup. This effect is known as Amdahl's law and can be formulated as:

$$Expected\,Speedup = \frac{1}{[F_p/P + (1 - F_p)]}, \tag{5.1}$$

where F_p is the parallel part of code and P is the number of processors (or processor cores). The part of SSP that can be parallelized is the IR substage, which takes 94 % ($F_p = 0.94$) of the total elapsed time in sequential execution. The maximal expected speedup on the ccNUMA platform with 12 processor cores is:

$$Expected\,SSP\,Speedup = \frac{1}{[0.94/12 + (1 - 0.94)]} = 7.22 \tag{5.2}$$

5.3.3 IR Parallelization and Optimization

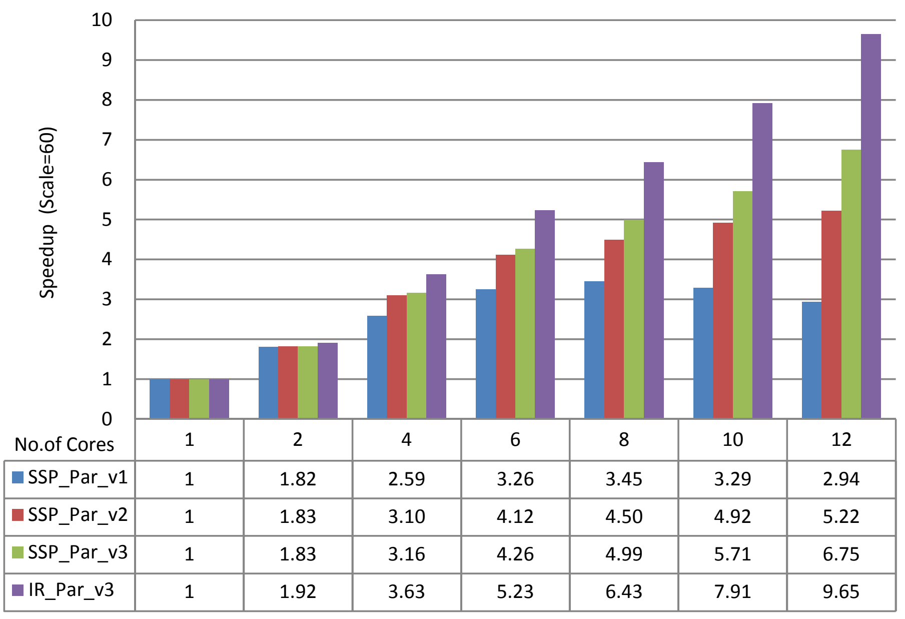
In order to obtain the expected SSP speedup on the ccNUMA platform, IR is parallelized for shared memory multiprocessing. Three incremental versions are implemented during the initial parallelization process. In the first version (SSP_Par_v1), only iterative loops are parallelized. As the most time consuming step, Interpolation Loop is the first one to be parallelized. It is then followed by steps 3, 6, and 8 (in Table 5.3) that apply compression and decompression loops. *2D-fftshift* and *transpose* operations are also parallelized in the first version. As shown in Figure 5.4, the maximal speedup for the first parallel version is only 3.45, which is obtained using 8 threads (cores). This speedup chart illustrates only the speedup obtained for large scale image reconstruction (Scale = 60). Fourier transform steps are intentionally left sequential in this first version because it is very important to investigate the impact that their parallelization has on the overall performance.

Table 5.3: SSP Profiling Results

	Processing Step	**SSP Sequential Profiling**	**IR Sequential Profiling**		**Type of Computation**
1	Read raw SAR sensor data from disk	20 Seconds (2 %)	Seconds	Percent	
2	Filter the echoed signal in fast-time	Image Reconstruction (IR) 926 Seconds (94 %)	12.6	1.31	FFT and Tranposition
3	Compress the filtered signal along slow-time		12.8	1.33	CEXP and MAC
4	Narrow-bandwidth polar format reconstruction along slow-time		4.6	0.5	FFT
5	Zero pad the spatial frequency domain's compressed signal		5	0.54	Padding
6	Decompress the zero-padded signal in slow time		47	5.2	CEXP and MAC
7	Digitally spotlight the SAR signal spectrum		24.6	2.5	FFT
8	Generate the reference signal's complex conjugate		47	5.2	CEXP and MAC
9	Circumvent edge processing effects		33	3.5	*2D-fftshift*
10	Interpolation Loop		639	69	Mapping and Reduction
11	Transform signal from Doppler domain to spatial domain		93	10	FFT and Transposition
12	Transform the spatial domain signal into a viewable image		11	1.15	CABS
13	Write image to disk	40 Seconds (4 %)			

In the second version (SSP_Par_v2), all steps that apply Fourier transforms (steps 2, 4, 7, and 11 in Table 5.3) are parallelized by using multithreaded FFTW routines. The desired number of threads can be given as an argument to plan creation routines, but the library will auto-tune this argument depending on the number of points in the transform. In some cases this might be helpful, but in some other scenarios it might degrade the performance.

In all steps that involve Fourier transforms, one-dimensional FFTs are applied row-wise. As the number of elements in each row is not so large, the execution of each transform does not use more that 4 threads, even in cases when more threads are desired. The usage of multithreaded FFTW routines improves the speedup factor of SSP to 5.2, which is still lower than the expected SSP speedup of 7.22.



No.of Cores	1	2	4	6	8	10	12
🟦 SSP_Par_v1	1	1.82	2.59	3.26	3.45	3.29	2.94
🟥 SSP_Par_v2	1	1.83	3.10	4.12	4.50	4.92	5.22
🟩 SSP_Par_v3	1	1.83	3.16	4.26	4.99	5.71	6.75
🟪 IR_Par_v3	1	1.92	3.63	5.23	6.43	7.91	9.65

Figure 5.4: SSP/IR on the ccNUMA Platform

In order to execute faster the steps involving Fourier transforms, an approach that parallelizes a complete step is taken. As each step is composed of multiple one-dimensional FFTs, one for each row, the loop that iterates through these rows is parallelized. To achieve this, a number of plans equal to the number of threads have to be created in sequential before entering the parallel region. It is very important to create these plans in sequential since plan creation routines are not thread-safe. Having one plan for each thread, loop iterations can be distributed among threads that execute plans privately row-by-row over the respective chunk of rows. Implemented in the third version (SSP/IR_Par_v3), this approach provides a coarse-grained parallelism that does not depend on the internal implementation of FFTW library. The only overhead is the creation of multiple plans, which might take some time in sequential.

When created, FFTW plans have a flag argument that is in most cases either *FFTW-MEASURE* or *FFTW-ESTIMATE*. *FFTW-MEASURE* instructs FFTW to run and measure the execution time of several FFTs in order to find the best way to compute the transform. This process takes some time depending on the platform and on the transform size. On the contrary, *FFTW-ESTIMATE* does not run any computation, but just builds a reasonable plan that is probably sub-optimal. In the first and second version, *FFTW-MEASURE* is used as the program creates only one plan per each step and performs many transforms of the same size. Since in the third version multiple plans are created in sequential for each step, *FFTW-ESTIMATE* is used to reduce time needed for plan creations. This increases a little bit time needed for plan execution, but as plans execute in parallel the overhead is not considerable. The obtained speedup of 6.75 is quite near to 7.22 as it represents 93.5 % of the expected SSP speedup.

Other optimizations, included in the third version (SSP/IR_Par_v3) are:

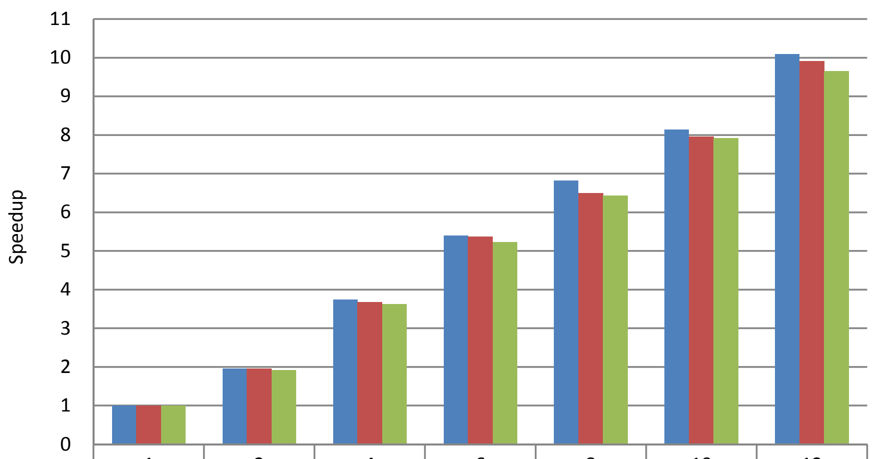
1. *Thread Pinning* and *First Touch Policy.* OpenMP affinity environment variable *GOMP-CPU-AFFINITY* is used to pin each thread to a specific core to prevent their migration from one core to another. Thread migration causes a lot of overheads in the form of scheduling overhead, context-switching overhead and cache cost. *First Touch* data placement policy improves data locality on a ccNUMA system by having each thread allocating data on the memory local to the processor core it starts executing on. Combining *Thread Pinning* and *First Touch* builds a suitable strategy for programs, in which updates to a given data element are typically performed by the same thread throughout the computation.

2. *Static/Dynamic Scheduling.* To control the manner in which loop iterations are distributed over threads, the scheduling mechanism is set to static for loops with regular workloads and to dynamic for loops with non-regular ones. Things are harder if the data access pattern is not uniform throughout the code, especially in the presence of conditional jumps. In such loops that create workload imbalance dynamic, scheduling is used.

3. *Private Variables.* The last but not the least important optimization is to increase the number of private variables for each thread. The use of private data is beneficial on a ccNUMA system. Typically, private data is allocated on the stack, which in a cache coherent NUMA aware OpenMP implementation is supposed to be in a memory local to the core the thread is executing on. This configuration cannot always be ensured with shared data, even when *First Touch* policy is appropriately used. When shared data is fetched from remote memory, the data cache local to the processor will buffer it, so that subsequent accesses can be fast. However, data may be displaced from cache in order to make place for another block of data. If so, another relatively expensive transfer from remote memory will occur next time it is needed.

As the most interesting part of SSP, IR substage contains processing steps that in the future will probably execute on-board of spacecrafts as part of a real space application. Since it is relevant to investigate the impact of parallelizing such steps, separate measurements are carried out only for the IR substage. All investigations in the rest of this dissertation apply to IR substage only.

As shown in Figure 5.4, the IR speedup obtained is around 9.7 out of the ideal theoretical speedup of 12. The time needed to reconstruct a large scale image on the ccNUMA platform is reduced from 15 minutes and 26 seconds to 1 minute and 36 seconds. The obtained speedup proves that IR and in general SAR processing applications scale very well in ccNUMA architectures with up to 12 cores. The almost linear speedup increase makes it possible to speculate that performance improvements might be obtained even in ccNUMA platforms with more cores. Figure 5.5 depicts results obtained on the ccNUMA platform when running image reconstruction for different image scales. Differences in the obtained speedups come from the fact that the amount of computations and communications increases while increasing image scale.



No. of Cores	1	2	4	6	8	10	12
🟦 Scale=10	1	1.97	3.74	5.40	6.82	8.14	10.09
🟥 Scale=30	1	1.97	3.68	5.37	6.50	7.96	9.91
🟩 Scale=60	1	1.92	3.63	5.23	6.43	7.91	9.65

Figure 5.5: Different Scale IR (IR_Par_v3) on the ccNUMA Platform

5.3.4 Data Integrity in the Parallel Interpolation Loop

As it is discussed in subsection 4.3.2 on page 64, the interpolation process converts data from polar coordinates to rectangular ones. This is achieved by mapping each row in the input image to several rows in the output one. In the specific implementation, one input row maps to eight output ones. Since the output image is not that much larger than the input one, this 1-to-8 mapping creates data overlapping. In sequential execution, this overlapping is not a problem for data integrity as overlapped rows accumulate respective values in each iteration.

In multithreaded execution each thread takes as input a chunk of rows from the input image and writes results into a chunk of rows in the output image. For two consecutive threads, there is an overlapping between their respective chunks of rows in the shared output image. Actually, some last rows of the first thread overlap with some first rows of the second thread. Since each thread initially starts processing the first rows of the respective chunk this overlapping does not corrupt the resulting image. When the first thread has to update the last rows of its chunk, the second thread has already finished updating the first rows of its chunk.

While increasing image scale, the number of rows allocated to each thread increases, giving so each thread more time to finish processing the first rows of his chunk before the other thread has to update them too. This is the reason that results are correct every time IR is run in a multithreaded environment. Nevertheless, no one can prove that results will always be correct. There is always the danger of a data race condition if thread execution does not go as predicted. In such a case, overlapped image regions might get corrupted.

As the image might not be completely corrupted, this approach might be accepted in applications that need images as fast as possible even though some regions are corrupted.

Sometimes it is better to have an initial output, from which one can extrapolate the complete or only some information, than to wait longer for a complete correct image.

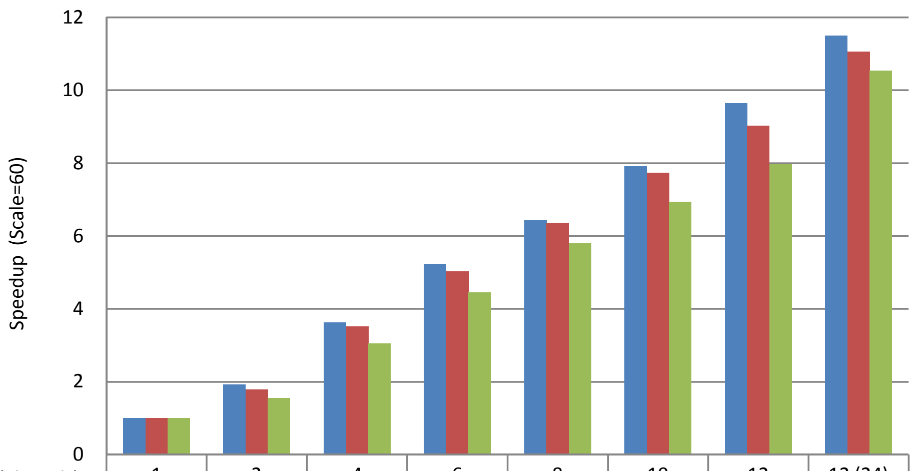
In cases when the application has to output 100 % correct images, synchronization and locking techniques are used in order to protect data integrity. For the concrete situation in which two threads might write on the same memory location, two approaches are taken. The first one is to apply a Replication and Reduction (R&R) technique. The output image is replicated for each thread that now updates a private copy of the data. When all threads finish processing, the resulting image is constructed by reducing replicated images. This reduction can be executed in parallel if thread workload distribution is done on row-basis and not on complete images. It might not be the most efficient parallelization due to the presence of cache misses, but it is going to execute faster than the sequential one.

The R&R technique, which is implemented in the fourth IR parallel version (IR_Par_v4) uses quite a lot more virtual memory and just a little bit more physical memory than version 3. This happens because for each single thread a complete image is allocated, but only overlapped regions are initialized from each respective thread. For large scale image reconstruction, the amount of virtual memory needed might exceed the available memory space, especially when using many threads. This makes this parallel version of IR suitable to run on platforms with abundant memory resources or with virtual memory swapping enabled.

The second approach is to update memory locations by using atomic operations that are performed without interference from any other thread. In hardware, the core executing the thread simultaneously reads a location and writes it in the same bus operation preventing any other thread (core) from writing or reading memory until the operation is complete. OpenMP provides the *atomic* construct, which enables threads to update shared data without interference on hardware platforms that support atomic operations.

In order to apply atomic operations to the updating instruction in Interpolation Loop, data being used in this instruction has to be first converted from complex floating-point data to floating-point data because OpenMP does not support atomic operations on complex data types. Converting from complex to floating-point requires just a little bit additional memory space to store converted values. This approach is implemented in the fifth IR parallel version (IR_Par_v5), which is suitable to run even on platforms with no virtual memory swapping enabled.

As shown in Figure 5.6, version 4 performs better than version 5 because the reduction overhead is lower than the one coming from atomic operations combined with complex-to-float conversions. In simultaneous multithreading mode (with 24 threads), the performance gap between versions 4 and 5 becomes smaller but still version 4 performs better. This makes version 4 and the R&R technique more suitable for parallel platforms with a small number of cores. Taking into consideration that the reduction overhead becomes larger and memory consumption keeps rising while increasing the number of threads, version 4 is not expected to scale at the same level as version 5, which is predicted to perform better on parallel platforms with more cores.



No. of Cores (Threads)	1	2	4	6	8	10	12	12 (24)
🟦 IR_Par_v3	1	1.92	3.63	5.23	6.43	7.91	9.65	11.49
🟥 IR_Par_v4 (R&R)	1	1.78	3.51	5.02	6.36	7.74	9.03	11.06
🟩 IR_Par_v5 (Atomic)	1	1.55	3.05	4.45	5.81	6.94	7.98	10.54

Figure 5.6: IR with Data Integrity Optimizations on the ccNUMA platform

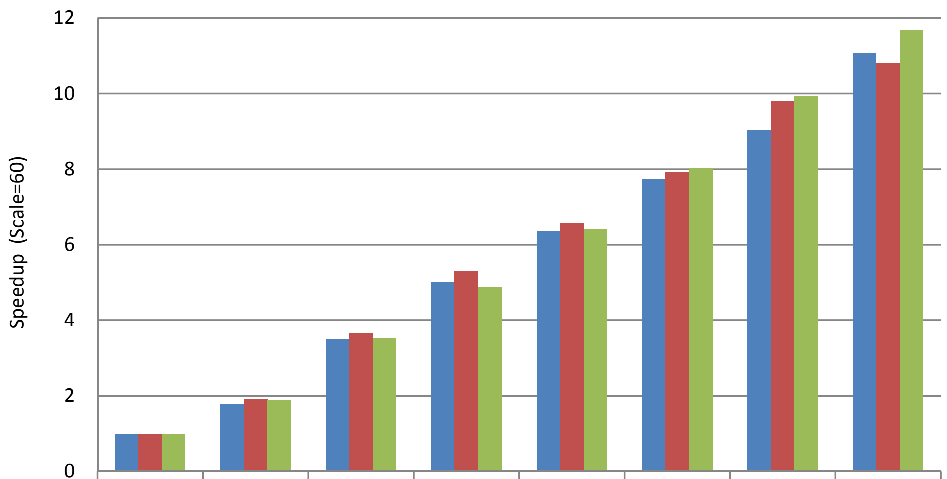
5.3.5 Distributed Memory Programming on the ccNUMA Platform

Better parallel performance can be obtained if remote accesses in the ccNUMA platform are reduced. Optimizations like *Thread Pinning, First Touch Policy* and *Private Variables* help in keeping remote memory accesses low. Another way to make sure that no remote memory access occurs is to use a distributed memory programming model on the shared memory system. Such programming models like MPI provide the possibility to have processes with its own local data that resides on the local memory associated with the core where it runs. The communication between processes takes place through message passing. Distributed and shared memory programming models can be combined in a shared memory environment to build a hybrid programming model.

The most well-known hybrid programming model is built by combining MPI and OpenMP. In such a model, MPI is used to create and schedule one process for each multi-core processor chip (separate socket) and to take care of the communication between different processes in a multi-chip NUMA platform. The complementary OpenMP distributes the internal work of each process to a number of threads that is usually equal to the number of cores in the multi-core chip. Such an approach minimizes remote memory accesses, making the application more scalable by message passing, and more efficient by the multithreaded parallel execution of OpenMP.

Figure 5.7 illustrates obtained results when using OpenMP, MPI and hybrid MPI + OpenMP programming to parallelize IR for the ccNUMA platform. Replication and reduction is used to avoid thread data conflicts in the interpolation loop. Recall that the 12-core ccNUMA platform is composed of two multi-core chips, with 6 cores each. For the pure OpenMP (version 4) and MPI (version 6) implementations the respective number of threads and processes equals the number of cores used in the ccNUMA platform. In

SMT mode, 24 threads/processes are used. For the hybrid (version 7) implementation these numbers differ in the following way. Up to 6 cores, one MPI process with up to 6 OpenMP threads is used. In 8, 10, and 12 cores, two MPI processes with respectively 4, 5, and 6 OpenMP threads are used. In SMT mode, 2 MPI processes are created and in each of them the workload is distributed on 12 OpenMP threads.



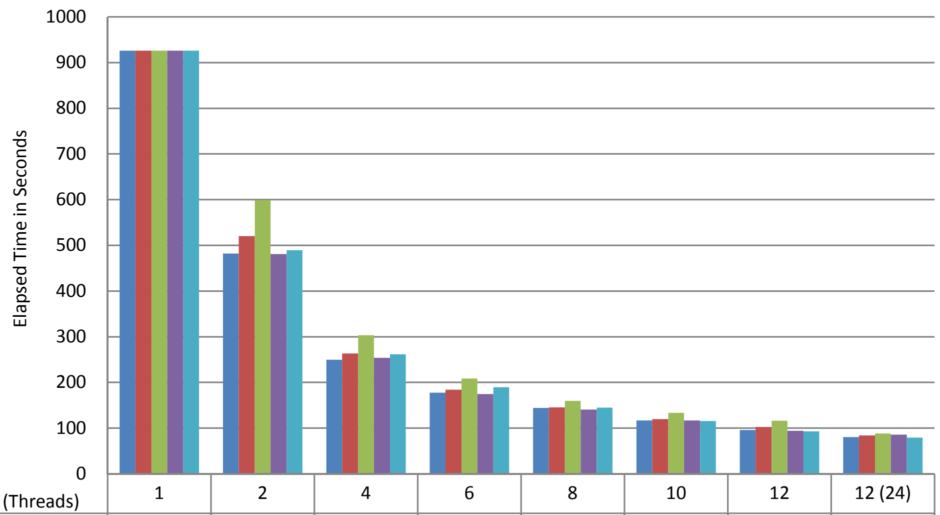
No. of Cores (Threads)	1	2	4	6	8	10	12	12 (24)
🟦 IR_Par_v4 (OpenMP R&R)	1	1.78	3.51	5.02	6.36	7.74	9.03	11.06
🟥 IR_Par_v6 (MPI R&R)	1	1.92	3.65	5.30	6.57	7.94	9.81	10.81
🟩 IR_Par_v7 (MPI+OpenMP)	1	1.89	3.54	4.88	6.40	8.02	9.94	11.69

Figure 5.7: IR with Different Programming Models on the ccNUMA platform

Regarding performance, up to 8 cores, MPI provides a slightly higher speedup than OpenMP and hybrid implementations. The poor performance of the hybrid implementation is related to the additional overhead of process initialization and thread scheduling at the same time. When using more than 8 cores, the work is distributed among two processor chips. In this scenario, MPI and hybrid implementations slightly outperform the OpenMP one because they help in reducing remote memory accesses in the multi-chip platform. While increasing the number of cores, the hybrid implementation provides better performance and higher flexibility in workload distribution compared to the MPI one.

In SMT mode, OpenMP and hybrid implementations outperform the MPI one due to the overall process creation and communication overhead. Figure 5.8 shows the time needed for large scale image reconstruction on the ccNUMA platform depending on the number of cores being used and on the implemented parallel version. It starts with the third parallel version of IR, which is the first full parallel version implemented.

Just to summarize, Table 5.4 lists all optimizations applied to SSP and IR in each implementation that is run on the ccNUMA platform. Next section discusses the implementation of some of them for a UMA SMP platform and extends them with new versions implemented with OmpSs programming model.



No. of Cores (Threads)	1	2	4	6	8	10	12	12 (24)
🟦 IR_Par_v3 (OpenMP)	926.1	482.2	249.8	177.3	144.5	117.1	95.9	80.6
🟥 IR_Par_v4 (OpenMP R&R)	926.1	520.0	263.6	184.3	145.6	119.7	102.6	83.7
🟩 IR_Par_v5 (OpenMP Atomic)	926.1	599.1	303.6	208.3	159.5	133.5	116.1	87.9
🟪 IR_Par_v6 (MPI R&R)	926.1	481.2	253.8	174.7	140.9	116.7	94.4	85.7
🟦 IR_Par_v7 (MPI+OpenMP)	926.1	489.2	261.8	189.8	144.6	115.5	93.2	79.2

Figure 5.8: IR Elapsed Time on the ccNUMA platform

Table 5.4: Optimizations applied on each parallel version of SSP (IR)

Version	Optimization
SSP_Par_v1	Iterative loops in steps 3, 5, 6, 8, 9, 10, and 12 are parallelized in OpenMP threads.
SSP_Par_v2	FFT transforms in steps 2, 4, 7, and 11 are parallelized by using multithreaded FFTW routines.
SSP_Par_v3	Loops of FFT transforms are parallelized in OpenMP threads. Other optimizations applied in this version are: Thread Pinning, First Touch Policy, Static/Dynamic Thread Scheduling, and increasing the number of private variables.
IR_Par_v3	Only IR substage is considered in measurements. No new optimizations are applied.
IR_Par_v4	Replication and Reduction (R&R) technique is applied to step 10 (Interpolation Loop) to improve data integrity during parallel execution.
IR_Par_v5	Atomic operations are used is step 10 instead of R&R. This version requires less memory during execution.
IR_Par_v6	MPI is used to parallelize all IR steps. A number of processes equal to the number of cores are created.
IR_Par_v7	Hybrid MPI + OpenMP programming is used to parallelize IR. One MPI process is created and 6 OpenMP threads are forked for each socket in the platform.

5.4 Image Reconstruction on the UMA SMP Platform

In the frame of benchmarking shared memory systems, a UMA SMP platform is also benchmarked with IR. Compared to the NUMA platform discussed in the previous section, the UMA one has 24 cores that get access to memory modules through the same memory controller. This provides to all processor cores in the system the same latency and bandwidth to access memory. In such a scenario, no remote memory accesses with high costs occur, but the problem stands in the shared memory bandwidth that should not be overwhelmed in order to avoid bottlenecks that degrade performance.

Two different programming models for shared memory environments are used in this platform; the traditional OpenMP and the task based OmpSs [19]. As already discussed in subsection 2.3.2 on page 16, OmpSs uses OpenMP pragmas, a source-to-source translator and a runtime system that schedules tasks while detecting dependencies between them. Most optimized OpenMP parallel versions of IR (versions 3, 4, and 5) are initially ported to the UMA SMP platform. The same versions are then ported to OmpSs programming model so that performance improvements can be obtained. Corresponding OmpSs parallel versions of IR are listed in Table 5.5.

Table 5.5: IR OmpSs Parallel Versions

Version	**Optimization**
IR_Par_v8	IR_Par_v3 is ported to OmpSs. No data protection in Interpolation Loop
IR_Par_v9	IR_Par_v4 is ported to OmpSs. Replication and Reduction is used in Interpolation Loop
IR_Par_v10	IR_Par_v5 is ported to OmpSs. Atomic Operations are used in Interpolation Loop

5.4.1 Porting Image Reconstruction to OmpSs

This section describes the steps taken to port IR to OmpSs. It is easier to start with the already parallelized version of IR, which is parallelized using OpenMP pragmas. To port IR to OmpSs, OpenMP *parallel for* directives outside the loops are removed, and OpenMP *task* construct is inserted in outer loops for all processing steps of IR.

Tasks have been initially defined such that each task processes only one row of the two-dimensional data. This is not a problem for small scale image reconstruction, but for large scales the number of rows increases up to 20.000. Creating lots of small tasks adds too much overhead. As a solution, loops are blocked to introduce coarser grained tasks. To create a number of tasks equal to the number of threads, the number of rows is divided by the number of threads to get the size of each chunk. In this way, each task will operate on chunks of *size = number of rows / number of threads.* In some steps with evident load imbalance, like Interpolation Loop, it is better to define more tasks than available threads for finer task granularity and better load balancing.

Taskwait directives are initially used between all IR steps. *Input*, *output,* and *inout* clauses are used to express data directions that help in identifying dependencies between tasks in different steps of IR. To improve the performance of the SAR application *taskwait* directives are removed between tasks with satisfied data dependencies. In such steps, data specified as output for one step has the same size and is accessed in the same pattern (for example: row-by-row element-wise) as data specified as input for the next task. The intention is to have as many consecutive tasks with no synchronization points in between. A step reordering in IR is needed to put together as many tasks with similar input and output data sets.

However, not all steps can be moved. Such steps are the ones that apply *transpose* and *2D-fftshift* operations. They have to be applied at a specific order with other steps. Another problem is that it is not possible to remove *taskwait* directives in tasks responsible for such operations. Transposition changes data layout, creating such a scenario where one step before transposition and one step after it, have different access patterns over data. This means that only one of the *taskwait* directives can be removed, either the one before transposition or the one after it. Which one, depends on whether reading or writing is carried out row-wise.

Transposition is used twice, once in the first and once in the last step of IR. In the first step, transposition is applied by reading column-wise and writing row-wise, which is also the access pattern of some steps following transposition. Therefore, the *taskwait* directive after this transposition can be removed. However, in the last step, the *taskwait* directive before transposition is removed and transposition is applied by reading row-wise and writing column-wise. This allows transposition steps to be part of the group with consecutive tasks that run without any synchronization point in between.

Recall that a *2D-fftshift* operation swaps the first quadrant of the image with the third one. At the same time, it swaps the second with the fourth one. Initially, *2D-fftshift* is implemented as an in-place operation. Quadrants exchange their corresponding locations by using temporal variables. Since each task operates on a chunk of rows, the swapping logic is like the one depicted in Figure 5.9. This picture shows that *n* half rows with *m/2* elements from quadrant A will swap values with *n* half rows from quadrant D, but at the same time the other half of the same rows will be swapped. In this way a chunk of rows has to be specified as input and output at the same time for the same task in OmpSs.

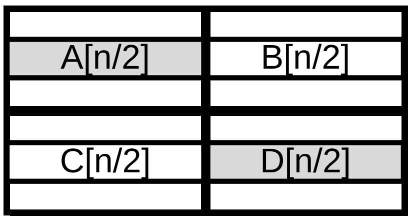


Figure 5.9: In-place two-dimensional *fftshift*

Specifying a location as input and output at the same time is a bit tricky for OmpSs. This is the reason that some steps get serialized during execution. In some runs, results are even incorrect when removing *taskwait* directives before and after *2D-fftshift* steps. The reason for incorrect results is again the specification of data locations as input and output at the same time. In order to adapt this step with other steps of IR, the in-place version of *2D-fftshift* is not used anymore. Two different arrays are allocated, one as input and one as output (out-of-place *2D-fftshift*). The access pattern of the input array is identical to the ones in steps before *2D-fftshift*, but this is not true for the output array. This is the reason that the *taskwait* directive before *2D-fftshift* is removed, but not the one after it.

Another way to deal with these problems (in *transposition* and *2D-fftshift* steps) in OmpSs is to specify disjoint memory regions as input and output, but this type of specification requires the data to be aligned. The alignment should not be a problem for small scale image reconstruction, but it is a problem for the large scale one because it is hard to find a memory alignment utility for huge chunks of memory and even if possible it will increase the size of the allocated memory. As a compromise between performance and memory consumption it is decided to leave some *taskwait* directives in *transposition* and *2D-fftshift* steps.

5.4.2 Results Obtained on the UMA SMP Platform

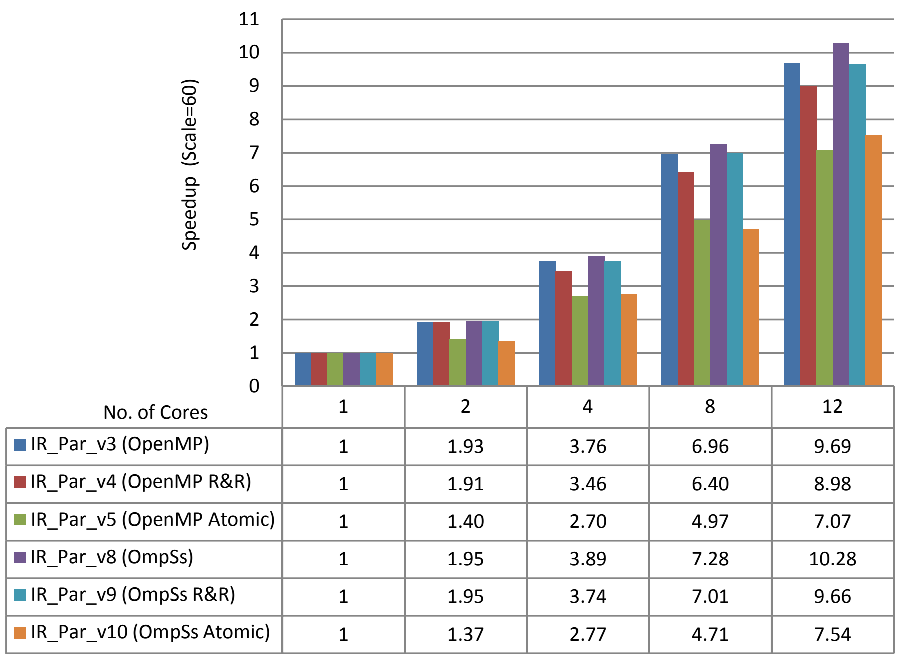
Six different parallel versions of IR are used to benchmark the UMA SMP platform. Three of them (IR OpenMP version 3, 4, and 5) are already used to benchmark the ccNUMA platform. They are ported to the UMA SMP platform with no additional optimizations. Recall that these three versions apply different data protection techniques in Interpolation Loop. Each of them is ported to OmpSs by applying the steps discussed in the previous subsection.

Large scale image reconstruction obtained speedups with up to 12 cores are shown in Figure 5.10. Similar behavior is obtained with both programming models (OpenMP and OmpSs). Plain versions 3 and 8 are the fastest, followed by replication and reduction versions 4 and 9. Versions 5 and 10 that apply atomic operations are the slowest. Compared to OpenMP versions, all OmpSs versions perform a bit faster. Considering IR_Par_v9 (OmpSs R&R) as the most optimal parallel version for up to 12 cores, the time needed for large scale image reconstruction is reduced from 23 minutes and 20 seconds to 2 minutes and 22 seconds.

As the amount of data being exchanged between processor cores and main memory increases, the available bandwidth gets easily saturated on UMA SMPs. With OpenMP, threads have to wait for data to become available. Some threads finish their work faster because they got their data faster, but at a synchronization point they will have to wait for other threads to finish their work.

The task-based approach of OmpSs minimizes waiting times among threads by having tasks with clear data dependencies consequently scheduled on threads. This means that as soon as a task from one processing step finishes his work, the consecutive task from the next step is started without waiting for other tasks to finish. The higher the number of consecutive tasks that run without synchronization points, the faster the application will execute. Increasing the number of tasks in each step increases flexibility and improves workload balance, but if there are not so many computations, the task creation overhead impacts overall performance. This explains the increased performance when using OmpSs.

When using more than 12 cores on the UMA SMP platform, not all versions can be executed. Replication and reduction versions require additional memory (especially virtual memory) for each additional thread (core). The available main memory on the UMA SMP



No. of Cores	1	2	4	8	12
🟦 IR_Par_v3 (OpenMP)	1	1.93	3.76	6.96	9.69
🟥 IR_Par_v4 (OpenMP R&R)	1	1.91	3.46	6.40	8.98
🟩 IR_Par_v5 (OpenMP Atomic)	1	1.40	2.70	4.97	7.07
🟪 IR_Par_v8 (OmpSs)	1	1.95	3.89	7.28	10.28
🟦 IR_Par_v9 (OmpSs R&R)	1	1.95	3.74	7.01	9.66
🟧 IR_Par_v10 (OmpSs Atomic)	1	1.37	2.77	4.71	7.54

Figure 5.10: OpenMP/OmpSs IR on 12 Cores of the UMA SMP Platform

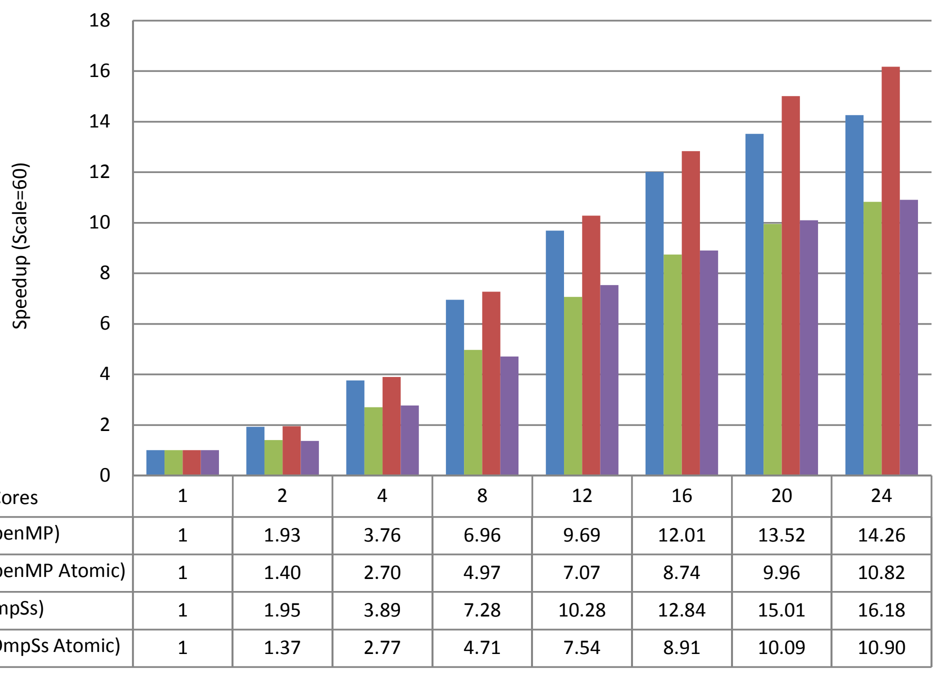
platform can accommodate data for this version with up to 12 threads (cores). Parallelism with up to 24 cores can be exploited only by other versions. The speedup obtained up to 24 cores with OpenMP and OmpSs plain and atomic versions is shown in Figure 5.11. Even up to 24 cores, OmpSs versions are faster than corresponding OpenMP ones.

Atomic operations are slower in both programming models compared to plain versions. However, speedup keeps increasing up to 24 cores. Furthermore, atomic operations (versions 5 and 10) on 24 cores perform better than replication and reduction (versions 4 and 9) on 12 cores, but they also consume more resources decreasing efficiency. With OmpSs Atomic IR (IR_Par_v10), the time needed for large scale image reconstruction on 24 cores is reduced to 2 minutes and 8 seconds, whereas with OmpSs IR (IR_Par_v8) to 1 minute and 27 seconds.

5.4.3 Image Reconstruction on Simulated Multiprocessor Machine

Different techniques applied for correct execution in Interpolation Loop of IR give different versions with different characteristics, especially when computing in parallel. We have seen that replication and reduction is not suitable for parallel platforms with limited memory resources. We have also seen that atomic operations do not increase memory requirements, but impact performance directly. Up to 12 cores, atomic operations perform around 20 % slower than replication and reduction. Taking into consideration that reduction overhead increases more than atomic overhead when increasing the number of cores (threads) being used, it is expected that R&R versions of IR show weaker scalability than Atomic IR versions.

Currently it is impossible to prove this on a real hardware platform as the number of cores



No. of Cores	1	2	4	8	12	16	20	24
🟦 IR_Par_v3 (OpenMP)	1	1.93	3.76	6.96	9.69	12.01	13.52	14.26
🟩 IR_Par_v5 (OpenMP Atomic)	1	1.40	2.70	4.97	7.07	8.74	9.96	10.82
🟥 IR_Par_v8 (OmpSs)	1	1.95	3.89	7.28	10.28	12.84	15.01	16.18
🟪 IR_Par_v10 (OmpSs Atomic)	1	1.37	2.77	4.71	7.54	8.91	10.09	10.90

Figure 5.11: OpenMP/OmpSs IR on 24 Cores of the UMA SMP Platform

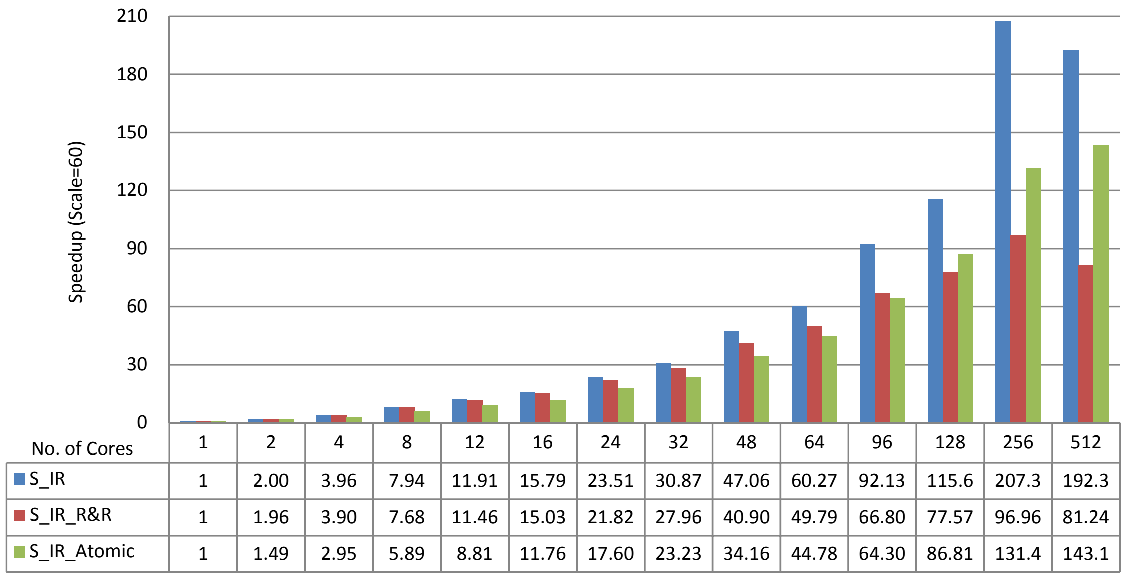
on available shared memory systems is limited to 12 or 24 cores. The only way to prove it is by simulation. The authors in [135], propose a simulator for parallel architectures running multithreaded applications using a trace-driven approach. This simulator is available at BSC computing platforms and is fully compatible with applications using OmpSs for task-based parallelization.

The simulation environment is composed of the simulation engine (TaskSim) and the dynamic component (Nanos++). Traces are collected by running the application in sequential. These traces contain information on tasks and their data dependencies. The simulation starts by running the master thread. The simulator runs sequentially until it reaches the first parallel region in the trace. A parallel region is simulated by feeding available simulated threads with tasks coming out of the trace. For each task, the overhead of its creation, and the runtime of its execution is simulated depending on characteristics of the target simulated machine, instead of depending on characteristics of the machine where the trace is captured.

Since the target simulated machine can be different from the host where the simulation is running, various target machines with various characteristics can be simulated. For the SAR Image Reconstruction application, it is very interesting to see how different versions scale in a simulated multiprocessor machine with more than 24 cores. Different target machines with up to 512 processor cores in a UMA SMP environment are simulated. Simulation results are a bit optimistic. This becomes visible when comparing simulated and real results obtained with up to 24 cores.

Nevertheless, the purpose of the simulation is to analyze the behavior of different versions of IR, especially R&R and Atomic versions, on the simulated machine. As shown in

Figure 5.12, R&R IR scales up to 48 cores. After that, R&R IR performance decreases while Atomic IR performance keeps scaling. When simulating a target machine with more than 128 cores, Atomic IR performs better that R&R IR. In general, performance improvements are visible up to 256 cores, but in real machines this number is expected to be lower.



No. of Cores	1	2	4	8	12	16	24	32	48	64	96	128	256	512
🟦 S_IR	1	2.00	3.96	7.94	11.91	15.79	23.51	30.87	47.06	60.27	92.13	115.6	207.3	192.3
🟥 S_IR_R&R	1	1.96	3.90	7.68	11.46	15.03	21.82	27.96	40.90	49.79	66.80	77.57	96.96	81.24
🟩 S_IR_Atomic	1	1.49	2.95	5.89	8.81	11.76	17.60	23.23	34.16	44.78	64.30	86.81	131.4	143.1

Figure 5.12: OmpSs IR on the Simulated Multiprocessor Machine

5.5 Conclusion

Considering that the benchmarked shared memory platforms are small-sized processing nodes with a few processor sockets on a single mother-board, the obtained parallel performance is satisfactory since 2DSSAR executes efficiently on such platforms without wasting valuable computing resources. Efficiency in terms of performance per power consumption and size is very important for space applications. Shared memory platforms suit to this purpose. However, more powerful computing systems have to be benchmarked and considered for the proposed high performance architecture because performance on shared memory platforms is limited mainly by hardware scalability. Next, a distributed memory platform is benchmarked with 2DSSAR. It provides much more computing resources that have to be exploited by efficiently parallelizing the application.

6 Benchmarking Distributed Memory Systems

This chapter brings into focus distributed memory systems. As a possible implementation for the proposed reliable spacecraft computing architecture, such a system is benchmarked with the IR substage of 2DSSAR. The benchmarked distributed memory platform is described focusing mainly on overall system parameters, node composition, and network topology. Parallelization techniques and optimizations suitable for such systems are described, discussed, and applied to IR. Each optimization is correlated with the respective obtained result in order to understand their impact on the overall application performance. Obtained results can also be seen as an advice for spacecraft computing platform designers. They help in defining the compromise between performance and other factors like power consumption, size, and weight. In this aspect, the term *efficiency* along this chapter is used to describe the extent to which the system is used to achieve better performance.

6.1 Benchmarked Distributed Memory Computing Platform

The Nehalem Cluster residing at the High Performance Computing Center Stuttgart HLRS[1] is used for benchmarking purposes. It comprises 700 NEC HPC-144 Rb-1 Server compute nodes that are interconnected by 24 leaf and 6 backbone Voltaire Grid Director (VGD 4036) switches in a double data rate Infiniband network (Figure 6.1). In this *Fat-Tree* topology, each leaf switch interfaces 30 nodes with each backbone switch through bidirectional links. Each switch provides 36 quad-data rate ports with 40 Gbit/s throughput.

Each compute node is a ccNUMA system (with a design like the one shown in Figure 5.1 on page 72) composed of two quad-core Intel Xeon X5560 Gainestown CPUs. Based on Nehalem microarchitecture, such CPUs support 8 threads in simultaneous multithreading and have an 8 MB L3 shared cache. The integrated memory controller communicates with the memory at 1333 MHz providing an aggregate bandwidth of 32 GB/s. Each core normally operates at 2.8 GHz, but in Turbo mode up to 3.2 GHz. Manufactured at 45 nm, each quad-core CPU has a TDP of 95 W. Most of the nodes have 12 GB triple-channel main memory, but some nodes are upgraded to 24 GB, 48 GB, 128 GB and 144 GB.

Nehalem Cluster runs on Scientific Linux 6.2 Operating System that provides all GNU development tools and the execution environment. Main C/C++ compilers like *GCC* and *ICC* are also provided. Both of them are OpenMP-enabled. This means that they can compile applications for shared memory environments. Different MPI implementations like

[1] HöchstLeistungsRechenzentrum Stuttgart

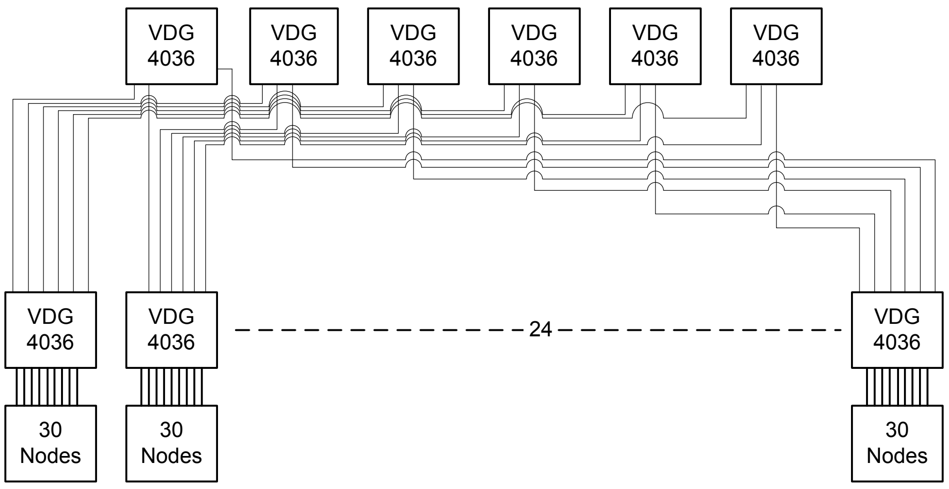


Figure 6.1: Nehalem Cluster Infiniband Network Topology

MPICH and OpenMPI are also provided for distributed memory programming. Taking into consideration that each node is a ccNUMA system, OpenMP can be preferred for programming at the node level. When more than one node is to be used, MPI is the only way to implement the communication between those nodes. MPI can also be used at the node level, when pure MPI applications have to be implemented.

For performance analysis purposes, the Scalable Performance Analysis of LArge SCale Application (SCALASCA) toolset is used. SCALASCA [136] is an integrated instrumentation, measurement and analysis open source toolset that supports different parallel programming paradigms (MPI, OpenMP and hybrid MPI + OpenMP) and languages (C, C++, and Fortran). It is composed of three components: the instrumenter, the measurement collector and analyzer, and the analysis report examiner. Generic used metrics are Time, Data Transfer Rates, and Hardware Counters. Time is further divided in Execution Time and Overhead. For MPI applications, the time spent in initialization, communication, and synchronization can be separately measured. For each collective or point-to-point communication in MPI, SCALASCA provides the number of bytes that are transferred.

For OpenMP applications the measured time hierarchy is composed of fork time, synchronization time, flush time, and idle time. Synchronization time is further divided into time spent in barriers and time spent in lock competitions. Interesting hardware counters can be selected from the Performance Application Programming Interface (PAPI [137]) available ones. Table 6.1 summarizes software environment details of the Nehalem Cluster at HLRS.

6.2 Parallelization Techniques and Optimizations

In order to benchmark distributed memory platforms and especially the Nehalem Cluster at HLRS, the IR substage of the 2DSSAR application is ported to MPI. The cluster provides a distributed memory programming and execution environment. Each node in the cluster is a ccNUMA system with shared-memory as communication medium. Nodes

Table 6.1: Software Environment on HLRS Nehalem Cluster

	Name	**Version**
Operating System	Scientific Linux release 6.2 (Carbon)	Linux 2.6.32-279.1.1.el6.x86_64
Compiler	GNU C Compiler (GCC)	4.4.6
Compiler (Intel)	Intel C Compiler (ICC)	11.1.0
Debugger	GNU Debugger (GDB)	7.2-50.el6
Profiler	GNU Profiler (GPROF)	2.20.51.0.2-5.28.el6
FFT Library	FFTW	3.2.2
MPI Library	OpenMPI	1.6
Performance Monitoring Tool	SCALASCA	1.4.1

are interconnected by an Infiniband network, which is the only communication medium available to compute nodes. For the communication over this network, the message passing paradigm is selected and for that the MPI API is used.

6.2.1 Porting IR to MPI

An incremental approach is taken to port IR to MPI. In the initial implementation a master-worker model is used. In this model the master (root process) is responsible for partitioning and distributing data and work to worker processes. Like all worker processes, the master has to process the respective amount of data too. The two-dimensional SAR data is partitioned into chunks of rows and distributed over processes. Figure 6.2 illustrates this data partitioning for a simplified case with 4 processes.

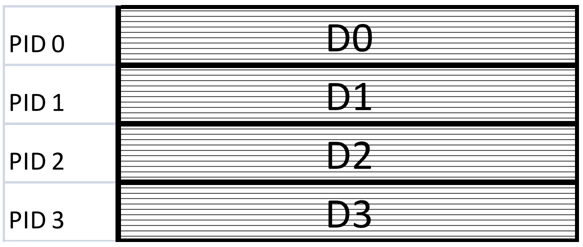


Figure 6.2: Data Distribution over 4 Processes

Data distribution by chunks of rows is very suitable for the IR substage, which on its own includes many processing steps that apply row-by-row FFT transforms. The number of rows in each chunk depends on the total number of rows and on the number of processes. The algorithm to calculate the number of rows for each process is illustrated in Algorithm 6.1, where *averow* is the average number of rows and *extra* is what remains after dividing the total number of rows with the number of processes. Each entry of the *offset* vector marks the beginning of the respective chunk. The *for* loop iterates through all process ranks from 0 to the maximal rank value. The *extra* value represents a number smaller than or equal to the maximal rank. These *extra* rows get equally distributed over all process ranks. When the number of rows for each process rank is determined and stored in the respective entry of *num_rows* vector, the beginning of the next chunk is calculated and stored in *offset* vector.

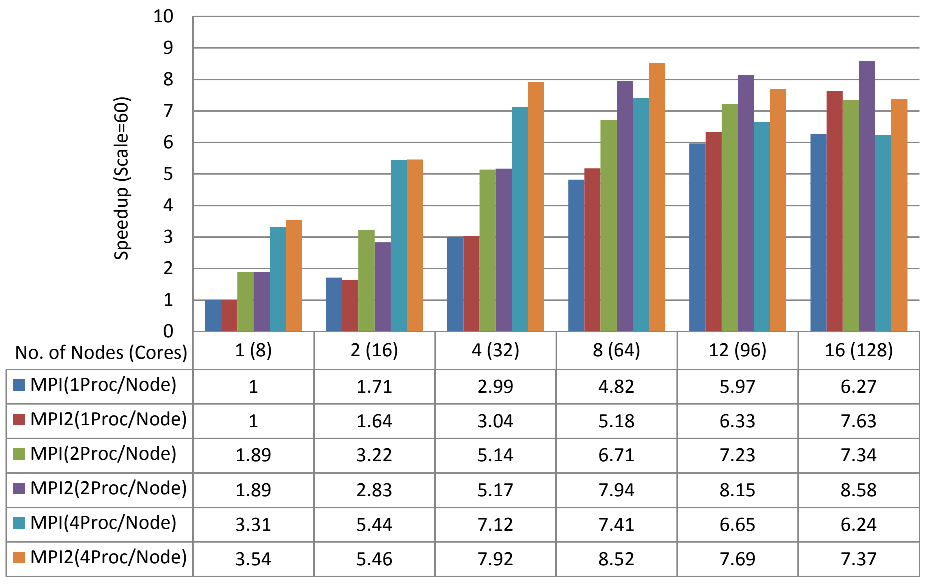
Algorithm 6.1 Calculation of the number of rows in each chunk

```
averow = total_num_rows / num_processes
extra = total_num_rows % num_processes
offset[0] = 0
max_rank = num_processes - 1
for rank = 0 to max_rank do
    if (rank <= extra) then
        num_rows[rank] = averow + 1
    else
        num_rows[rank] = averow
    end if
    offset[rank+1] = offset[rank] + num_rows[rank]
end for
```

Figure 6.3 shows initial results obtained with the master-worker MPI implementation. The application does not scale very well while increasing the number of processes being used. The reason for this is the communication overhead between master and worker processes. In the first MPI implementation, the master process is sending and receiving each chunk of data row-by-row. As MPI introduces some overhead when sending/receiving many short messages, the master process in the second MPI implementation sends and receives the whole chunk of data at once (MPI2 in Figure 6.3). Memory is linearly allocated in MPI2, so that chunks of data are sent only by indicating the starting address and the number of elements in each chunk. This linearization improves performance by improving data locality. With these optimizations the application obtains a maximal speedup factor of 8.52 that represents only 13.3 % of the theoretical linear speedup on 8 nodes (64 cores).

Both MPI implementations are run on the cluster with up to 4 processes per node in order to keep memory usage under 12 GB, which is the total amount of available main memory on most cluster nodes. Since in distributed memory programming each process has its own address space, some data has to be replicated so that each process has access to it. The amount of replicated data depends mainly on the application. In the case of IR, a lot of data has to be replicated along different processes, increasing so the amount of used main memory. Additional memory is needed for buffering in processing steps that need an inter-process synchronization. Such steps are the one responsible for *2D-fftshift*, *transposition* and the reduction operation of the *Interpolation Loop*.

In the Interpolation Loop the input data set is distributed among different processes but the output data set has to be replicated because in this loop the writing location on the output data set is determined based on the content of another array and is not known at compile time. Figure 6.4 illustrates the tendency that memory consumption has while increasing the number of processes. It shows that only 4 processes can be supported in a normal node with 12 GB main memory. But on the other side, the full computational power of each node with 8 cores is not fully exploited only by 4 MPI processes. At least 18 GB of main memory is needed to be able to schedule 8 MPI processes in one node. For this, some specific nodes with 24 GB main memory can be used, but there are not



No. of Nodes (Cores)	1 (8)	2 (16)	4 (32)	8 (64)	12 (96)	16 (128)
🟦 MPI(1Proc/Node)	1	1.71	2.99	4.82	5.97	6.27
🟥 MPI2(1Proc/Node)	1	1.64	3.04	5.18	6.33	7.63
🟩 MPI(2Proc/Node)	1.89	3.22	5.14	6.71	7.23	7.34
🟪 MPI2(2Proc/Node)	1.89	2.83	5.17	7.94	8.15	8.58
🟦 MPI(4Proc/Node)	3.31	5.44	7.12	7.41	6.65	6.24
🟧 MPI2(4Proc/Node)	3.54	5.46	7.92	8.52	7.69	7.37

Figure 6.3: MPI Implementations of IR on the Nehalem Cluster at HLRS

so many such nodes in the cluster. It is hard to find enough available nodes with 24 GB main memory so that performance measurements can be carried out.

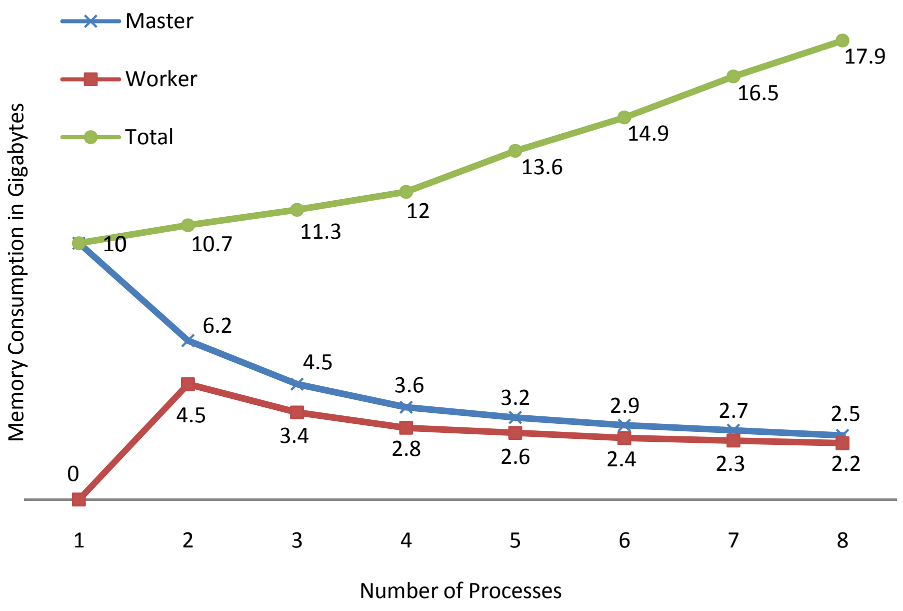


Figure 6.4: Memory Consumption with MPI

6.2.2 IR Hybrid MPI and OpenMP Implementation

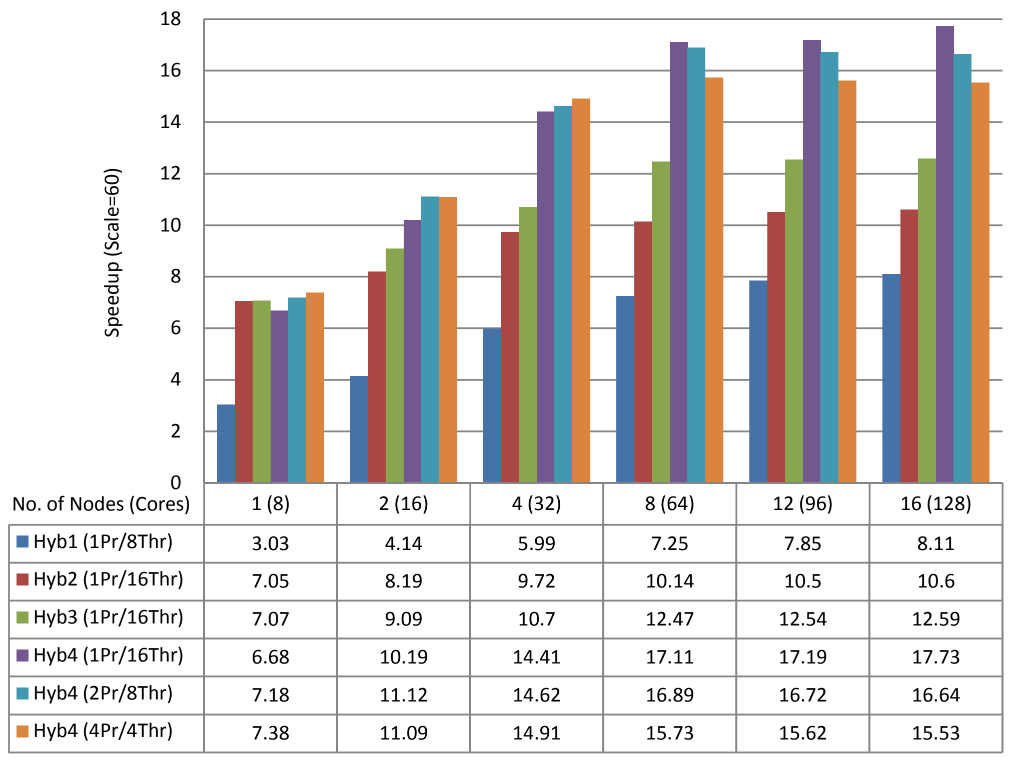
In order to increase performance and scalability of the image reconstruction application on the distributed memory Nehalem Cluster, a hybrid MPI + OpenMP version is implemented. The decision to combine MPI with OpenMP is made based on following reasons:

1. The Nehalem Cluster at HLRS is a distributed memory system composed of shared memory compute nodes. Such a system supports the hybrid paradigm by enabling programming and execution of applications that use threads for processing inside each node and message passing for inter-process communication among nodes.
2. For pure MPI implementations, the limited main memory resource in each node makes it difficult or sometimes impossible to efficiently exploit the full computational power of each node. Since each MPI process has its own private address space, data cannot be accessed among processes. Therefore, data needed from each process has to be replicated, increasing so the size of used memory while increasing the number of processes. In this way, main memory size limits the number of processes that can be scheduled on each node.
3. For hybrid implementations, no data replication is needed inside each node since threads share the same address space. The only replication needed is along MPI processes on different nodes, but this is not problematic taking into consideration that some parts of the original data set get split among the processes. Since there is no data replication overhead, as many threads as wanted can be scheduled on each node. As a consequence, hybrid applications can take advantage of the computational power provided in each node by scheduling an equal number of threads to the number of cores, or by oversubscribing with the double amount of threads. This enables the application to benefit from the simultaneous multithreading technology in order to obtain better performance.
4. Applications might be more scalable since the communication pattern is simplified in hybrid implementations. This happens because fewer processes are used and the number of communications between processes on different nodes is reduced. In pure MPI implementations there might be frequent message exchanges taking place between a group of processes in one node and another group in another node. In hybrid implementations there is only one such process per node and there are less communications taking place, but the message length is greater. Exchanging bigger messages in MPI, reduces the overhead of replicated addressing information needed in each message.
5. The last but not least important reason is the fact that applications can be efficiently programmed with hybrid programming. It can be easier to make a hybrid application scale compared to a pure MPI one since less processes are involved and less explicit communications have to be instructed. The programmer does not have to explicitly manage inter-thread communications inside each node as it is taken care of by cache coherency protocols. The impact of coherency protocol overhead on application performance can be neglected for nodes with a small number of cores (up to 8), but it should be taken into consideration on large ccNUMA systems with multiple blades integrating multiple cores.

Optimizations in the hybrid IR implementation are applied incrementally. In the first version (Hyb1), processes are distributed one per node and within each node the computation takes place through OpenMP threads. For the parallel execution of FFT transforms the multithreaded FFTW routines are used. In the second version (Hyb2), FFT transforms are manually parallelized so that they can get executed on OpenMP threads. Simultaneous multithreading (hyper-threading) is used in this version.

Since there is not so much available memory (including virtual memory) on each node, the OpenMP R&R implementation of the Interpolation Loop cannot be used. The only option is to use the OpenMP Atomic implementation to protect data at the node level. An MPI R&R technique is implemented to provide data integrity at system level. In the third version (Hyb3), some processing is done only by the master process to avoid communication overhead arising when distributing non-computational intensive work. In the fourth version (Hyb4), MPI send and receive operations are applied to the complete chunk of data and not only to each separate row.

As shown in Figure 6.5, each incremental optimization improves performance. The possibility to run multiple processes in one node, each running multiple OpenMP threads, is also investigated even though it does not improve performance. The most efficient result is obtained when using 8 nodes (64 cores). At the speedup of 17.7 that represents 26.7 % of the theoretical linear speedup, the time needed for large scale image reconstruction is reduced from 10 minutes and 39 seconds to 37 seconds. Even though there is an improvement in performance, the speedup obtained up to now is not satisfactory. Now that the problem with the exploited computational power is solved by combining MPI with OpenMP, the inter-process communication has to be optimized in order to further improve application scalability.



No. of Nodes (Cores)	1 (8)	2 (16)	4 (32)	8 (64)	12 (96)	16 (128)
🟦 Hyb1 (1Pr/8Thr)	3.03	4.14	5.99	7.25	7.85	8.11
🟥 Hyb2 (1Pr/16Thr)	7.05	8.19	9.72	10.14	10.5	10.6
🟩 Hyb3 (1Pr/16Thr)	7.07	9.09	10.7	12.47	12.54	12.59
🟪 Hyb4 (1Pr/16Thr)	6.68	10.19	14.41	17.11	17.19	17.73
🟦 Hyb4 (2Pr/8Thr)	7.18	11.12	14.62	16.89	16.72	16.64
🟧 Hyb4 (4Pr/4Thr)	7.38	11.09	14.91	15.73	15.62	15.53

Figure 6.5: Hybrid Implementations of IR on the Nehalem Cluster at HLRS

6.2.3 Optimizations on Data Distribution and Synchronization Patterns

This subsection discusses optimizations applied in processing steps of IR that exhibit poor communication performance. The initial master-worker communication model is not very scalable because, at some point in time during the execution, data is collected by the master process and then distributed again to the workers. Such a scenario occurs in three points during the IR execution (refer to Table 4.4 on page 68):

1. Before the 2D-*fftshift* operation (step 9)
2. Before *transpose* operations (step 2 and step 11)
3. After the Interpolation Loop (step 10)

6.2.3.1 Optimization 1: *2D-fftshift* Operation

Recall that for two-dimensional data the *2D-fftshift* operation swaps the first quadrant with the third the second quadrant with the fourth (Figure 4.8 on page 66). To achieve this in the initial master-worker model, the whole two-dimensional data set is collected at the master process that applies the *2D-fftshift* operation and then distributes again the new data set to worker processes. In the mean time, worker processes are idle waiting for data from the master process. In addition to this, the communication with one master process is very time consuming since all workers want to communicate with only one master process, which in turn services workers one by one.

In order to reduce these waiting times and obtain better application scalability, the implementation is changed so that communication does not take place before the *2D-fftshift* operation but while the operation is being applied. A peer-to-peer communication pattern is proposed, where nodes communicate in couples. Nodes that have data from the first and second quadrants send and receive data only to and from nodes with the third and fourth quadrants respectively. To achieve this communication order, the way how data is distributed among processes is modified, so that it fits to the quadrants idea. The first half of processes has to work on the first half of the two-dimensional data set (image) and the second half of processes has to work on the second half of the image. Corresponding processes in the first and second half need to have the same amount of data, so that they can send and receive data of the same size among each other. Figure 6.6 illustrates how this works. It shows the quadrant partitioning according to the notion in Figure 4.8 and a simplified data distribution among 4 processes. It also depicts the communication within peers in a group of 4 processes.

The first half of the image (quadrants A and B) goes to processes 0 and 1 and the second half (quadrants C and D) goes to processes 2 and 3. Process 0 has to be able to work on the same number of rows as process 2 and the same stands for processes 1 and 3. Process 0 sends A1 and B1 to process 2 and waits to receive C1 and D1 from it. In the same time process 1 sends A2 and B2 to process 3 and waits to receive C2 and D2 from him. When each process has completed receiving the chunk of data, it needs to swap the left half of the chunk with the right half. For example, process 0 will have to swap C1 with D1.

The peer-to-peer communication pattern is internally implemented using non-blocking send/receive MPI functions that enable overlapping of communications and computations

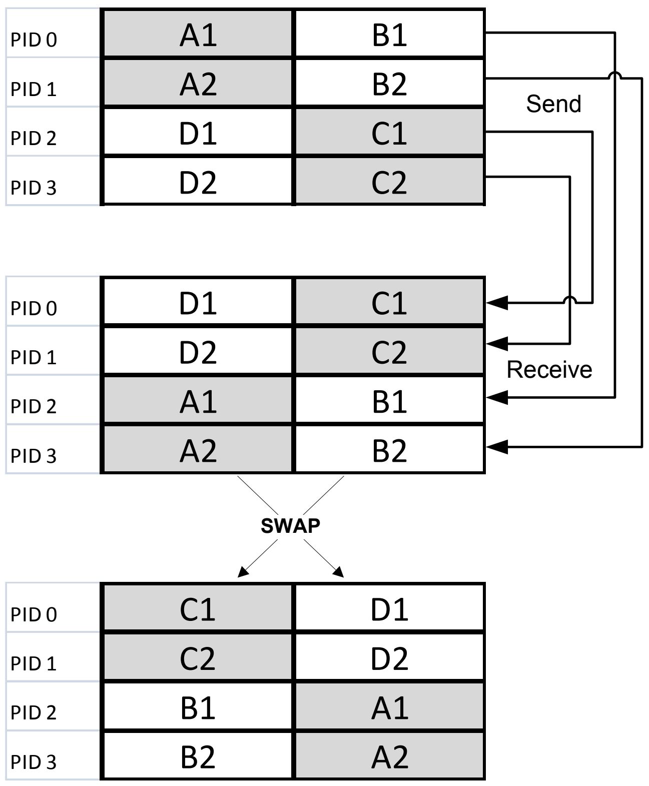


Figure 6.6: Inter-Process Communication and Swapping in the *2D-fftshift* Operation

among different processes. The only drawback of such an approach is that additional buffering memory is needed to store data being sent and received.

6.2.3.2 Optimization 2: Transpose Operation

The transpose operation in the context of IR processing reflects each element of the two-dimensional data set (image) over the main diagonal. In the context of programming languages, this can be translated as a transformation of rows into columns or columns into rows. In the initial master-worker implementation, the whole image is collected at the master process that transforms rows into columns and then distributes again the new image to worker processes. Again, communication problems impact application scalability and performance.

To solve these problems, a similar approach to the one taken for *2D-fftshift* is taken for transpose operations too. By combining the communication with the transpose operation itself, the time needed for communication is overlapped with the time needed for transpose operation itself. To apply this update to the communication pattern for transpose operations, the chunk of data that each process is working on, is partitioned into tiles as shown in the first stage of Figure 6.7. By using buffers to store data that has to be moved, an all-to-all communication pattern is implemented. This communication pattern is expected to be better than the master-worker one because now each process is sending and receiving less data to and from all other processes (the central master is omitted).

First, data is divided into chunks and distributed to processes 0 to 3 (in the simplified example with 4 processes). To efficiently implement transposition, a vertical partitioning is proposed. For a number of processes equal to *N*, the image gets divided into *N* tiles. Tiles positioned in the diagonal of the image do not have to move. The respective process transposes and stores them locally while it is exchanging the second tile with another process. As soon as an exchange finishes, the received tiles get transposed and stored while another exchange has already started. In this way, tile exchanging is overlapped with local tile transposition and storing operations. Each process sends and receives to and from all processes other than itself, as shown in Figure 6.8. On the receiving side the data is buffered, transposed, and then stored at the right position in the destination image tile and chunk.

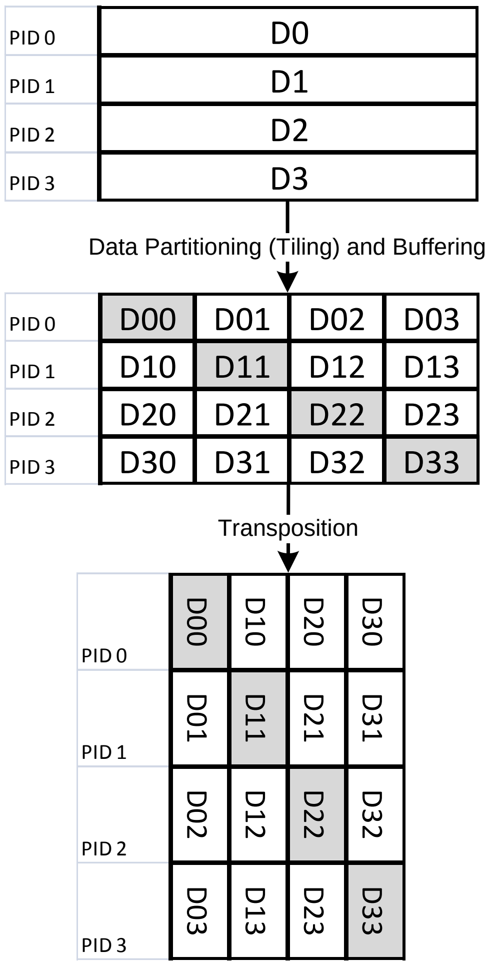


Figure 6.7: Overlapping Communications and Tile-Transpositions

6.2.3.3 Optimization 3: Interpolation Loop Reduction

Additional communication overhead is coming from communications needed to reduce overlapped image regions after the Interpolation Loop. Figure 6.9 shows the logical data

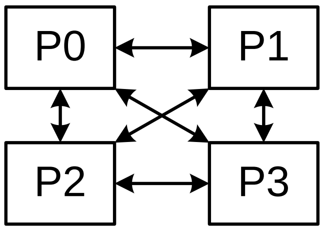


Figure 6.8: Inter-Process Communication in Transpose Operations

distribution with overlapped regions among 4 processes. The number of overlapped regions depends only on the number of MPI processes. In the pure MPI implementation of IR, more processes are scheduled, compared to the hybrid MPI + OpenMP implementation. However, this does not mean that more overlapped regions occur in the pure MPI implementation compared to the hybrid one because for the latter one there are additional overlapped regions between data that is distributed among threads inside each process. This inter-thread reduction carried out by using OpenMP atomic operations is discussed in subsection 5.3.4 on page 82. Here, only the inter-process MPI reduction is discussed. It is applied to both MPI and hybrid implementations of IR.

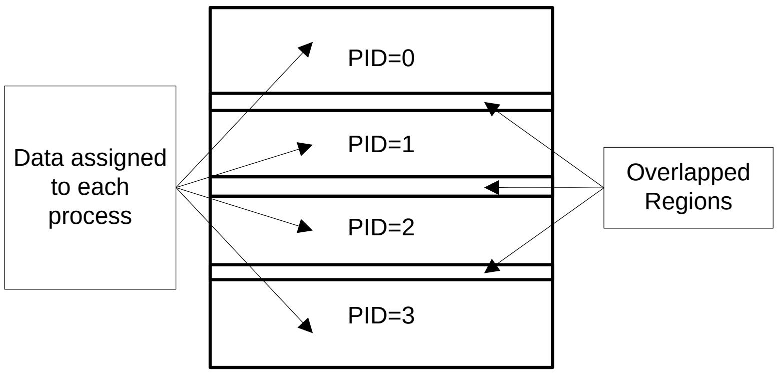


Figure 6.9: Overlapped Regions

To avoid the communication overhead coming from collective reduction operations, a local reduction between neighbor processes is implemented. In this way, communication takes place only for the amount of data which is overlapped and not for the complete image. During the reduction, the first and the last processes (process 0 and process 3 in the simplified example) communicate only with one neighbor process. All other processes have to reduce their data with two respective neighbor processes. Data reduction is scheduled in an ordered way so that the first process sends data to the second process, which accumulates new values with the old ones and sends results back to the first process.

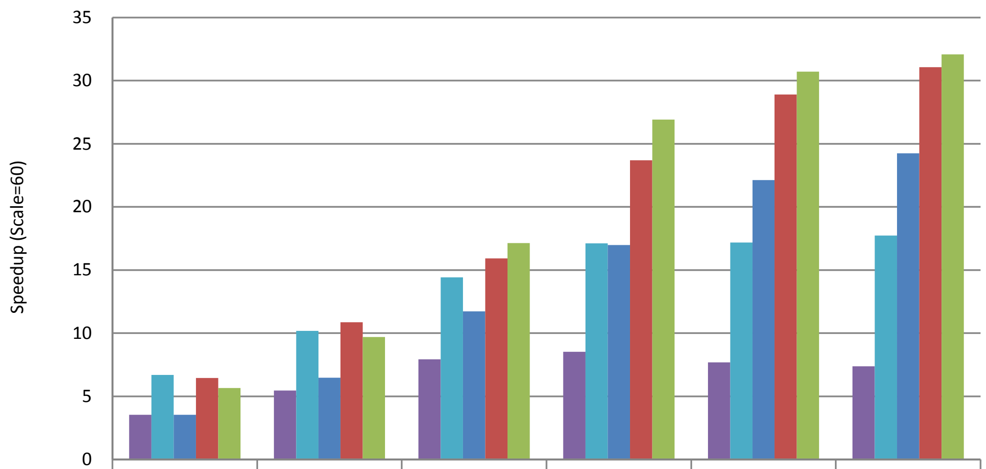
6.2.3.4 Performance Results

By integrating all these three communication optimizations on the pure MPI and hybrid MPI + OpenMP implementations, the third MPI and the fifth hybrid versions are created.

Obtained results are depicted in Figure 6.10, which compares the speedup factor between old implementations (MPI2 and Hyb4) and new ones (MPI3 and Hyb5). For MPI3, the speedup is measured for two specific cases. In the first one, 4 MPI processes are scheduled in each node without using more than 12 GB of main memory. In order to be able to schedule more processes, nodes that provide more main memory have to be used. As a consequence, nodes with 24 GB of main memory are used when scheduling 8 MPI processes per node. The second case improves performance, but with high memory consumption costs.

Compared to the old MPI2 implementation, the new MPI3 improves quite a lot the performance with most efficient speedup factor of 23.7 on 8 nodes. With this speedup factor that represents 37 % of the theoretical linear speedup, the time needed for large scale image reconstruction is reduced from 10 minutes and 39 seconds to 26 seconds. This time is reduced to 20 seconds when 16 nodes are used, but the speedup factor of 32 now represents only 25 % of the theoretical linear speedup. This shows that, even though higher speedup values are obtained, application scalability is getting worse when using more than 8 nodes. The reason for this is the amount of inter-process communications that keeps rising while increasing the number of processes.

Unfortunately this is also true for the Hyb5 implementation. Even though Hyb5 performs a little better than MPI3, the application scalability is not at the desired levels. IR performance is improved quite a lot with Hyb5 when compared to the old Hyb4 implementation. On 8 nodes, elapsed time is reduced to 21.5 seconds giving the most efficient speedup factor of 27, which represents 42 % of the theoretical linear speedup. On 16 nodes, elapsed time is further reduced to 18 seconds, but the speedup factor of 32.3 represents only 25.2 % of the theoretical linear speedup.



No. of Nodes (Cores)	1 (8)	2 (16)	4 (32)	8 (64)	12 (96)	16 (128)
🟪 MPI2 (4Proc/Node)	3.54	5.46	7.92	8.52	7.69	7.37
🟦 Hyb4 (1Proc:16Thr/Node)	6.68	10.19	14.41	17.11	17.19	17.73
🟦 MPI3 (4Proc/Node-12GB)	3.53	6.48	11.71	16.99	22.13	24.25
🟥 MPI3 (8Proc/Node-24GB)	6.45	10.87	15.93	23.69	28.90	31.06
🟩 Hyb5 (1Proc:16Thr/Node)	5.66	9.68	17.13	26.92	30.72	32.08

Figure 6.10: IR with Communication Optimizations on the HLRS Linux Cluster

6.2.4 Embarrassingly Parallel Pipelining Technique

In order to use the distributed memory Nehalem Cluster in a more efficient way, a pipelined version of IR is implemented. This pipelined version exploits a specific feature of an iterative image reconstruction scenario, in which multiple images get processed in the long run. It is named *pipelining* because different image reconstructions are pipelined along different nodes in the cluster. This technique is possible in distributed memory environments with multiple nodes that have enough memory resources to accommodate the complete image reconstruction. Multiple processes communicating by message passing are distributed along different nodes, but the parallel execution inside each node is selected to be done by OpenMP threads, since the hybrid implementation shows a slightly better parallel performance than the pure MPI one.

By having each node processing a single image, the time spent for inter-node communication is reduced (Figure 6.11). In the pipelined IR version, it takes some time to output the first image, but then other images are generated much faster. The number of images processed in one round depends on the number of nodes used in the cluster.

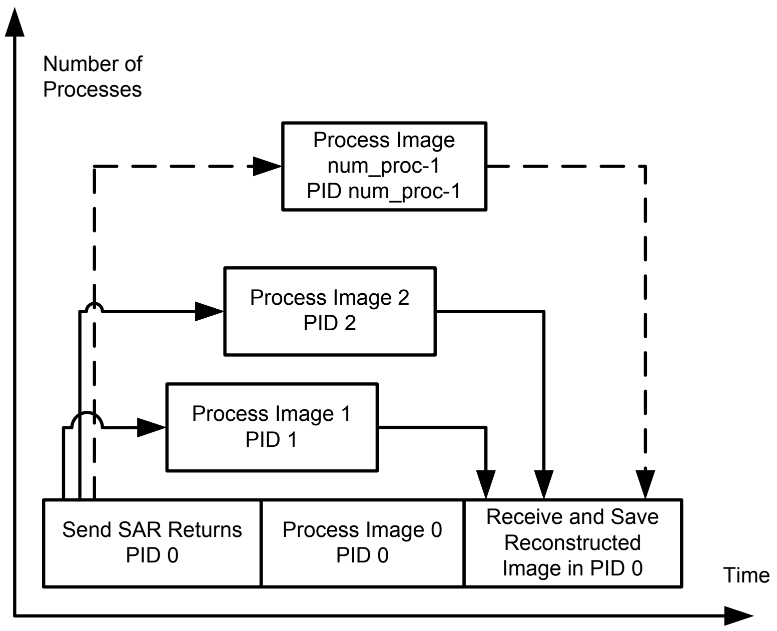
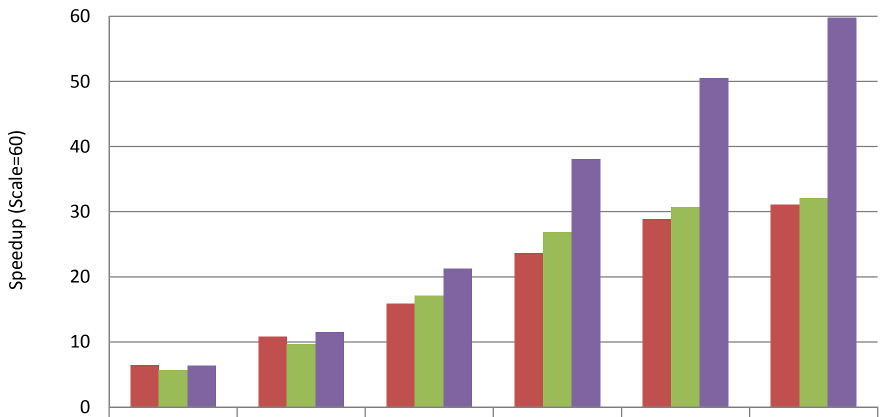


Figure 6.11: Pipelined SAR Image Reconstruction

A separate process is created for each node. Each process that reconstructs a separate image is responsible for:

1. Retrieving the corresponding SAR sensor returns (raw data) via MPI,
2. Reconstructing the SAR image out of SAR sensor returns in parallel by using OpenMP threads,
3. Returning the reconstructed SAR image back to the root process, which itself saves the image.

In Figure 6.12, the speedup obtained with the pipelining technique is compared to the best one obtained with the pure MPI and the hybrid MPI + OpenMP implementations. This chart shows that the benefit of using pipelining technique starts to become visible when using more than 4 nodes. The number of processes and nodes being used increases while increasing the number of images that have to be reconstructed. More important is that the obtained parallel performance increases too. For the pipelined IR implementation, the most efficient speedup factor is also the highest one, which is obtained when using 16 nodes. To summarize, a speedup factor of 60 is obtained when using 128 cores in 16 nodes of the Nehalem Cluster and the average elapsed time per image reconstruction is reduced to 10.7 seconds. This means that the distributed memory system is used with 47 % efficiency. When using 64 cores in 8 nodes a speedup factor of 38 is obtained with an efficiency of 59.4 %. Figure 6.13 depicts the reduction in the time needed to reconstruct a large scale image depending on the number of nodes being used and on the optimization or programming model being applied. Table 6.2 summarizes all implemented IR versions listing respective applied optimizations.

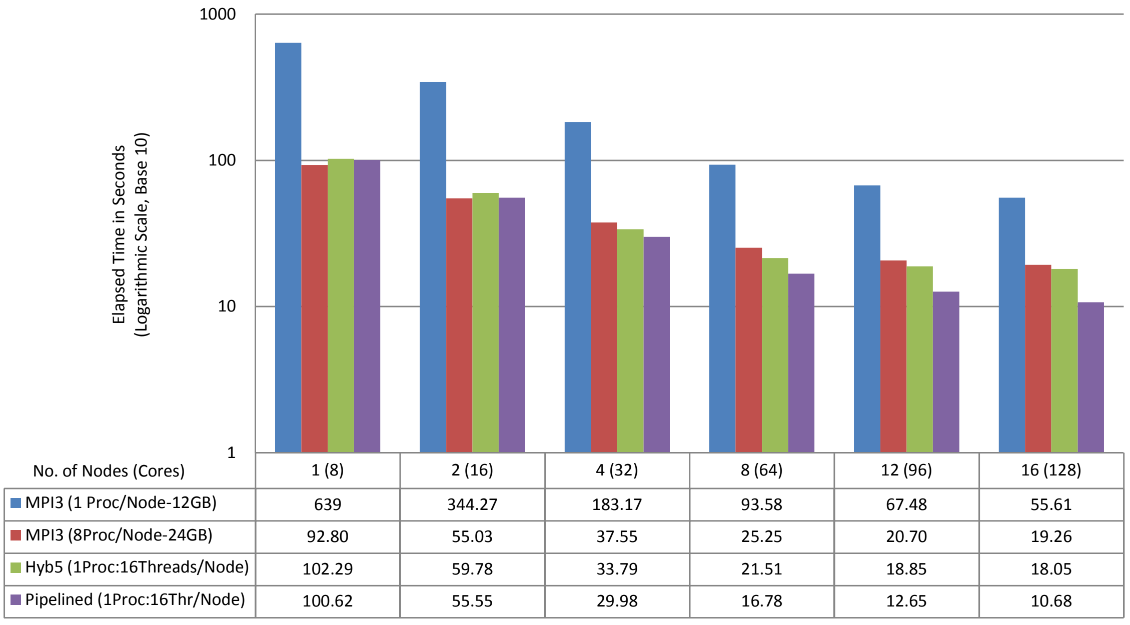


No. of Nodes (Cores)	1 (8)	2 (16)	4 (32)	8 (64)	12 (96)	16 (128)
🟥 MPI3 (8Proc/Node-24GB)	6.45	10.87	15.93	23.69	28.90	31.06
🟩 Hyb5 (1Proc:16Threads/Node)	5.66	9.68	17.13	26.92	30.72	32.08
🟪 Pipelined (1Proc:16Thr/Node)	6.35	11.50	21.30	38.05	50.48	59.80

Figure 6.12: Pipelined IR on the HLRS Linux Cluster

6.3 Conclusion

Obtained results prove that performance of such image reconstruction applications can be improved by using a certain number processing nodes on a distributed memory platform. However, programmers need to spend some time parallelizing them in an efficient way. Contrary to shared memory systems performance that is limited mostly by available hardware resources, performance on distributed memory systems is limited by the application scalability factor too. For 2DSSAR, scalability is mainly impacted by inter-process communications needed in IR. Such communications on shared memory platforms are either implicitly executed through memory operations that are much faster than network communications on distributed memory platforms or unnecessary when shared data resides



No. of Nodes (Cores)	1 (8)	2 (16)	4 (32)	8 (64)	12 (96)	16 (128)
🟦 MPI3 (1 Proc/Node-12GB)	639	344.27	183.17	93.58	67.48	55.61
🟥 MPI3 (8Proc/Node-24GB)	92.80	55.03	37.55	25.25	20.70	19.26
🟩 Hyb5 (1Proc:16Threads/Node)	102.29	59.78	33.79	21.51	18.85	18.05
🟪 Pipelined (1Proc:16Thr/Node)	100.62	55.55	29.98	16.78	12.65	10.68

Figure 6.13: IR Elapsed Time on the HLRS Nehalem Cluster

Table 6.2: Optimizations applied on each IR version for distributed memory systems

Version	Optimization
MPI1	MPI is used to implement a master-worker communication model. Many small messages are sent/received among processes.
MPI2	Less, but bigger messages are sent/received. Linear chunks of rows.
MPI3	Customized MPI communication pattern for some processing steps of IR.
Hyb1	MPI is used to implement a master-worker model for communication among nodes. OpenMP is used for multithreading within each node. Multithreaded FFTW routines are used.
Hyb2	FFT transforms are parallelized in OpenMP threads. SMT is enabled.
Hyb3	Communication overheads arising from the distribution of non-computational intensive work are avoided by having such work executing only on one node.
Hyb4	Less, but bigger messages are sent/received. Linear chunks of rows.
Hyb5	Customized MPI communication pattern for some processing steps of IR.
Pipelined	Multiple Image reconstructions are distributed among nodes. Each node reconstructs a full image.

on processor's local memory. In the current position, being unsatisfied with the limited performance of shared memory systems and with the low scalability of the application on distributed memory ones, there is only one alternative left: heterogeneous CPU/GPU systems. In such systems, better and more efficient parallel performance is expected to be obtained by using the shared memory multiprocessor supported by accelerator modules that incorporate GPUs and very high speed memory modules.

7 Benchmarking Heterogeneous Systems

This chapter focuses on heterogeneous systems composed of CPU-based processing modules and GPU-based accelerators. As alternative to shared and distributed memory ones, heterogeneous systems should also be considered for the proposed reliable spacecraft computing architecture. Typical platforms with multi-core CPUs and many-core GPUs are benchmarked with the SAR Image Reconstruction substage of 2DSSAR. This chapter begins by describing specific features of benchmarked heterogeneous platforms.

To make a performance evaluation and comparison between novel CPU and GPU technologies, the CPU implementation of IR is revised and further optimized for vector processing on CPU SIMD units. Other sequential optimizations are also applied in order to obtain the best sequential IR version, performance of which is later compared with the one obtained on the GPU. IR is ported to CUDA, so that performance estimations on accelerator modules can be made. The same CUDA implementation is also ported to OmpSs, which is a programming model for heterogeneous systems that among other things tends to increase programming productivity.

7.1 Benchmarked Heterogeneous Computing Platforms

Two similar platforms are used for benchmarking purposes, one available at Leibniz Supercomputing Center (LRZ[1]) and the other one available at Barcelona Supercomputing Center (BSC). They are both based on the same system architecture composed of the ccNUMA CPU subsystem and the GPU accelerator module. As shown in Figure 7.1, the ccNUMA CPU subsystem is a multi-socket shared memory system with two multi-core CPUs, each associated with a local part of main memory and interfaced with the accelerator module through independent PCIe lanes. Nehalem CPUs used in both platforms, provide integrated memory controllers and QPI ports. As discussed in subsection 5.1.1 on page 71, Intel Nehalem microarchitecture improves performance in NUMA systems by providing 32 GB/s aggregate memory bandwidth per multi-core CPU and 25.6 GB/s bandwidth per each QPI port.

Concerning the ccNUMA subsystem, the benchmarked platforms differ only by a few parameters. The LRZ one uses two quad-core Nehalem CPUs running at 2.13 GHz. Each CPU has direct access to 6 GB of memory, building a 12 GB main memory that is shared by 8 cores in the ccNUMA subsystem. The BSC platform provides a much more powerful ccNUMA subsystem with two hexa-core Nehalem CPUs running at 2.53 GHz that share 24 GB of main memory. Table 7.1 lists software environment details for both platforms.

[1]Leibniz Rechenzentrum

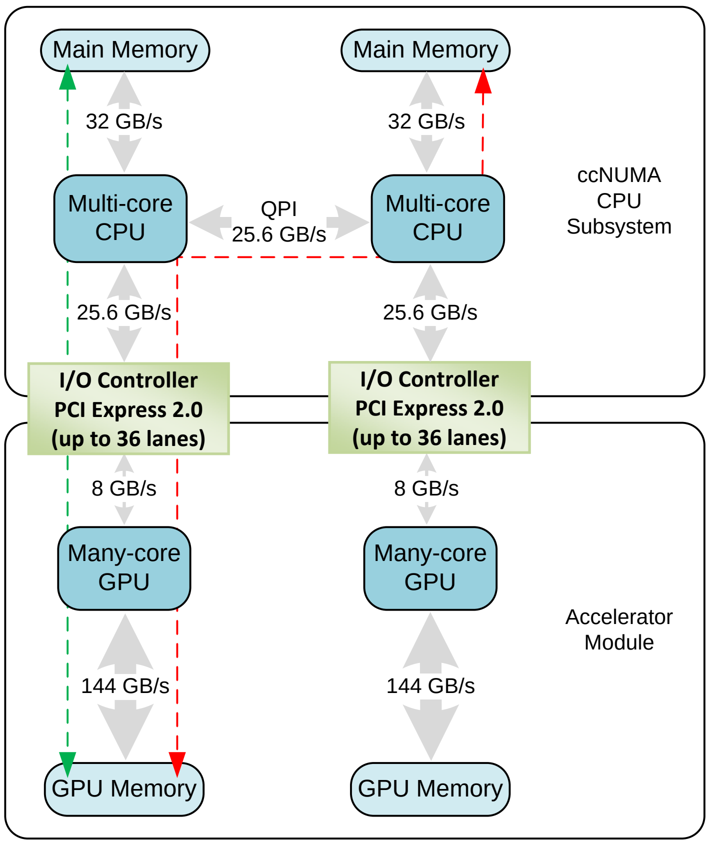


Figure 7.1: Architecture of the Benchmarked Heterogeneous Platforms

The accelerator module on both platforms is composed of two GPU boards, each containing an NVIDIA Tesla GPU, which accesses 6 GB of GPU global memory with a memory bandwidth of 144 GB/s (4.5 times higher than the CPU memory bandwidth). The most limiting parameter in the whole platform is the GPU PCIe bandwidth that can reach only up to 8 GB/s. If enabled, ECC on the GPU memory can fix single-bit errors and report double-bit errors. Enabling ECC will cause some of the GPU memory to be used for ECC bits, so the user available memory will decrease from 6 GB to 5.25 GB. Two NVIDIA C2070 boards are used on the LRZ platform. Each of them integrates 448 CUDA cores running at 1.15 GHz in 14 SMs. The BSC platform uses NVIDIA M2090 boards that integrate 512 CUDA cores running at 1.3 GHz in 16 SMs. TDP reaches 238 Watt in C2070 and 225 Watt in M2090. Being based on Fermi architecture [138], Tesla GPUs provide following features:

1. The third generation SM provides 32 CUDA cores and 8 times more double precision floating-point performance over previous architectures. The dual warp scheduler simultaneously schedules and dispatches instructions from two independent warps. A CUDA warp is a group of 32 threads, which represents the minimum data size to be processed in SIMD fashion by a CUDA multiprocessor. Each SM has its own local memory. In Fermi GPUs it is possible to use some of this local memory as a first-level (L1) cache for global memory references. The local memory is 64K in size,

Table 7.1: Software Environment on Heterogeneous Platforms

	LRZ	**BSC**
Operating System	SUSE Linux Enterprise Server 10 (x86_64)	Red Hat Enterprise Linux Server 6.0 (x86_64)
GNU C Compiler	GCC 4.6.0	GCC 4.6.1
Intel C Compiler	ICC 12.0	ICC 12.0.4
OmpSs Mercurium Compiler	-	MCC 1.3.5.8
GPU Programming	CUDA 3.2	CUDA 3.2
FFT Library	FFTW 3.2.2	FFTW 3.2.2
Performance Monitoring Tool	Periscope 4.1	Paraver 4.3.4

and can be split 16K/48K or 48K/16K between L1 cache and shared memory.

2. The second generation of parallel thread execution instruction set architecture includes a unified address space with full C/C++ support. It supports the full IEEE 754-2008 32-bit and 64-bit precision and full 32-bit integer path with 64-bit extensions. Memory access instructions support transition to 64-bit addressing. The memory unifies three separate address spaces (thread private local, block shared, and global) for load and store operations that were previously specific for each of them.

3. The improved memory subsystem is the NVIDIA's parallel data cache hierarchy with configurable L1 and unified L2 caches. Fermi is the first architecture with ECC memory support and with improved atomic memory operation performance. By combining more atomic units in hardware and by adding the L2 cache in Fermi, atomic operations performance is increased compared to previous generations. Global memory is shared between all SMs in the grid. Fermi provides six 64-bit DRAM channels that support SDDR3 and GDDR5 DRAMs. Up to 6GB of GDDR5 DRAM can be connected to the chip.

4. Fermi architecture features a two-level distributed thread scheduler. A global work distribution engine schedules thread blocks to different SMs at chip level, while inside the SM each warp scheduler distributes 32 thread warps to its execution units. The GigaThread global scheduler provides 10 times faster application context switching, concurrent kernel execution, out of order thread block execution and dual overlapped memory transfer engines.

Further analysis is needed concerning platform communications. Remote memory accesses impact application performance on a NUMA architecture. This impact becomes even larger when transferring data between GPU and remote CPU memories. As shown in Figure 7.1, transfers between GPU memory and remote CPU memory (red line) take longer than local transfers (green line) because data has to pass through an additional QPI link between CPUs, which adds an additional latency to the communication. This fashion of transfers takes place especially when threads on the remote CPU initialize data according to the *First Touch* policy, but also when transferring data from one GPU to the other. In the latter case, the transfer will take place in two steps. First, data is transferred to the CPU memory that is local to the sending device. Then, the receiving device has to copy data from the CPU memory, which is remote in this case.

To make use of both memories connected to each CPU, it might be better to pin threads that manage GPU contexts on different CPUs, so that they can use separate local CPU memories. If only one GPU of the platform is used, remote memory accesses occur only when CPU threads initialize data on remote memory and the same data has to be copied to GPU memory. If both GPUs are used, additional remote accesses occur when data has to be copied from one device to the other.

Another point worth discussing is the limited bandwidth in the PCIe link that is used to connect each CPU with the respective GPU. The PCIe 2.0 theoretical bandwidth of 8GB/s can cause a throughput bottleneck when a significant amount of data is transferred between a CPU and a GPU in a heterogeneous system. The subject is well known in the GPU computing community. A number of researchers discuss bandwidth troubles that arise with frequent or poorly managed data movements between devices. The authors in [139] examine multiple GPU systems and acknowledge that unless a full working set of data can fit into GPU memory, PCIe will be a bottleneck. Other authors (in [140] and [141]) express similar concerns and recommend rewriting algorithms in order to limit PCIe transfers as much as possible.

7.2 SAR Image Reconstruction on Heterogeneous Systems

This section discusses approaches taken to port IR to heterogeneous systems with multi-core CPUs and many-core GPUs. Respective obtained results are discussed and compared so that performance considerations can be extracted. It begins with sequential code optimizations for efficient execution on a single CPU core. It is very important to obtain this optimized sequential version of IR so that its performance can be later compared to the ones obtained on the GPU. The main purpose is to optimize IR for vector processing on the CPU because it has no meaning to compare results obtained on the GPU with the one obtained on the CPU, if the CPU code is not vectorized. Then, parallel versions of the vectorized and original IR are optimized for multithreaded execution on the ccNUMA subsystem of each benchmarked platform.

In order to be able to run IR on the accelerator module, it has to be first ported to CUDA C programming language. Different incremental versions are implemented, beginning with the small and large scale IR, going to a version that uses both CPU and GPU for IR processing, and concluding with the version that uses multiple GPU devices. OmpSs programming model is used to improve programming productivity for different heterogeneous systems. IR is ported with OmpSs to heterogeneous CPU/GPU systems, but it can also be ported to other systems much easier than CUDA C versions.

7.2.1 Optimizing IR for the ccNUMA Subsystem

Recall that GPU computing involves vector and SIMD processing to exploit fine-grain data parallelism in the application. In order to be able to consistently compare performance obtained on the GPU with the one obtained on the CPU, the application has to be able to take advantage of SIMD units that are present in most novel CPU architectures. Nehalem processors provide direct support for SIMD instructions through multiple functional units and separate core registers called MMX and XMM registers. There are 8 64-bit MMX registers aliased for legacy x87 instructions, and 16 128-bit XMM registers that store 4 single-precision or 2 double-precision floating-point operands. Load and store units can retrieve and save 128 bit operands from cache or from main memory. Other features of Nehalem microarchitecture related to sophisticated control logic and large cache hierarchy are used to increase application performance. The impact of such sequential optimizations on parallel application performance is also investigated in this subsection.

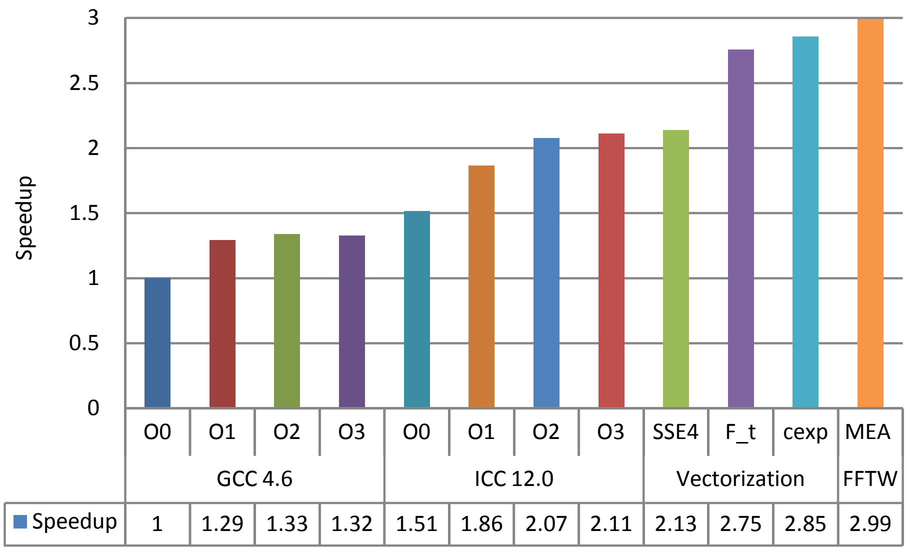
7.2.1.1 IR Sequential Optimizations

IR is further optimized for sequential and parallel CPU execution, so that results obtained with the best CPU version of IR can be later compared with the ones obtained on the GPU. Basic sequential optimizations can be carried out by most common compilers. Some optimizations try to reduce machine code size, whereas others try to create faster code, potentially increasing its size. Different optimization levels have been defined to simplify the process that enables them before compilation. Optimization level zero provides no optimization at all. The first level optimization O1 reduces the size of the compiled code while increasing its performance. The purpose of the first level optimization is to produce an optimized binary in a short amount of time. These optimizations typically do not require significant amount of compile time to complete. The second level optimization O2 performs all other supported optimizations within the given architecture that do not involve a trade-off between size and speed. For example, *loop unrolling* and *function inlining* optimizations are not performed. Such optimizations have the effect of increasing code size while also potentially generating faster code.

The third level O3 enables even more optimizations by putting speed over size. This includes optimizations enabled at O2 plus *register renaming* and *function inlining*. Register renaming gets rid of *write after read* and *write after write* data dependencies and exposes more instruction-level parallelism. With function inlining, the compiler inserts the complete function body in every place the function is called, rather than generating code that calls the function in the place it is defined. This improves performance, but depending upon functions that are inlined, it can also drastically increase the size of the object. Although O3 can generate fast code, the binary size increase can have negative effects on its speed. For example, if the binary is bigger than the available instruction cache, serious performance penalties can be observed. Therefore, it may be better to simply compile at O2 to increase chances that the binary fits in instruction cache.

Because of these conditions, these optimizations are applied one by one and their impact on application performance is analyzed. The vectorization compiler capability is usually enabled with O3. In terms of elementary optimizations, vectorization involves unrolling of a loop combined with the generation of packed SIMD instructions. A loop iterating independently on multiple data elements can execute more efficiently because packed instructions operate on more than one data element at a time. It is sometimes called auto-vectorization to emphasize that the compiler is able to automatically identify and optimize suitable loops on its own. If not, intrinsic operations have to be used to explicitly vectorize the code. The latter case requires much more programming efforts and time.

Two different compilers (GCC 4.6 and ICC 12.0) are investigated for compiler optimizations on IR. Figure 7.2 shows the increase in speedup when applying different compiler and other sequential optimizations related to data layout and to FFTW library calls. These results are obtained on the platform available at LRZ. With the GCC compiler, performance improves mostly when applying optimization O1. Applying O2 improves performance a little bit further. However, O3 does provide any improvement probably because the capacity of the instruction cache is exceeded. An increase in speedup is obtained when switching to the native ICC compiler, which probably provides more optimizations for Intel processors. Almost the same behavior is observed when enabling optimization levels in ICC. O1 improves performance a lot and after that comes O2. In this case O3 increases the speedup with 1.5 %, as opposed to the O3 in GCC, which slows down the application.



	O0	O1	O2	O3	O0	O1	O2	O3	SSE4	F_t	cexp	MEA
	GCC 4.6				ICC 12.0				Vectorization			FFTW
Speedup	1	1.29	1.33	1.32	1.51	1.86	2.07	2.11	2.13	2.75	2.85	2.99

Figure 7.2: IR Sequential Optimizations

Even though auto-vectorization is enabled with the O3 optimization, the compiler needs a specific flag (e.g. -msse4 for ICC), which instructs to use specific SSE instruction set extensions. Speedup is further increased only by 1.23 % when specifying this flag. Considering that a higher increase in the speedup is expected, reasons that make it impossible for the compiler to vectorize loops in IR code have to be investigated. As usual the place to start is the most time consuming step of IR, the Interpolation Loop, which is composed of three nested loops. The two outer ones scan the 2-dimentional input data (image) and an indexing array that contains offsets to specific rows in the output image. The most inner loop iterates through overlapped regions along columns of the output image (on the left side of Figure 7.3). In each iteration of the inner loop, only one element of the specified row is updated. This fashion of accessing the output image makes vectorization impossible, because accessed data is not consecutively stored in memory.

In order to fulfill compiler requirements for auto-vectorization, the loop is revised to work on the transposed version of the output image, which is then later transposed back for further processing. The access pattern is now like the one shown on the right side of Figure 7.3. Having this pattern, for each row of the image, consecutive elements can be updated along iterations of the inner loop. This means that the compiler is now able to vectorize that loop. In Figure 7.2 this is identified by column F_t, representing the transposed version of the output image F. This optimization brings 60 % speedup increase for the Interpolation Loop and 22.5 % for the overall image reconstruction substage.

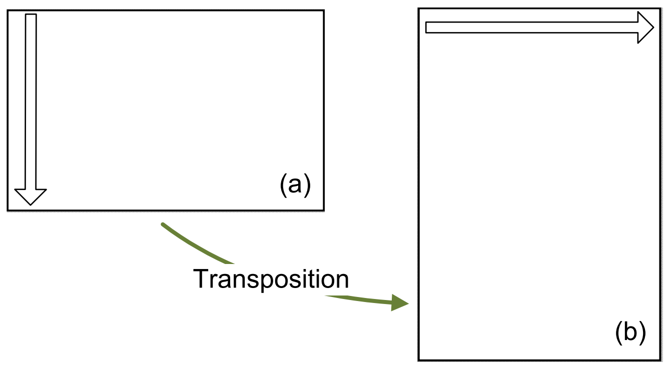


Figure 7.3: Interpolation Inner Loop Access Pattern over the Output Image. Image (a) shows the access pattern before transposition, Image (b) shows the access pattern after transposition.

There is still room for further optimizations. The compiler is not able to vectorize some compression and decompression loops. The problem for these loops stands neither in data layout nor in the access pattern because elements of input and output data are accessed in a consecutive way. The problem stands in the CEXP function used to calculate the exponential value of a complex variable. There is no implementation for this function in the SSE4 instruction set extension. To deal with this, the CEXP function is decomposed into the following Equation 7.1, which calculates the exponential value of the complex variable z:

$$exp\left(z\right) = exp\left(creal\left(z\right)\right) \times \left(cos\left(cimag\left(z\right)\right) + I \times sin\left(cimag\left(z\right)\right)\right) \tag{7.1}$$

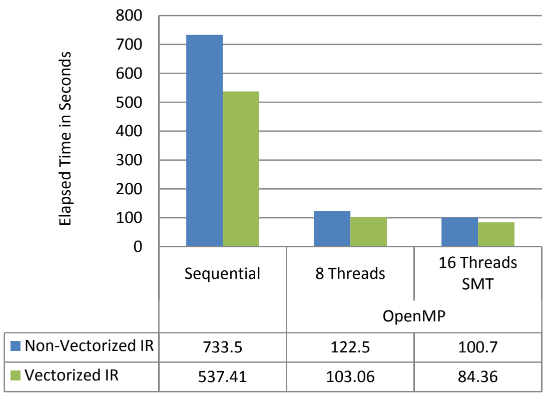
Functions *creal()* and *cimag()* return respectively the real and the imaginary part of the complex variable and functions *sin()* and *cos()* return the *sine* and the *cosine* of a float variable. When applying this decomposition, the compiler is able to vectorize these loops and IR obtains 3.42 % higher speedup.

In FFT transforms, plan creation and execution routines can be further optimized. For sequential FFT transforms, it is very important that the FFTW API creates only once a very well optimized plan, which will be executed many times. This is the reason that *FFT-MEASURE* is now used for sequential plan creation instead of *FFT-ESTIMATE*. Recall from subsection 5.3.3 on page 78 that *FFTW-MEASURE* instructs FFTW to run and measure the execution time of several FFTs in order to find the best way to compute the transform. This process takes some time, depending on the machine and on the size of the transform. On the contrary, *FFTW-ESTIMATE* does not run any computation and just builds a reasonable plan that is probably sub-optimal. In sequential execution, the application performs many (for a large scale image, up to 20 thousand) transforms of the same size and initialization time with *FFT-MEASURE* does not influence that much the overall performance.

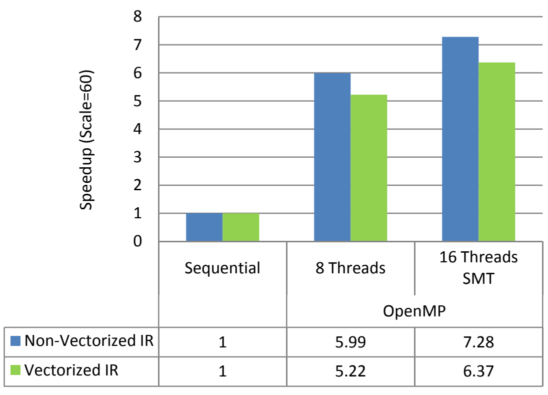
When applying all these sequential optimizations a speedup factor of 2.989 is obtained and the time needed for large scale image reconstruction goes down from 26 minutes and 40 seconds to less than 9 minutes. This latest result is used as a reference value when calculating the speedup gained on the GPU compared to the CPU.

7.2.1.2 Vectorization and Multithreading

To harness the computational power of multi-core CPUs in the ccNUMA subsystem, the best sequential IR code is further parallelized in OpenMP threads. The elapsed time and the speedup obtained for large scale IR are depicted in Figure 7.4. These charts compare results obtained with the old non-vectorized code (blue) with the new vectorized one (green). Sometimes a very well optimized sequential code impacts application scalability on multi-core architectures. This happens because the available parallelism of the application decreases when the code is vectorized. Memory bandwidth capacity can be easily exceeded when multiple threads issue vector instructions that load and store multiple data elements. Similar behavior is observed in IR. The vectorized and multithreaded IR is faster (takes less time) but the speedup factor is lower than the non-vectorized multithreaded one. This is also true for the simultaneous multithreaded version of IR. Please note that for comparison reasons, speedups are calculated by taking into consideration the respective elapsed time of each of the best sequential version (vectorized and non-vectorized).



	Sequential	8 Threads	16 Threads SMT
		OpenMP	
🟦 Non-Vectorized IR	733.5	122.5	100.7
🟩 Vectorized IR	537.41	103.06	84.36



	Sequential	8 Threads	16 Threads SMT
		OpenMP	
🟦 Non-Vectorized IR	1	5.99	7.28
🟩 Vectorized IR	1	5.22	6.37

Figure 7.4: Parallel large scale IR on the LRZ ccNUMA subsystem

The main lesson learned is that parallel performance of some applications can be improved by vectorization techniques, but one should estimate earlier needed efforts to apply vectorization to the application. If the code can be easily auto-vectorized by the compiler, every improvement obtained is acceptable as the programmer does not have to spend so much time vectorizing the application. However, if the application has to be manually vectorized by using specific intrinsic instructions, the programmer has to estimate if it is feasible to do this for the current application. Maybe it is possible to obtain most of the performance by just multithreading and optimizing it for parallel execution.

7.2.2 Porting IR to CUDA C

Next sections describe different IR implementations for heterogeneous systems based on CUDA accelerator modules. Furthermore, they discuss results obtained on benchmarked platforms available at LRZ and BSC. CUDA C and OmpSs programming models are introduced in subsection 2.3.2 on page 16. The LRZ heterogeneous platform is mainly used to discuss performance results obtained with the CUDA C IR implementation, whereas the BSC platform is used to compare results obtained with the OmpSs IR implementation to the CUDA C IR ones. OmpSs discussions focus mainly on obtained performance and programmer's productivity.

To harness the computational power of the GPU and to make better use of the resources, the 2-dimentional data that has to be processed by each kernel on the GPU is partitioned in square tiles. Indeed, this is a natural choice given the two-dimensional nature of SAR processing and the fact that data is relatively large (refer to Table 4.3 on page 65). In this way, all data elements of a tile are computed by a block of threads. Tiling technique increases the number of active blocks in the grid by increasing so the level of occupancy. Data is partitioned into square tiles of size *tsize* = 32 (each tile is composed of 32 x 32 = 1024 data elements). While keeping the size of each tile small, the total number of threads in each block is kept under 1024. This is the maximal allowable block size. Each thread in each block calculates one data element. It uses its *block index* value to identify the tile that contains the data element before it uses its *thread index* value to identify the element inside the tile. In other words, each thread uses both *thread* and *block indexes* to identify the data element to work on. In *x* and *y* coordinates, *thread indexes* are abbreviated as *tx* and *ty* and *block indexes* are abbreviated as *bx* and *by*. The logical two-dimensional tiling is portrayed in Figure 7.5 where *bx*, *by*, *tx*, and *ty* values are marked in both *x* and *y* dimensions.

All threads computing on data elements within a tile have the same *block index* (*bx* and *by*) values. Two-dimensional data is divided in each dimension into sections of *tsize* elements each. Each block handles such a section. Therefore, each thread can find *x* and *y* index values of the element it is computing by calculating *block index * tsize + thread ind*ex. Hence, thread (*tx*, *ty*) in block (*bx*, *by*) computes the element residing at row *by*tsize+ty* and column *bx*tsize+tx* of the data set. Each thread inside the block is indexed using variables *tx* and *ty*, which take values from *0* to *tsize-1*. The number of blocks in each dimension (*x* and *y*) depends on the respective data size at the respective processing step of IR. To build a more flexible application, two other variables (*size_x* and *size_y*) are introduced to help in calculating grid size in each dimension. If data size in each dimension is divisible by *tsize*, grid size in the respective dimension is simply calculated by dividing it with *tsize*. Otherwise, grid sizes have to be calculated by using extended *width* and *height* values (*size_x* and *size_y*). These extended values are obtained by padding the original ones so that values become divisible by *tsize*.

Indeed, two-dimensional tiling technique is easily implemented in most processing steps of IR. Some processing steps that involve FFT transforms make very good use of functions

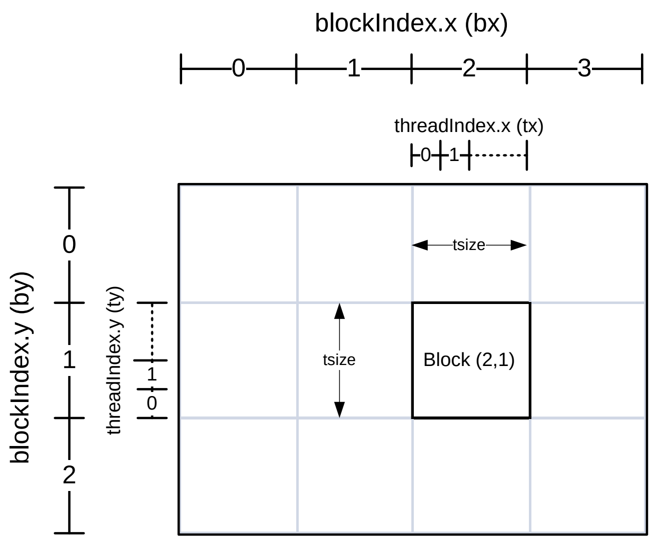


Figure 7.5: Two-Dimensional Multi-Block Tiling

provided by CUFFT library. This library provides a simple interface to compute parallel FFTs on an NVIDIA GPU. It allows users to leverage the floating-point power and parallelism of the GPU without having to develop a custom GPU-based FFT implementation. FFT libraries typically vary in terms of supported transform sizes and data types. CUFFT API provides a much more flexible solution than the CPU FFTW one for the parallelization of one-dimensional FFT transforms, especially when many of the same type have to be executed. CUFFT implements batch execution for multiple one-dimensional transforms in parallel and hides the underlying details of the implementation. The only drawback is the amount of used memory on the host side, which increases with the size of the transform and the number of the configured transforms in one batch execution.

CUDA C provides support for complex data with build-in data types and operations on those data types, but it lacks some complex operations like the multiplication of two complex numbers, multiplication of a complex number by a scalar, and complex exponential operations. Missing operations are manually implemented because they are widely used in IR. Some specific atomic operations are implemented too. CUDA provides a lot of support for atomic operations but not for *float* or *complex float* data. An *atomic-add* function for float data types is constructed by using the *atomicCAS* (Compare and Swap) operation. The most appropriate way to implement atomic operations for complex data, is to first extract real and imaginary parts of the complex value and then to use the corresponding atomic operation on these two floating-point variables.

Data conflicts in threads are actually easy to avoid in CUDA. An atomic operation is able to read, modify, and write back a memory location without the interference of any other thread. This guarantees that a race condition will never occur. Atomic operations in CUDA work for both shared and global memory locations. In shared memory, they are used to prevent race conditions between different threads within the same thread block. In global memory, they are used to prevent race conditions between two different threads regardless of which thread block they are in. Please note that shared memory is much faster than global memory. Even though atomic operations are sometimes necessary in algorithms, it is important to avoid their usage when possible, especially with global memory accesses. If at the same time two threads perform an atomic operation over the same memory location, those operations will be serialized. The order in which operations complete is unknown, which is acceptable, but the serialization can sometimes be quite costly. So whenever atomic operations are used, one should try to avoid serialization that results in a loss of parallelism, and thus a loss in performance. Nevertheless, when atomic operations are used correctly, they are extremely useful.

To obtain better performance in some processing steps of IR, transcendental instructions such as *sine*, *cosine,* and *square root* are used. Such instructions are efficiently executed on Special Function Units (SFUs). In Fermi GPUs, each SM contains 4 SFUs for single-precision floating-point transcendental functions. Each SFU executes one instruction per clock for each thread. Therefore, a warp executes over eight clocks. SFU pipeline is separated from the dispatch unit allowing the dispatch unit to issue instructions to other execution units. This improves application performance by running other instructions while SFUs are occupied.

7.2.3 Performance Optimizations and Results on the LRZ Heterogeneous Platform

Application behavior is observed for different image reconstruction scales. The main interest is in large scale image processing that needs high performance computing to fulfill time requirements. However, small scale image reconstruction is used as a starting point for results and discussions since the data set fits completely in the GPU memory. Once the data has been sent to the GPU in the beginning phase, the GPU computes all steps consecutively and sends the result back to the CPU in the end. Even when transferring data to and from the GPU, the overhead is not so high because the amount of data being transferred is minimal in this case. The IR code with GPU kernels is initially implemented for small scale image reconstruction, applying all techniques from previously discussed points, and then the same code is updated to work also for larger scales.

Besides, the data set for large scale image reconstruction does not fit in the GPU memory, especially in the Interpolation Loop where two separate sets of data have to be allocated, one for the input and one for the output data of that step. Table 7.2 shows that the total amount of memory needed for large scale Interpolation Loop on the GPU board reaches 7.54 GB. As previously discussed, on the benchmarked platforms there is a total of 6 GB global GPU memory. When ECC is enabled, which for space applications should be, the available memory shrinks to 5.25 GB. To fulfill the main requirement (strong scaling) that SAR applications ask from high performance platforms, the communication pattern in the heterogeneous system has to be carefully analyzed. Frequent CPU-GPU communications impact application scalability, changing so the efficiency rate in terms of computations per power consumption. Facing the problem of not being able to fit the data set into the GPU memory, requires the implementation of an efficient communication pattern that if possible overlaps with computations.

The resulting application should exploit the fine grain parallelism on the GPU, even with limited memory resources. Taking into consideration the access pattern over the arrays

Table 7.2: Memory Allocation in the Interpolation Loop for Large Scale IR

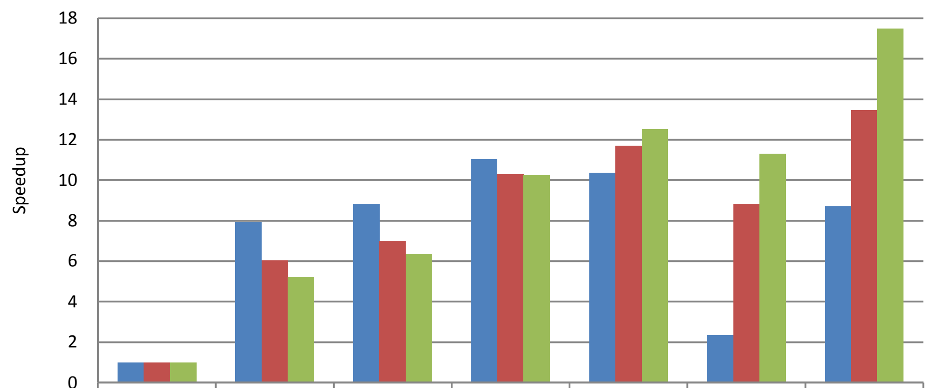
Arrays	Number of Elements	Size in Giga Bytes
complex float in_img[n][m]	437234160	3.26
float idx[n][m]	437234160	1.63
complex float out_img[nx][m]	336674872	2.5
Total		7.54

in the Interpolation Loop, array *out_img* has to be completely allocated in the GPU memory. This array contains the output image of this step. Accesses to it are non-regular because access indexes are calculated out of the content of respective locations in the *idx* array. This indirect addressing complicates the access pattern over array *out_img* by having different threads jumping to and from different locations. It is impossible to know on which part of array *out_img* these jumping locations will be. This is the reason that this array is completely allocated in the GPU memory after the required space is freed. The other two arrays (*in_img* and *idx*) reside in the GPU memory before entering Interpolation Loop because the data set fits into it until this step. To have a lower number of transfers, only *in_img* array is moved to the CPU memory while *idx* is kept in the GPU memory.

In_img array is moved to the CPU memory and the respective space is freed to create space for the new output array *out_img*. After allocating array *out_img* in the GPU memory, a space of 1.63 (*idx*) + 2.5 (*out_img*) = 4.13 GB is used out of 5.25 GB total available memory. There is still 5.25 - 4.13 = 1.12 GB free available memory on the accelerator module. Since only a small amount of data can fit in it, the array *in_img* is tiled into 4 same-sized chunks of rows, each using 815 MB, so that it can be copied one chunk at a time to the GPU memory. When each chunk is copied to the GPU, the Interpolation Loop kernel is called with respective parameters and pointers to the respective parts of input and output data. The output array *out_img* stays allocated in GPU memory for the rest of the processing steps coming after Interpolation Loop. In this way, frequent data movements between devices are reduced to avoid bandwidth problems in the PCIe link.

Another important point to discuss here is again related to the unpredictable access pattern over array *out_img* in the Interpolation Loop. In a fine grain parallelism environment, like the one in the GPU, caution should be exercised to ensure data integrity. While running the Interpolation Loop, multiple threads try to modify simultaneously same memory locations of array *out_img*. Atomic operations are used to ensure data integrity in the GPU environment with fine grain CUDA threads. Such operations perform a read-modify-write atomic operation on a 32-bit or 64-bit word residing in global or shared memory. These operations impact application performance on the GPU. However, obtained results (Figure 7.6) show that the execution on the GPU is around 10 times faster than the execution of the best sequential version on the CPU. When comparing the best multithreaded CPU version with the GPU version, one can say that the speedup almost doubles on the GPU.

Figure 7.6 illustrates speedups obtained on the LRZ heterogeneous platform with different implementations. The group of results marked *GPU* represents the speedup obtained when using only the GPU device for computations. Other implementations are discussed in following paragraphs. Please take into consideration that Figure 7.6 shows the speedup obtained for different image scales. The data set for small scale image reconstruction fits completely in the GPU memory, whereas for large scale image reconstruction it does not.



	CPU Sequential	CPU 8 Threads	CPU 16 Threads (SMT)	GPU	CPU + GPU	2 GPUs	2 GPUs Pipelined
🟦 Scale=10	1	7.95	8.82	11.05	10.37	2.35	8.71
🟥 Scale=30	1	6.04	7.01	10.29	11.70	8.84	13.44
🟩 Scale=60	1	5.21	6.37	10.24	12.53	11.30	17.49

Figure 7.6: IR Speedup on the LRZ Heterogeneous Platform

Computations in a heterogeneous system can be scheduled only on CPU, only on GPU, or on both CPU and GPU. In the latter case, the application will make use of the processing power provided by each computing component, but the programmer should try to avoid transfers between them. To achieve this, the programmer has to parallelize the application according to a different model taking into consideration many factors like: data dependencies, scheduling algorithms, and system resources such as the available GPU memory. Depending on the type of computation, some applications might be suitable for CPU processing, GPU processing, or heterogeneous CPU/GPU processing. Even within parts of the same application, some independent processing steps might show this behavior. It is programmer's responsibility to identify this situation on different applications or even among different parts of the same one.

Following features can be identified in the case of IR. For all processing steps, except Interpolation Loop, it is not feasible to distribute work between CPU and GPU due to the low ratio of computations over predicted communications. IR is expected to show communication overhead in *transposing* and *fft-shifting* stages between those processing steps. This problem is first identified when distributing workload into multiple CPUs in a distributed cluster environment and something similar will probably happen in a heterogeneous system due to the low bandwidth capacity of PCIe. Another fact is that all those processing steps take only 30 % (Interpolation Loop takes 70 %) of the total execution time on the CPU. Considering this, a new version of IR is implemented, in which the Interpolation Loop is distributed among CPU and GPU. The *GPU + CPU* group of results in Figure 7.6 shows that this does not make sense for small scale image processing. The reasons for this are:

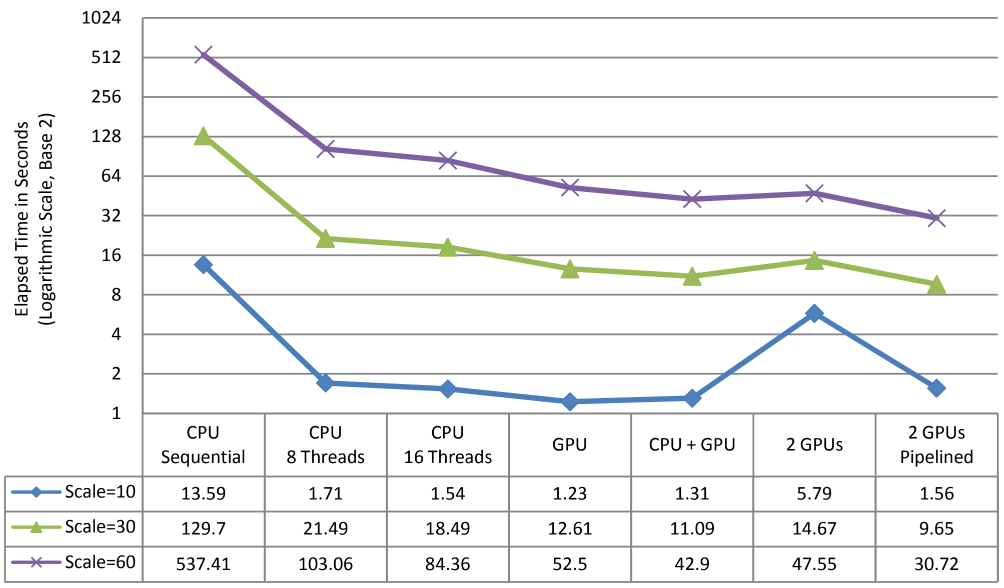
1. The lack of intensive computations,
2. The higher impact of the communication overhead in the overall performance, and
3. The reduction needed after the Interpolation Loop to collect results into one device (in the IR case into the GPU).

Defining the amount of work that is to be distributed among host and device is not a trivial job, since they exhibit very different features and processing capabilities. Taking into consideration the fact that the GPU is twice faster than the CPUs (2 CPUs by 4 cores), 3/4-th of the data is sent to the GPU for processing and 1/4-th is kept on the CPUs. Other distributions are also tested, but this one proves to be the most balanced. The CPU part itself has to be split again in three subparts, so that each subpart is scheduled to be processed in the same time with the corresponding one on the GPU. The time needed to reconstruct a large image is now only 43 seconds, which is 12.5 times faster (Figure 7.6) than the best sequential CPU execution time of around 9 minutes. One of the main issues that should be addressed in heterogeneous computing is load balancing. For the specific IR application with fixed computational requirements, the static approach provides the required level of load balancing. However, other applications might require more flexible solutions that can be provided only by a dynamic load balancing approach.

To further improve performance and to further decrease time needed to reconstruct the SAR image, the second GPU device on the accelerator module is used. Two OpenMP threads are created to provide separate contexts for each GPU device. This is implemented through independent calls to CUDA memory management functions and to CUDA kernels from each thread, once in the parallel region. To harness the computational power of both GPUs, two different approaches are taken. At first, the work for the reconstruction of the same image is distributed among two devices. This approach does not further improve results. Indeed, the performance gets even worse compared to the single device version.

As shown in Figure 7.6, the speedup goes down from 12.5 (in the *GPU + CPU* version) to 11.3 (in *2 GPUs* version). The reason for this is the PCIe link bottleneck combined with the overloaded QPI link between two CPUs and between CPUs and corresponding memories. In this version (*2 GPUs*), data is first copied from one GPU to the CPU memory and from there to the other GPU, in some steps in which both GPUs need to exchange data between them. To avoid such communication overhead, another approach is taken when implementing the *Pipelined* version. Now, separate images get reconstructed on separate GPU devices. Having available memory resources on each GPU device allows the implementation of the pipelined version on this heterogeneous system. It is not possible to implement it on one GPU or even on one CPU system with limited memory resources because the double amount of memory is consumed during execution. In the pipelined scenario, raw data is distributed in the initial phase from each CPU to each GPU and the reconstructed images are collected back only in the end.

By avoiding inter-GPU communication, each large scale image is reconstructed in only 30.7 seconds obtaining so a speedup of 17.49 (*2 GPUs Pipelined* version in Figure 7.6). Again, as in the case of the *CPU + GPU* version, it is not worth to use multiple devices for the reconstruction of small scale images. Figure 7.7 summarizes the tendency of the elapsed time throughout different optimization levels that have been applied to IR for heterogeneous systems. For large scale image reconstruction the elapsed time goes always down. This is not always true for the small scale one, especially when using 2 GPUs. The communication overhead between devices impacts performance so much that it slows down the application, even when compared to parallel versions of IR on CPUs.



	CPU Sequential	CPU 8 Threads	CPU 16 Threads	GPU	CPU + GPU	2 GPUs	2 GPUs Pipelined
◆ Scale=10	13.59	1.71	1.54	1.23	1.31	5.79	1.56
▲ Scale=30	129.7	21.49	18.49	12.61	11.09	14.67	9.65
× Scale=60	537.41	103.06	84.36	52.5	42.9	47.55	30.72

Figure 7.7: IR Elapsed Time on the LRZ Heterogeneous Platform

7.2.4 Image Reconstruction on BSC Heterogeneous Platform with OmpSs

OmpSs programming model is considered to improve programming productivity in heterogeneous systems. It gives the possibility to implement a task based application, portable to different heterogeneous systems by allowing different task implementations. Data movements inside the system are carried out automatically by the runtime based on detected data dependencies. This simplifies the programmer's assignment, which in the case of OmpSs has to take care only of task implementations and data direction specifications.

To port IR to the BSC Heterogeneous Platform with NVIDIA GPUs, the already implemented CUDA kernels are embedded in OmpSs tasks. They are specified by the *target* directive as code that should be executed on the GPU. Then, tasks responsible for data management and kernel calling are specified. No modifications have to be applied to the CUDA kernels and what is more important no explicit allocation and data movement to and from the GPU memory is required. Everything is taken care automatically during runtime. It is only required to specify data directions (input, output, and inout) to each task, so that dependencies can get detected at runtime.

Recall that for small scale image reconstruction the data set fits into GPU memory. Only one task is needed to copy necessary data to the GPU and to start calling all kernels one by one. In the end the resulting image is copied back to CPU memory. The only problem for small scale image reconstruction with OmpSs is that the only way to allocate memory on the GPU is to copy data from CPU memory. This means that when temporal variables have to be allocated in GPU memory, respective copies have to be allocated first in CPU memory and then this data has to be copied to GPU memory. This brings some overhead compared to the plain CUDA version, in which CUDA API calls are used to allocate memory directly in GPU memory.

For large scale image reconstruction, IR processing steps are split in three tasks because it is necessary to have the Interpolation Loop (Step 10 in the IR substage) in a separate task. In this way, all steps before Interpolation Loop are included in the first task, Interpolation Loop in the second task, and steps after Interpolation Loop in the third task. Indeed, Interpolation Loop has to be called by a separate task because in this step the data set does not fit in GPU memory. Data has to be sent by chunks to the GPU so that it can be processed one chunk at a time. Furthermore, the GPU memory has to be freed before entering Interpolation Loop. This is achieved by finishing the first task and starting the second one before entering Interpolation Loop. Some additional overhead is introduced because data is copied from GPU memory to main memory when the first task finishes and then, when the second task is started some of that data is copied again from main memory to GPU memory. Something similar happens when finishing the second task and starting the third one. Some options that are still in development for this version of OmpSs that might help to overcome this situation are:

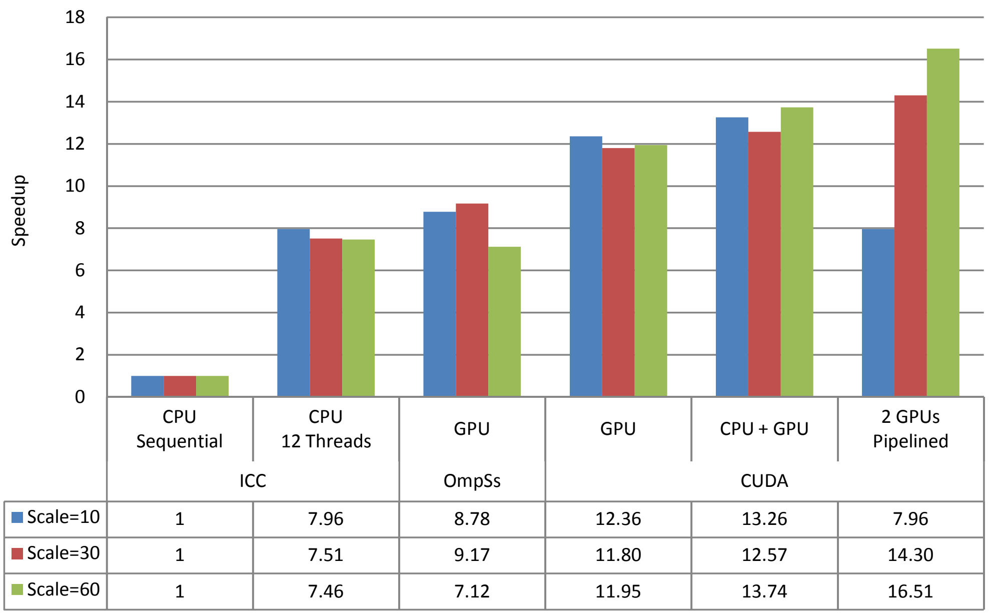
1. The *taskwait noflush* directive. If the *noflush* clause is used in the taskwait directive when finishing one task, then data will not be copied to main memory.
2. The *taskwait on data* directive. If it is possible to wait on specific data, only that data will be copied to main memory when the task is finished.

Obtained results when applying OmpSs to IR on the BSC heterogeneous platform are shown in Figure 7.8. Respective speedups are calculated in relation to the best sequential execution time on the CPU. Since CPUs in the heterogeneous platform are Intel ones, the native Intel C compiler generates the most optimal code by auto-vectorizing most of the loops, including the innermost loop of Interpolation Loop.

For performance comparisons, Figure 7.8 shows in regard to image reconstruction of different scales:

1. The speedup gained when using only CPUs of the platform (2 CPUs by 6 cores each). The OpenMP R&R implementation of IR is used here because it provides better performance than the one using atomic operations for a small number of cores.
2. The speedup gained when using only the GPU device for computational purposes. OmpSs IR implementation that embeds CUDA kernels is compared with the pure CUDA IR implementation.
3. The speedup gained when using different CUDA implementations that harness the computational power of the heterogeneous platform.

When using OmpSs the speedup factor is lower than when using only CUDA due to the overhead coming from memory operations. In the OmpSs IR version a lot of data is copied to and from GPU memory. Such data has to be copied to the GPU when simple allocations are needed and also when one task finishes and another one starts. Leaving aside this drawback of OmpSs, one can say that OmpSs is much more flexible because it increases programmer's productivity by hiding data management tasks to the programmer. Of course, flexibility and productivity have a cost, which in this case is the performance.



	CPU Sequential	CPU 12 Threads	GPU	GPU	CPU + GPU	2 GPUs Pipelined
	ICC		OmpSs	CUDA		
🟦 Scale=10	1	7.96	8.78	12.36	13.26	7.96
🟥 Scale=30	1	7.51	9.17	11.80	12.57	14.30
🟩 Scale=60	1	7.46	7.12	11.95	13.74	16.51

Figure 7.8: IR on the BSC Heterogeneous Platform (OmpSs vs CUDA)

7.3 Conclusion

Heterogeneous systems are considered in this chapter as the best alternative for the proposed high performance architecture. They provide improved parallel performance over shared memory systems and improved efficiency (in terms of power consumption and size) over distributed memory ones. Regarding programmer's productivity, some additional efforts are needed to port the application for heterogeneous CPU/GPU computing and to tune it for efficient execution on GPU accelerator modules, but that is acceptable since performance improvements are obtained.

8 Conclusions and Future Work

This final chapter presents conclusions of this dissertation and potential future work directions.

8.1 Conclusions

Looking back at the work conducted in the frame of this dissertation, it is always important to evaluate if initial goals have been reached and initial claims have been proved. It is also substantial to asses at which extent of achievement each goal has been reached, and to present reasons that explain these levels of achievements. In addition to undeniable factors that allowed a satisfactory achievement level for some goals, one should state the limiting factors that confined the achievement of other goals. The main objective of introducing high performance computing technologies in the design process of spacecraft computing platforms has been achieved. The reliable high performance architecture proposed in this dissertation is used to achieve this goal. The architecture proposal is logic and sound. It is based on previously proven concepts and novel approaches that have never been combined before in a complete system. The suitability of such novel approaches for future space applications is proved in this dissertation by benchmarking different high performance platforms with a real spotlight SAR spatial frequency interpolation application. Each benchmarked platform can be considered as an alternative solution for the main design goals (Reliability, Portability and Modularity), but they have to be classified according to the extent of achievement regarding other goals like:

1. Efficiency in terms of obtained performance per consumed power and physical size or weight of such platforms,
2. Space applications scalability of on such platforms,
3. The ability of space applications to harness the computational power of such platforms, and
4. Programming productivity in terms of efforts needed to optimize space applications for such platforms.

8.1.1 The Reliable High Performance Architecture

The proposed architecture provides general descriptions and guidelines on how to improve performance of space applications without sacrificing system reliability. The proposal successfully introduces high performance computing components for on-board processing of space applications. To address system reliability, hardware replication techniques have been proposed to be combined with software fault-tolerance ones. The required level of reliability defines the amount of hardware and software resources that can be used for that purpose. The multi-node approach proposed in the architecture enables a flexible hardware and software allocation of such resources. This can be seen as a powerful tool in the hands of each spacecraft computing platform designer. The proposed architecture is not intended for spacecraft control systems, but it is designed as an accelerator module to improve performance of computational demanding applications. Computing platform designers can use the proposed architecture design to implement specific systems that fit to their specific requirements and constraints. Provided flexibility in the proposed architecture enables them to easily decide on:

1. The total number of parallel processing nodes. This number depends not only on parameters like size, weight, and power consumption, but also on the computational power of each node and on application performance scalability.

2. The required hardware redundancy level. The designer can decide whether to use processing nodes in a redundant fashion for reliability purposes or to use all of them only for processing to improve application performance.

3. The amount of resources allocated to software fault-tolerance. If no hardware redundancy is provided, the designer should increase system reliability by using more software resources for fault-tolerance.

8.1.2 Benchmarking High Performance Platforms

In order to further assist spacecraft computing platform designers, different high performance systems have been benchmarked with a real space application. Coming from a family of applications with high computational demands, a two-dimensional spotlight SAR (2DSSAR) application is implemented and optimized for efficient parallel execution on these systems. The main purpose of such application is to reconstruct a two-dimensional image from raw sensor data collected by the radar aperture. In order to mimic different computational requirements, this application provides the possibility to configure the size of the image being reconstructed. In this way, low-to-high resolution image reconstructions can be used for benchmarking. The amount of computations and memory space needed for the reconstruction increases while increasing the image size (resolution). The implemented application can also be used to benchmark computing systems for a broad range of military and commercial image processing applications where data is acquired and processed in one order, and later retrieved and processed in a different order.

Parallel performance of 2DSSAR application is evaluated on different high performance platforms implemented as shared memory systems, distributed memory systems, and heterogeneous CPU/GPU systems. In each of them, three different image scales (small, medium, and large scale) are used in the evaluation. The main focus was on parallelization techniques used in each system and the obtained performance in regard to time needed for SAR image reconstruction. The goal was to find out, which system provides the most scalable and efficient performance in terms of power consumption and system size.

8.1.2.1 2DSSAR on Shared Memory Platforms

First results are obtained on two shared memory platforms, one ccNUMA and one UMA SMP. 2DSSAR application is first profiled so that most interesting and most computationally demanding parts can be identified. The incremental approach taken to apply different parallelization techniques allows seeing the impact of each technique in the overall application performance. Shared memory programming with OpenMP and OmpSs, distributed memory programming with MPI, and hybrid programming with MPI and OpenMP have been used to port 2DSSAR to shared memory platforms. Distributed memory and hybrid programming paradigms appear to be suitable for ccNUMA platforms with physically distributed but logically shared memory. Such paradigms improve performance by reducing accesses to physically remote memories.

They are not so relevant for UMA SMP platforms, where the concept of remote memory access does not stand. For such platforms it is very important to efficiently use threads for parallel work without overwhelming the bandwidth capacity of the only memory link that is shared among all threads. This is the reason that better performance is obtained when parallelizing 2DSSAR with OmpSs programming model for SMP platforms. The task based OmpSs increases load balancing by allowing a task to continue execution as soon as its data is available. Regarding programmer's productivity, minor efforts were needed to port 2DSSAR to shared memory platforms with OpenMP and OmpSs. When using distributed memory and hybrid programming, additional efforts were needed to make 2DSSAR run efficiently without wasting computing resources.

Specific attention is paid to data integrity in shared memory environments. Two essential techniques have been examined, implemented, and evaluated. *Replication and Reduction* technique provides better performance than *Atomic Operations* when scaling up to a small number of cores (threads). Atomic operations are expected to scale better than replication and reduction when using much more cores than 12 or 24 that were available on benchmarked platforms. This is due to the replication and reduction overhead, which increases with the number of cores being used. When SMT is enabled on the platform, performance of the application with atomic operations is improved, but still the application is not faster than when using replication and reduction technique. Application scalability is inspected on a simulated multiprocessor machine with up to 512 cores. As expected, the application with atomic operations scales better than the one with replication and reduction.

Results obtained on real platforms show that 2DSSAR is scalable to 12 and 24 processor cores in shared memory platforms since all implementations show good speedup values. The time needed to reconstruct a large scale (high resolution) SAR image is reduced from more than 15 minutes to around 1 minute and 30 seconds. Taking into consideration that such systems are basically small-sized multi-socket single nodes with not so high power consumption, one can say that 2DSSAR executes efficiently on shared memory platforms. Nevertheless, performance will always be limited from computational resources on the shared memory platform being used.

8.1.2.2 2DSSAR on Distributed Memory Platforms

Distributed memory platforms have been benchmarked as an alternative to further improve performance of the 2DSSAR application. The goal was to further reduce the time needed for image reconstruction by efficiently using resources of such platforms. Focusing on efforts needed to achieve that, different parallelization techniques and optimizations for distributed memory systems have been discussed for and implemented in 2DSSAR application. The transition from shared memory to distributed memory programming is definitely not an easy step due to the fact that in distributed memory programming data distribution has to be explicitly instructed additionally to work distribution.

In order to simplify this transition process and to solve memory insufficiency problems in distributed memory platforms, a hybrid MPI + OpenMP 2DSSAR application is implemented. There are also other advantages of hybrid programming that supported this decision. The simplified communication pattern in hybrid programming improves application scalability, which compared to the pure MPI version can be programmed with much less efforts. Additional optimizations were needed to make 2DSSAR exploit more efficiently computing resources on the platform. Summing up all efforts needed to achieve this, it is obvious that programmer's productivity decreases when programming for distributed memory systems compared to shared memory ones.

Regarding performance, with the initial hybrid 2DSSAR version, time needed for large scale image reconstruction is reduced from 10 minutes and 39 seconds to 37 seconds when using 8 processor nodes in a distributed memory cluster. Taking into consideration that each node has 8 processor cores in a ccNUMA architecture, the speedup value of 17.7 on 64 cores represents only 26.7 % of the theoretical linear speedup. This version of 2DSSAR is not efficiently using resources in the distributed memory platform. To increase efficiency, communication and synchronization patterns among distributed processes are revised and updated. With the new version of 2DSSAR on 8 processor nodes, time needed for large scale image reconstruction is reduced to 21.5 seconds giving the most efficient speedup factor of 27 that represents 42 % of the theoretical linear speedup. On 16 nodes, the elapsed time is further reduced to 18 seconds, but the speedup factor of 32.3 represents only 25.2 % of the theoretical linear speedup. These results conclude that application performance improves when using more nodes, but the efficiency of using resources does not.

Another version of 2DSSAR application is implemented to increase resource usage efficiency. It uses a pipelining technique to exploit the embarrassing parallelism, which is present in a long run of 2DSSAR application. In an iterative image reconstruction scenario, different image reconstructions are pipelined along different nodes in the distributed platform. With this version, a speedup factor of 60 is obtained when using 128 cores in 16 nodes and the average image reconstruction time is reduced to 10.7 seconds. This means that the distributed memory system is used with 47 % efficiency. Being able to reconstruct an image in around 11 seconds can be considered a success, but when taking into consideration size (number of nodes being used) and power consumption, one has to search for much more efficient solutions.

8.1.2.3 2DSSAR on Heterogeneous CPU/GPU Platforms

Heterogeneous CPU/GPU systems are considered as an efficient alternative for SAR image reconstruction. Before starting to work on the GPU version of 2DSSAR application, CPU sequential and parallel ones are first optimized for vector processing. A speedup factor of around 3 is obtained only through sequential optimizations. When comparing respective parallel performance of the optimized sequential code with the original one, it is apparent that the optimized code asks for less image reconstruction time, whereas the original one obtains higher speedup values. This happens because a well optimized sequential code impacts application scalability on parallel platforms.

Being focused mainly on efforts needed to port 2DSSAR application to heterogeneous systems with NVIDIA GPUs, approaches taken to parallelize and optimize this application are examined and applied. Different programming paradigms like CUDA and OmpSs are used, and respective performance optimizations and results are discussed. It turns out that CUDA is able to provide better performance, whereas OmpSs improves programmer's productivity.

Regarding performance, various CUDA versions of 2DSSAR application are implemented for heterogeneous computing and compared with sequential and parallel CPU versions. Large scale image reconstruction time is reduced from around 9 minutes (Sequential on CPU) to 52.5 seconds when using only one GPU device, to 43 seconds when using both CPU and GPU, and to 31 seconds when using 2 GPUs (with pipelined image reconstructions). When comparing these results with previous ones, it is understandable that large scale image reconstruction is about 3 times faster on the heterogeneous platform than on the shared memory one, but 3 times slower than on the distributed memory platform. However, it is relevant to say that 2DSSAR uses resources more efficiently on the heterogeneous platform. At least 4 processor nodes (32 cores) have to be used in the distributed memory platform to obtain a comparable result.

From programmer's productivity point of view, it was not so easy to port and optimize 2DSSAR for optimal performance on the heterogeneous platform compared to efforts needed to improve its scalability on the distributed memory platform. For other applications that do not suit for GPU computing, much more efforts might be needed to obtain optimal performance, and sometimes it might not be possible to obtain that performance at all. In such cases, the CPU-based alternatives might be the preferred choice. However, SAR processing is suitable for heterogeneous computing and can exploit fine grain parallelism on the GPU. Indeed, such heterogeneous CPU/GPU systems provide the best solution for the proposed high performance architecture. Fine grain accelerator modules have to be integrated on future on-board computing architectures, in order to achieve expected computational power without drastically increasing power consumption, size, and weight.

8.2 Future Work

The work started by this dissertation can be further continued in many directions. They are summarized in the following subsections.

8.2.1 Simulating and Prototyping the Proposed Computing Architecture

One of the most interesting direction to work on in the future is the simulation of the proposed reliable high performance architecture. Such a simulation will probably be implemented on reconfigurable FPGA platforms. The goal of this simulation will be to create a prototype design, which can later be used by spacecraft computing platform designers. Taking into consideration features of the proposed architecture, multiple FPGA modules will have to be interfaced by an interconnection system. Concrete interconnection technologies will be benchmarked too. Everything will be managed by a controlling subsystem, probably implemented through a radiation hardened processor system.

The prototype will consist of multiple IP Cores implemented in a hardware description language, which can be VHDL or Verilog. Such languages provide the possibility to design, simulate, debug, and verify integrated circuits. This will allow to simulate and verify proposed solutions that will provide system reliability, including here hardware redundancy and software fault-tolerance techniques. The focus will not be on the internal implementation of each parallel processing node in the architecture, but on the overall system logic representation and its components. After obtaining the simulation platform, with or without implemented hardware redundancy, the research for software fault-tolerance can begin. Various fault-tolerance techniques can be implemented and verified for correctness on the simulation platform. Specific fault-tolerance techniques can also be proposed and researched as part of the reliable architecture simulation.

8.2.2 Porting 2DSSAR to Modern High Performance Systems

It would be interesting to port 2DSSAR application to other modern computing platforms that will be available in the future. Such platforms may integrate a heterogeneous CPU/GPU system on chip, which is a technology being currently under development at AMD. It may be a shared memory platform based on Intel MIC architecture that integrates many cores in a single chip. Other many-core systems on chip like the ones provided by Tilera can be benchmarked in the future. They provide a distributed memory platform on chip. The Maestro processor, which is currently being developed by NASA, is based on the Tile64 microarchitecture from Tilera. Other novel CPU and GPU technologies will be developed in the near future to address high performance computing challenges like power consumption and scalability. Reconfigurable computing is also an interesting research direction for future space applications. It can also be seen as an alternative to improve reliability of the proposed architecture as it allows dynamic runtime reconfiguration for fault masking and recovery.

In the direction of heterogeneous computing, an OpenCL implementation of 2DSSAR will make it possible to benchmark various CPU/GPU-based heterogeneous systems. The OpenCL 2DSSAR implementation will broaden the spectrum of platforms that can be benchmarked. Additional to NVIDIA GPUs, AMD GPUs and other processing modules being based on DSP processors and FPGAs will have the possibility to be benchmarked with the OpenCL 2DSSAR version. Already implemented 2DSSAR versions and those that will be implemented in the future can also be used to benchmark systems intended for other kinds of applications as medical, astronomical, and reconnaissance imaging.

8.2.3 Benchmarking with Other Novel Space Applications

Even though 2DSSAR application represents and mimics computational demands and challenges of a wide range of space applications, it still implements some specific processing algorithms that are present only in SAR applications. As discussed in section 2.7 on page 30, the big family of current and future applications includes also various applications like CRE, HSI, HRSW SAR, STAP, GMTI, LIDAR, DDC, and PDFE.

It will be very interesting to determine how such applications behave on high performance platforms and to identify features that make them specific in their purpose. This will also enable their profiling according to suitability for different platforms. Some of them might be suitable for shared memory CPU computing and others for distributed memory CPU computing. Probably some of them may be capable to harness the GPU computational power. Using other applications for benchmarking purposes requires their parallelization and optimization. Further experience will be gained as part of the work to port such applications to different computing platforms. Exposing a variety of communication and computation features, these applications will have to be optimized in different ways. Different approaches will have to be taken to provide the specific required solution.

List of Abbreviations

2DSSAR	Two-Dimensional Spotlight Synthetic Aperture Radar
ABFT	Algorithm Based Fault-Tolerance
ADC	Analog-to-Digital Converter
AGGA	Advanced GPS/Galileo ASIC
ALU	Arithmetic and Logic Unit
AMBA AHB	Advanced Microcontroller Bus Architecture High Performance Bus
API	Application Programming Interface
APU	Accelerator Processing Unit
ASI	Application Specific Interface
ASIC	Application Specific Integrated Circuit
BAE	British Aerospace Electronics
BSC	Barcelona Supercomputing Center
CABS	Complex Absolute
CCD	Charge Coupled Device
CCSDS	Consultative Committee on Space Data System
CEXP	Complex Exponential
CMOS	Complementary Metal Oxide Semiconductor
CMP	Chip Multi Processor
CMP	Chip Multi Processor
COTS	Commercial Off The Shelf
CPCIe	CompactPCI Express
CPU	Central Processing Unit

CRC	Cyclic Redundancy Codes
CRE	Cosmic Ray Elimination
CRTR	Chip-level Redundant Threading with Recovery
CUDA	Compute Unified Device Architecture
DDC	Digital Down Conversion
DDR	Double Data Rate
DM	Dependable Multiprocessor
DMA	Direct Memory Access
DSP	Digital Signal Processing
ECC	Error Correcting Codes
EDAC	Error Detection and Correction
ESA	European Space Agency
FFT	Fast Fourier Transform
FLOPS	Floating-Point Operations per Second
FPGA	Field Programmable Gate Array
FPU	Floating Point Unit
FSB	Front Side Bus
FTL	Fault-Tolerance Layer
FTM	Fault-Tolerance Manager
FTMP	Fault-Tolerant Multiprocessor
FTP	Fault-Tolerant Processor
FTPP	Fault-Tolerant Parallel Processor
GB	Gigabyte
Gbit/s	Gigabits per second
GCC	GNU C Compiler
GDB	GNU Debugger
GDDR	Graphic Double Data Rate

GMTI	Ground Moving Target Indicator
GNSS	Global Navigation Satellite System
GPROF	GNU Profiler
GPU	Graphics Processing Unit
HLRS	HöchstLeistungsRechenzentrum Stuttgart (High Performance Computing Center Stuttgart)
HPC	High Performance Computing
HRWS	High Resolution Wide Swath
HSI	Hyper-spectral Imaging
I/O	Input Output
IC	Integrated Circuit
ICC	Intel C Compiler
ILP	Instruction Level Parallelism
IMC	Integrated Memory Controller
IP	Intellectual Property
IR	Image Reconstruction
ISS	International Space Station
IU	Integer Unit
JPL	Jet Propulsion Laboratory
KB	Kilobyte
LEO	Low Earth Orbit
LIDAR	Airborne Light Detection and Ranging
LRR	Lehrstuhl fur Rechnertechnik und Rechnerorganisation
LRZ	Leibniz Rechenzentrum (Leibniz Supercomputing Center)
LVDS	Low Voltage Differential Signaling
MAC	Multiply-Acumulate
MB	Megabyte

Mbit	Megabit
MBU	Multiple Bit Upset
MCC	Mercurium Compiler
MCP	Multi/Many-Core Processor
MDPA	Multi-DSP/micro-Processor Architecture
MDT	Message Delivery Time
MIC	Many Integrated Core
MPI	Message Passing Interface
MS	Message Size
MTT	Message Transmission Time
NASA	National Aeronautics and Space Administration
NUMA	Non-Uniform Memory Access
NVP	N-Version Programming
OAO	Orbiting Astronomical Observatory
OmpSs	OpenMP Superscalar
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
PAPI	Performance Application Programming Interface
PCIe	Peripheral Component Interconnect Express
PDFE	Probability Distribution Function Estimation
PID	Process Identification Number
PPN	Parallel Processing Node
PRF	Pulse Repetition Frequency
QPI	QuickPath Interconnect
R&R	Replication and Reduction
RHMU	Radiation Hardened Management Unit
RHPU	Radiation Hardened Processing Unit

SAR	Synthetic Aperture Radar
SCALASCA	Scalable Performance Analysis of LArge SCale Application
SCC	Single-chip Cloud Computer
SDG	Synthetic Data Generation
SDRAM	Synchronous Dynamic Random Access Memory
SEE	Single Event Effects (SEU + SET)
SEFI	Single Event Functional Interrupt
SEL	Single Event Latch-up
SET	Single Event Transient
SEU	Single Event Upset
SFI	Spatial Frequency Interpolation
SFU	Special Function Unit
SIMD	Simple Instruction Multiple Data
SM	Streaming Multiprocessor
SMP	Symmetric Multiprocessing
SMT	Simultaneous Multithreading
SOI	Silicon-On-Insulator
SRT	Simultaneously and Redundantly Threaded
SSCA	Scalable Synthetic Application Number
SSP	SAR Sensor Processing
ST	Space Technology
STAP	Space-Time Adaptive Processing
STAR	Self-Testing-and-Repairing
SWIFT	Software Implemented Fault-Tolerance
TDP	Thermal Design Power
TID	Total Ionizing Dose
TLP	Thread Level Parallelism

TMR	Triple Modular Redundancy
TRL	Technology Readiness Level
TUM	Technische Universität München
UMA	Uniform Memory Access
VDG	Voltaire Grid Director
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLIW	Very Long Instruction Word
XPP	eXtreme Processing Platform

Bibliography

[1] F. Kraja and G. Acher, “Using many-core processors to improve the performance of space computing platforms,” in *2011 IEEE Aerospace Conference*, March 2011, pp. 1 –17.

[2] F. Kraja, G. Acher, and A. Bode, “Parallelization techniques for the 2D Fourier matched filtering and interpolation SAR algorithm,” in *2012 IEEE Aerospace Conference*, March 2012, pp. 1 –10.

[3] F. Kraja, A. Murarasu, G. Acher, and A. Bode, “Performance evaluation of SAR image reconstruction on CPUs and GPUs,” in *2012 IEEE Aerospace Conference*, March 2012, pp. 1 –16.

[4] F. Kraja, A. Bode, and X. Martorell, “2D-FMFI SAR Application on HPC Architectures with OmpSs Parallel Programming Model,” in *2012 NASA/ESA Conference on Adaptive Hardware and Systems*, June 2012, pp. 1 –7.

[5] N. H. E. Weste and K. Eshraghian, *Principles of CMOS VLSI Design.*, second edition ed. Addison Wesley, 1994.

[6] B. Barney, “Posix threads programming,” Website, 08 2011. [Online]. Available: https://computing.llnl.gov/tutorials/pthreads/

[7] OpenMP-ARB, “The OpenMP® API specification for parallel programming,” September 2011. [Online]. Available: http://openmp.org/wp/openmp-specifications/

[8] MPI-Forum, “The message passing interface (MPI) standard,” Website. [Online]. Available: http://www.mpi-forum.org/

[9] B. Mutnury, F. Paglia, J. Mobley, G. K. Singh, and R. Bellomio, “QuickPath Interconnect (QPI) design and analysis in high speed servers,” in *19th Conference on Electrical Performance of Electronic Packaging and Systems*, October 2010, pp. 265 –268.

[10] A. Castonguay and Y. Savaria, “Architecture of a Hypertransport Tunnel,” in *Proceedings of 2006 IEEE International Symposium on Circuits and Systems, ISCAS*, May 2006, p. 4.

[11] D. K. Panda and P. Balaji, “Designing high-end computing systems with InfiniBand and 10-Gigabit Ethernet iWARP,” in *2007 IEEE International Conference on Cluster Computing*, September 2007, p. xiv.

[12] T. G. Mattson, R. Van der Wijngaart, and M. Frumkin, “Programming the Intel 80-core network-on-a-chip Terascale Processor,” in *International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2008, pp. 1 –11.

[13] Intel, “Single-Chip Cloud Computer,” Website, February 2012. [Online]. Available: http://techresearch.intel.com/ProjectDetails.aspx?Id=1

[14] I. Wald, “Fast Construction of SAH BVHs on the Intel Many Integrated Core (MIC) Architecture,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 1, pp. 47 –57, January 2012.

[15] I. Choi, M. Zhao, X. Yang, and D. Yeung, “Experience with improving distributed shared cache performance on Tilera’s Tile processor,” *Computer Architecture Letters*, vol. 10, no. 2, pp. 45 –48, February 2011.

[16] Y. Zhang, L. Peng, B. Li, J. K. Peir, and J. Chen, “Architecture comparisons between Nvidia and ATI GPUs: Computation parallelism and data communications,” in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, November 2011, pp. 205 –215.

[17] A. Branover, D. Foley, and M. Steinman, “Amd’s Llano Fusion APU,” *IEEE Micro*, vol. PP, no. 99, p. 1, February 2012.

[18] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts, “A fully integrated multi-CPU, GPU and memory controller 32nm processor,” in *2011 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, February 2011, pp. 264 –266.

[19] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, “OmpSs: a Proposal for Programming Heterogeneous Multi-Core Architectures,” *Parallel Processing Letters*, vol. 21, no. 2, pp. 173 –193, 2011.

[20] NVIDIA, “Cuda Parallel Programming Made Easy,” Website, February 2012. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[21] Khronos-Group, “OpenCL - The open standard for parallel programming of heterogeneous systems,” Website, February 2012. [Online]. Available: http://www.khronos.org/opencl/

[22] E. Ayguadé, R. M. Badia, P. Bellens, D. Cabrera, A. Duran, R. Ferrer, M. González, F. Igual, D. Jiménez-González, and J. Labarta, “Extending OpenMP to Survive the Heterogeneous Multi-Core Era,” *International Journal of Parallel Programming*, vol. 38, no. 5-6, pp. 440 –459, 2010.

[23] C. Constantinescu, “Trends and challenges in VLSI circuit reliability,” *IEEE Micro*, vol. 23, no. 4, pp. 14 –19, July-August 2003.

[24] R. Velazco and F. J. Franco, “Single event effects on digital integrated circuits: Origins and mitigation techniques,” in *IEEE International Symposium on Industrial Electronics, ISIE*, June 2007, pp. 3322 –3327.

[25] H. J. Barnaby, “Total-Ionizing-Dose effects in modern CMOS technologies,” *IEEE Transactions on Nuclear Science*, vol. 53, no. 6, pp. 3103 –3121, December 2006.

[26] E. Petri, S. Saponara, M. Tonarelli, I. Del Corona, L. Fanucci, and P. Terreni, “Mitigating radiation effects on ICs at device and architectural levels: the SpaceWire router case study,” in *ISIE 2007. IEEE International Symposium on Industrial Electronics*, June 2007, pp. 3310 –3315.

[27] A. H. Johnston, G. M. Swift, and B. G. Rax, “Total dose effects in conventional bipolar transistors and linear integrated circuits,” *IEEE Transactions on Nuclear Science*, vol. 41, no. 6, pp. 2427 –2436, December 1994.

[28] J. C. Laprie, “Dependable computing and fault tolerance : Concepts and terminology,” in *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 'Highlights from Twenty-Five Years.'*, June 1995, p. 2.

[29] A. Kellner, H. J. Kolinowitz, and G. Urban, “A novel approach to fault tolerant computing in space systems,” in *2001 IEEE Aerospace Conference*, vol. 3, 2001, pp. 3/1127 –3/1131.

[30] A. Pakstas, I. Schagaev, and J. Zalewski, “Redundancy classification for fault tolerant computer design,” in *2001 IEEE International Conference on Systems, Man, and Cybernetics*, vol. 5, 2001, pp. 3193 –3198.

[31] M. Rausand and A. Hoyland, *System Reliability Theory: Models, Statistical Methods, and Applications*, 2nd ed. Wiley-Interscience, 2003.

[32] A. Arora and S. S. Kulkarni, “Designing masking fault-tolerance via nonmasking fault-tolerance,” *IEEE Transactions on Software Engineering*, vol. 24, no. 6, pp. 435 –450, June 1998.

[33] W. Z. Jun, W. Xin-an, and L. Guo-liang, “Dynamic fault-tolerant design for array processors based on immunology,” in *2nd International Conference on Advanced Computer Control (ICACC)*, vol. 5, March 2010, pp. 16 –20.

[34] Q. Zhou and K. Mohanram, “Cost-effective radiation hardening technique for combinational logic,” in *Proceedings of the 2004 IEEE/ACM International Conference on Computer-Aided Design*, 2004, pp. 100 –106.

[35] L. Zhang, X. Hu, X. Lu, G. Zhu, and Y. Zhang, “Simulation analysis for the materials shielding effectiveness of EMP,” in *Cross Strait Quad-Regional Radio Science and Wireless Technology Conference (CSQRWC)*, vol. 1, July 2011, pp. 32 –35.

[36] S. Voldman, D. Hui, L. Warriner, D. Young, R. Williams, J. Howard, V. Gross, W. Rausch, E. Leobangdung, M. Sherony, N. Rohrer, C. Akrout, F. Assaderaghi, and G. Shahidi, “Electrostatic discharge protection in silicon-on-insulator technology,” in *Proceedings of the 1999 IEEE International SOI Conference*, 1999, pp. 68 –71.

[37] J. R. Schwank, V. Ferlet-Cavrois, M. R. Shaneyfelt, P. Paillet, and P. E. Dodd, “Radiation effects in SOI technologies,” *IEEE Transactions on Nuclear Science*, vol. 50, no. 3, pp. 522 –538, June 2003.

[38] S. Lin and D. J. Costello, Jr., *Error Control Coding: Fundamentals and Applications.* Prentice Hall, Englewood Cliffs, NJ., 1983.

[39] M. Medwed and S. Mangard, “Arithmetic logic units with high error detection rates to counteract fault attacks,” in *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2011, pp. 1 –6.

[40] D. Schor, J. Scowcroft, C. Nichols, and W. Kinsner, “A command and data handling unit for pico-satellite missions,” in *Canadian Conference on Electrical and Computer Engineering*, May 2009, pp. 874 –879.

[41] A. Avizienis, *Software Fault Tolerance.* John Wiley & Sons, 1995, vol. 2, ch. The Methodology of N-Version Programming, pp. 22–45.

[42] G. Miremadi, J. Harlsson, U. Gunneflo, and J. Torin, "Two software techniques for on-line error detection," in *Digest of Papers of the Twenty-Second International Symposium on Fault-Tolerant Computing*, July 1992, pp. 328 –335.

[43] M. Hiller, "Executable assertions for detecting data errors in embedded control systems," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2000, pp. 24 –33.

[44] D. K. Pradhan, *Fault-tolerant computer system design.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[45] G. Latif-Shabgahi, J. M. Bass, and S. Bennett, "Integrating selected fault masking and self-diagnosis mechanisms," in *Proceedings of the Seventh Euromicro Workshop on Parallel and Distributed Processing*, February 1999, pp. 97 –104.

[46] T. J. Dysart and P. M. Kogge, "Reliability Impact of N-Modular Redundancy in QCA," *IEEE Transactions on Nanotechnology*, vol. 10, no. 5, pp. 1015 –1022, September 2011.

[47] Z. Chen and J. Dongarra, "Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources," in *20th International Parallel and Distributed Processing Symposium*, April 2006, p. 10.

[48] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software Implemented Fault Tolerance," in *In Proceedings of the 3rd International Symposium on Code Generation and Optimization*, 2005, pp. 243 –254.

[49] T. Lewis, "Primary processor and data storage equipment for the orbiting astronomical observatory," *IEEE Transactions on Electronic Computers*, vol. EC-12, no. 6, pp. 677 –687, December 1963.

[50] A. Avizienis, G. Gilley, F. Mathur, D. Rennels, J. Rohr, and D. K. Rubin, "The STAR (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design," *IEEE Transactions on Computers*, vol. C-20, no. 11, pp. 1312 –1321, November 1971.

[51] NASA, "Voyager - the interstellar mission," Website, February 2012. [Online]. Available: http://voyager.jpl.nasa.gov/mission/didyouknow.html

[52] R. E. Kuehn, "Computer Redundancy: Design, Performance, and Future," *IEEE Transactions on Reliability*, vol. R-18, no. 1, pp. 3 –11, February 1969.

[53] P. G. Norman, "The new AP101S general-purpose computer (GPC) for the space shuttle," *Proceedings of the IEEE*, vol. 75, no. 3, pp. 308 –319, March 1987.

[54] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock, "SIFT: Design and analysis of a fault-tolerant computer for aircraft control," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1240 – 1255, October 1978.

[55] E. Clune, Z. Segall, and D. Siewiorek, "Validation of fault-free behavior of a reliable multiprocessor system FTMP: A case study," in *American Control Conference*, June 1984, pp. 1112 –1120.

[56] I. L. Yen, “Specialized N-modular redundant processors in large-scale distributed systems,” in *Proceedings of the 15th Symposium on Reliable Distributed Systems*, October 1996, pp. 12 –21.

[57] NASA, “Galileo - true distributed computing in space,” Website, February 2012. [Online]. Available: http://history.nasa.gov/computers/Ch6-3.html

[58] F. D. Macchetto, “The Hubble Space Telescope,” in *Proceedings of the 16th IEEE Instrumentation and Measurement Technology Conference*, vol. 2, 1999, pp. 966 – 970.

[59] R. Berger, L. Burcin, D. Hutcheson, J. Koehler, M. Lassa, M. Milliser, D. Moser, D. Stanley, R. Zeger, B. Blalock, and M. Hale, “The RAD6000MC system-on-chip microcontroller for spacecraft avionics and instrument control,” in *2008 IEEE Aerospace Conference*, March 2008, pp. 1 –14.

[60] R. W. Berger, D. Bayles, R. Brown, S. Doyle, A. Kazemzadeh, K. Knowles, D. Moser, J. Rodgers, B. Saari, D. Stanley, and B. Grant, “The RAD750TM-a radiation hardened PowerPC processor for high performance spaceborne applications,” in *2001 IEEE Aerospace Conference*, vol. 5, March 2001, pp. 2263 –2272.

[61] A. L. R. Pouponnot, “Hardware and documentation status of the ERC32 3-chipset microprocessor,” TOS-EDD/2003.22/ALRP, March 2004, issue:2 Rev.:1. [Online]. Available: http://microelectronics.esa.int/erc32

[62] ——, “Hardware and documentation status of the ATMEL TSC695F microprocessor (ERC32 Single Chip),” TOS-EDD/2004.13/ALRP, March 2004, issue:1 Rev.:1. [Online]. Available: http://microelectronics.esa.int/erc32

[63] Gaisler, “LEON processors,” Website, 2008. [Online]. Available: http://www.gaisler.com/cms/index.php?option=com_content&task=section&id=4&Itemid=33

[64] J. Rosello, P. Silvestrin, G. Risuen, R. Weigand, J. Perello, J. Heim, and I. Tejerina, “AGGA-4: Core device for GNSS space receivers of this decade,” in *5th ESA Workshop on Satellite Navigation Technologies and European Workshop on GNSS Signals and Signal Processing*, December 2010, pp. 1 –8.

[65] C. Pham, H. Malcom, R. Maurer, D. Roth, and K. Strohbehn, “LEON3FT Proton SEE Test Results for the Solar Probe Plus Program,” in *IEEE Radiation Effects Data Workshop*, July 2011, pp. 1 –4.

[66] F. Koebel and J. F. Coldefy, “SCOC3: a space computer on a chip,” in *Design, Automation Test in Europe Conference Exhibition*, March 2010, pp. 1345 –1348.

[67] M. Stettler, M. Caffrey, P. Graham, and J. Krone, “Radiation effects and mitigation strategies for modern FPGAs,” 2004.

[68] M. Berg, “Embedding asynchronous FIFO memory blocks in Xilinx Virtex series FPGAs targeted for critical space system applications,” in *Proceedings of the Military/Aerospace PLD Conference*, 2009.

[69] F. L. Kastensmidt, L. Carro, and R. Reis, *Fault-Tolerance Techniques for SRAM-Based FPGAs (Frontiers in Electronic Testing).* Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[70] C. Gauer, B. J. LaMeres, and D. Racek, “Spatial avoidance of hardware faults using FPGA partial reconfiguration of tile-based soft processors,” in *2010 IEEE Aerospace Conference*, March 2010, pp. 1 –11.

[71] Xilinx, “PicoBlaze Controller information and resources,” Website, 2009. [Online]. Available: http://www.picoblaze.info/

[72] W. Lie and W. Feng-yan, “Dynamic partial reconfiguration in FPGAs,” in *Third International Symposium on Intelligent Information Technology Application*, vol. 2, November 2009, pp. 445 –448.

[73] M. A. Syed and E. Schueler, “Reconfigurable parallel computing architecture for on-board data processing,” in *First NASA/ESA Conference on Adaptive Hardware and Systems*, June 2006, pp. 229 –236.

[74] J. Ramos, J. Samson, D. Lupia, I. Troxel, R. Subramaniyan, A. Jacobs, J. Greco, G. Cieslewski, J. Curreri, M. Fischer, E. Grobelny, A. George, V. Aggarwal, M. Patel, and R. Some, “High-performance, dependable multiprocessor,” in *Aerospace Conference, 2006 IEEE*, 0-0 2006, p. 13.

[75] J. Samson, G. Gardner, D. Lupia, M. Patel, P. Davis, V. Aggarwal, A. George, Z. Kalbarcyzk, and R. Some, “High Performance Dependable Multiprocessor II,” in *2007 IEEE Aerospace Conference*, March 2007, pp. 1 –22.

[76] J. Samson, “Implementation of a Dependable Multiprocessor CubeSat,” in *2011 IEEE Aerospace Conference*, March 2011, pp. 1 –10.

[77] Intel, “Intel’s Teraflops Research Chip,” Website. [Online]. Available: www.intel.com/go/terascale

[78] R. Ginosar, “The Plural Architecture-Shared Memory Many-core with Hardware Scheduling,” January 2012. [Online]. Available: http://webee.technion.ac.il/~ran/papers/PluralArchitectureJan2012.pdf

[79] C. Villalpando, D. Rennels, R. Some, and M. Cabanas-Holmen, “Reliable multicore processors for NASA space missions,” in *2011 IEEE Aerospace Conference*, March 2011, pp. 1 –12.

[80] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. C. Miao, J. F. Brown, and A. Agarwal, “On-chip interconnection architecture of the Tile processor,” *IEEE Micro*, vol. 27, no. 5, pp. 15 –31, Sept-Oct 2007.

[81] J. P. Walters, R. Kost, K. Singh, J. Suh, and S. P. Crago, “Software-based fault tolerance for the Maestro many-core processor,” in *2011 IEEE Aerospace Conference*, March 2011, pp. 1 –12.

[82] S. P. Crago, D. I. Kang, M. Kang, R. Kost, K. Singh, J. Suh, and J. P. Walters, “Programming models and development software for a space-based many-core processor,” in *2011 IEEE Fourth International Conference on Space Mission Challenges for Information Technology (SMC-IT)*, August 2011, pp. 95 –102.

[83] T. K. Sarkar, “A pragmatic approach to adaptive antennas and space-time adaptive processing (STAP),” in *Proceedings of the 5th International Symposium on Antennas, Propagation and EM Theory*, 2000, p. 581.

[84] M. Mallick and B. F. La Scala, “Differential geometry measures of nonlinearity for ground moving target indicator (GMTI) filtering,” in *8th International Conference on Information Fusion*, vol. 1, July 2005, pp. 219 –226.

[85] H. Oshio, T. Asawa, A. Hoyano, and S. Miyasaka, “Estimation of tree crown structure in urban areas using high resolution airborne LiDAR,” in *2011 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, July 2011, pp. 2145 –2148.

[86] Y. J. Wang, M. Li, L. Wang, and P. Zhang, “A digital down conversion of WB radar based on intersection of spectrum,” in *2nd Asian-Pacific Conference on Synthetic Aperture Radar*, October 2009, pp. 921 –925.

[87] J. C. Rolon and P. Salembier, “Improved local PDF estimation in the wavelet domain for generalized lifting,” in *Picture Coding Symposium*, December 2010, pp. 546 –549.

[88] B. Vinnakota and N. K. Jha, “Synthesis of algorithm-based fault-tolerant systems from dependence graphs,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 8, pp. 864 –874, August 1993.

[89] K. J. Mighell, “CRBLASTER: A parallel-processing computational framework for embarrassingly-parallel image-analysis algorithms,” *Publications of the Astronomical Society of the Pacific*, vol. 122, no. 896, p. 8, 2010.

[90] D. Manolakis, “Detection algorithms for hyperspectral imaging applications: a signal processing perspective,” in *2003 IEEE Workshop on Advances in Techniques for Analysis of Remotely Sensed Data*, October 2003, pp. 378 –384.

[91] G. Cieslewski, A. Jacobs, C. Conger, and A. D. George, “Advanced space computing with system-level fault tolerance,” 2008.

[92] J. Zhang, J. Chen, B. Zou, and Y. Zhang, “Modeling and simulation of polarimetric hyperspectral imaging process,” *IEEE Transactions on Geoscience and Remote Sensing*, vol. PP, no. 99, pp. 1 –16, 2011.

[93] D. Lamarre, D. Aminou, P. Van den Braembussche, P. Hallibert, B. Ahlers, M. Wilson, and H. Luhmann, “Meteosat third generation: The infrared sounder instrument,” in *OSA Technical Digest on Fourier Transform Spectroscopy*, 2011.

[94] R. Caves, O. Turpin, T. Nagler, and D. Miller, “The role of Earth Observation in snowmelt runoff monitoring from high latitude basins: SAR aspects,” in *Proceedings of the IEEE International Geoscience and Remote Sensing Symposium*, vol. 4, July 1998, pp. 1858 –1860 vol.4.

[95] F. Caltagirone, “Status, results and perspectives of the Italian Earth Observation SAR COSMO - SkyMed,” in *European Radar Conference*, October 2009, pp. 330 –334.

[96] A. Freeman, “On ambiguities in SAR design,” *JPL-TRS*, 1992. [Online]. Available: http://hdl.handle.net/2014/39753

[97] J. Mittermayer and A. Moreira, “Interferometric processing of spaceborne SAR data in advanced SAR imaging modes,” in *RTO MP-61 (RTO SET Symposium on Space-Based Observation Technology)*, October 2000.

[98] M. Suess, B. Grafmueller, and R. Zahn, “A novel high resolution, wide swath SAR system,” in *2001 IEEE International Geoscience and Remote Sensing Symposium*, vol. 3, 2001, pp. 1013 –1015.

[99] C. Heer, C. Fischer, and C. Schaefer, “Frontend technology for digital beamforming sar,” *igarss08com*, vol. 49, pp. 2–3, February 2008.

[100] K. Harzallah and K. C. Sevcik, “Hot spot analysis in large scale shared memory multiprocessors,” in *Proceedings of the Supercomputing Conference*, November 1993, pp. 895 –905.

[101] M. Garland and D. B. Kirk, “Understanding throughput-oriented architectures,” *ACM Communications*, vol. 53, pp. 58 –66, November 2010.

[102] A. E. Eichenberger, P. Wu, and K. O’Brien, “Vectorization for SIMD architectures with alignment constraints,” in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, ser. PLDI ’04, 2004, pp. 82–93.

[103] PCISIG, “PCIe Base 3.0 Specification,” Website, February 2012. [Online]. Available: http://www.pcisig.com/specifications/pciexpress/base3/

[104] RTA, “RapidIO - The embedded fabric choice,” Website, 2012. [Online]. Available: http://www.rapidio.org/specs/current

[105] R. C. Ogus, “Fault-Tolerance of the Iterative Cell Array Switch for Hybrid Redundancy,” *IEEE Transactions on Computers*, vol. C-23, no. 7, pp. 667 – 681, July 1974.

[106] P. Layton, D. Czajkowski, J. Marshall, H. Anthony, and R. Boss, “Single event latchup protection of integrated circuits,” in *Fourth European Conference on Radiation and Its Effects on Components and Systems. RADECS 97*, sep 1997, pp. 327 –331.

[107] E. N. M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson, “A survey of rollback-recovery protocols in message-passing systems,” *ACM Computing Surveys*, vol. 34, pp. 375 –408, September 2002.

[108] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, “Detailed design and evaluation of redundant multi-threading alternatives,” in *Proceedings. 29th Annual International Symposium on Computer Architecture*, 2002, pp. 99 –110.

[109] B. T. Gold, B. Falsafi, J. C. Hoe, and K. Mai, “REDAC: Distributed, Asynchronous Redundancy in Shared Memory Servers,” Computer Architecture Lab at Carnegie Mellon (CALCM), Tech. Rep., 2008.

[110] PLX-Technology, “ExpressLane PCI Express (PCIe) Switch Family,” Website, February 2012. [Online]. Available: http://www.plxtech.com/products/expresslane/switches

[111] Intel-LAN-Access-Division, “Intel® 82598EB 10 Gigabit Ethernet Controller Datasheet,” Datasheet, December 2011. [Online]. Available: http://www.intel.com/content/www/us/en/ethernet-controllers/82598-10-gbe-controller-datasheet.html

[112] Altera, “Altera Corporation,” Website, February 2012. [Online]. Available: http://www.altera.com/

[113] Vitesse, “Making next-genaration networks a reality,” Website, February 2012. [Online]. Available: https://www.vitesse.com/

[114] Micrel, “Inovation Though Technology,” Website, February 2012. [Online]. Available: http://www.micrel.com/index.do

[115] Advanet-Inc, “Pci express - serial rapidio bridge board,” Datasheet, 2012. [Online]. Available: http://www.eurotech.com/DLA/datasheets/Products_Eurotech/AdEXP1566_sf.pdf

[116] IDT, “Integrated Device Technology - The Analog and Digital Company,” Website, February 2012. [Online]. Available: http://www.idt.com/

[117] PICMG, “CompactPCI Express - the logical next step,” Flyer, 2005. [Online]. Available: http://www.picmgeu.org/whats_new/picmg_europe_flyer_cpci_exp.pdf

[118] ASI-SIG, “Advanced Switching Technology,” Tech Brief, 2005. [Online]. Available: http://www.picmg.org/pdf/ASI_AdvSwitch_TB_0216.pdf

[119] Tilera, “Tilera Processors,” Website, February 2012. [Online]. Available: http://www.tilera.com/products/processors

[120] S. Damaraju, V. George, S. Jahagirdar, T. Khondker, R. Milstrey, S. Sarkar, S. Siers, I. Stolero, and A. Subbiah, “A 22nm IA multi-CPU and GPU System-on-Chip,” in *2012 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, feb. 2012, pp. 56 –57.

[121] NVIDIA, “NVIDIA Kepler GK110 Next-Generation CUDA Compute Architecture,” Datasheet, May 2012. [Online]. Available: http://www.nvidia.co.uk/content/PDF/kepler/NV_DS_Tesla_KCompute_Arch_May_2012_LR.pdf

[122] S. Soumekh, *Synthetic Aperture Radar Signal Processing with MATLAB Algorithms*. John Wiley & Sons, 1999.

[123] Y. K. Chan and V. C. Koo, “An introduction to Synthetic Aperture Radar (SAR),” in *Progress In Electromagnetics Research B, Vol 2*, 2008, pp. 27 –60.

[124] J. O. SmithIII, “Hamming window,” W3K Publishing, 2011. [Online]. Available: https://ccrma.stanford.edu/~jos/sasp/Hamming_Window.html

[125] D. A. Bader, K. Madduri, J. R. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy, “Designing Scalable Synthetic Compact Applications for Benchmarking High Productivity Computing Systems,” *Cyberinfrastructure Technology Watch*, November 2006. [Online]. Available: http://gauss.cs.ucsb.edu/publication/ctwatch-ssca.pdf

[126] J. W. Eaton, “GNU octave,” Website, March 2012. [Online]. Available: http://www.gnu.org/software/octave/

[127] M. Frigo and S. G. Johnson, “FFTW: an adaptive software architecture for the FFT,” in *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing*, vol. 3, May 1998, pp. 1381 –1384.

[128] M. E. Thomadakis, "The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms," Research Report, March 2011. [Online]. Available: http://sc.tamu.edu/systems/eos/nehalem.pdf

[129] TSU, "The Linux Programming Environment," Tutorial. [Online]. Available: http://www.cs.txstate.edu/labs/tutorials/tut_docs/Linux_Prog_Environment.pdf

[130] J. Osier, "The GNU profiler (gprof)," Website, January 1993. [Online]. Available: http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html

[131] Eclipse, "Eclipse CDT (C/C++ Development Tooling)," Website, 2012. [Online]. Available: http://www.eclipse.org/cdt/

[132] M. Gerndt, K. Fuerlinger, and E. Kereku, "Periscope: Advanced Techniques for Performance Analysis," in *Parallel Computing: Current & Future Issues of High-End Computing*, ser. NIC, vol. 33, no. ISBN 3-00-017352-8, 2006, pp. 15 –26.

[133] V. Petkov and M. Gerndt, "Integrating parallel application development with performance analysis in Periscope," in *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, April 2010, pp. 1 –8.

[134] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "PARAVER: A Tool to Visualize and Analyze Parallel Code," In WoTUG-18, Tech. Rep., 1995.

[135] A. Rico, A. Duran, F. Cabarcas, Y. Etsion, A. Ramírez, and M. Valero, "Trace-driven simulation of multithreaded applications," in *ISPASS*, 2011, pp. 87–96.

[136] M. Geimer, F. Wolf, B. J. N. Wylie, D. Becker, D. Böhme, W. Frings, M. A. Hermanns, B. Mohr, and Z. Szebenyi, "Recent Developments in the Scalasca Toolset," in *Proceedings of the 3rd Parallel Tools Workshop Tools for High Performance Computing.* Springer, 2010, ch. 4, pp. 39–51.

[137] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," *Tools for High Performance Computing*, pp. 157 –173, 2009.

[138] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," Whitepaper, 2009. [Online]. Available: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[139] D. Schaa and D. Kaeli, "Exploring the multiple-GPU design space," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing.* Washington, DC, USA: IEEE Computer Society, 2009, pp. 1 –12.

[140] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879 –899, May 2008.

[141] Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU Cluster for High Performance Computing," in *Proceedings of the 2004 ACM/IEEE Supercomputing Conference*, November 2004, p. 47.