TECHNISCHE
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR INFORMATIK

Sonderforschungsbereich 342:
Methoden und Werkzeuge für die Nutzung
paralleler Rechnerarchitekturen

Integration of Load Distribution into ParMod-C

Thomas Schnekenburger

TUM-I9536
SFB-Bericht Nr.342/19/95 A
November 1995

Integration of Load Distribution into ParMod-C[1]

Thomas Schnekenburger

Institut für Informatik, Technische Universität München
Orleansstr. 34, D-81667 München, GERMANY
email: schneken@informatik.tu-muenchen.de

Abstract

The major part of environments for parallel programming in distributed systems either neglects load distribution support or realizes load distribution by the trivial task farming paradigm. This paper presents new language concepts for the support of application integrated load distribution. In addition to the task farming approach, the system can also dynamically migrate objects. Active objects and passive objects are combined to a very flexible and elegant programming model. Specific tasks for initialization and migration are used to realize efficient and portable load distribution mechanisms. The paper describes the new programming model and presents language constructs realizing this model. The language constructs are an extension of the parallel and distributed programming language ParMod-C. A program example shows the usage of the new language constructs and demonstrates easy implementation of load distribution using the new concept.

1 Introduction

Environments for parallel programming in distributed systems are presently realized either as library as for example PVM [20] and p4 [5], or as parallel programming language as for example Linda [7], Concert/C [2], and ParMod-C [22]. The discussion about parallel programming environments is going on and many different paradigms and language constructs have been proposed in the last years. Main aspects of the discussion are the expressiveness of language constructs, the simple usage of the programming environment, the flexibility of the programming model, and the possibility of an efficient language implementation.

This paper emphasizes another aspect which is often neglected by programming environments but which is essential for an efficient execution: *load distribution support* in parallel programming environments. Generally, load distribution has to manage the assignment of service demands to distributed resources of a system. In the context of a parallel programming environment, load distribution has to manage the mapping of *subproblems* of a parallel program to *processes* with the goal of minimizing the runtime of a parallel program. In the following, we denote subproblems also as *objects* where the definition and implementation of objects depends fully on the particular application: Objects may be passive (for example variables, complex data structures, external files) or active (for example processes or threads).

We present a new programming model based on cooperating agents and language constructs realizing this model. A language based approach for the support of load distribution has several important advantages compared to a programming library or the direct usage of the operating system:

[1]This report is available via
http://wwwpaul.informatik.tu-muenchen.de/projekte/sfb342/pub/sfb342-19-95A.ps.gz

- Semantics: By integrating the parallel programming model into the language implementation, the runtime system has more information about the (parallel) semantics of the application.

- Instrumentation: Transparent integration of implicit runtime system calls can facilitate the difficult task of parallel programming. For example, allocation and deallocation of message buffers can be performed implicitly by the runtime system.

- Compiler checks: The compiler of a parallel programming language has more possibilities to find errors relating to communication and synchronization.

The next section classifies parallel applications and programming environments with respect to load distribution. Section 3 presents the new programming model. Section 4 describes the language constructs realizing the programming model. An example in section 5 illustrates the language constructs. Section 6 gives a summary and a conclusion.

2 Classification and Related Work

This section classifies parallel applications and programming environments with respect to dynamic load distribution. Considering the requirements for dynamic load distribution, parallel applications can be classified into two classes.

The first class are applications which can be implemented using the *task farming* paradigm [13] (also called *pool of tasks* or *task bag* [1] paradigm). Load distribution is realized by dynamically assigning new objects to processes. Typical examples are backtracking programs like traveling salesman. Load distribution support is relatively easy to accomplish for this application class: The function of the load distribution system is just to find *underloaded* processes and to assign new objects to these processes. The emerging problems are similar to the traditional non-preemptive assignment of processes to computing units.

An efficient execution of task farming programs can only be achieved if the semantics of the program does not fix the mapping of subproblems to processes. Furthermore, the synchronization among subproblems must allow a reasonable degree of parallelism. Applications which do not meet these requirements form the second class. These are for example many numerical algorithms (e.g., PDE solvers and simulation programs [9]), and typical client-server applications where server requests are bound to specific servers and therefore cannot be assigned according to load distribution requirements. For applications in this second class, dynamic load distribution can only be achieved by *redistributing* or *migrating* objects.

Implementing efficient parallel programs using object migration for dynamic load distribution however is usually a complex and error proning task:

- Either a protocol has to ensure globally consistent information about the location of migration objects or messages concerning migration objects have to be forwarded if they are received by a wrong process.
- Due to real-time dependencies of the load distribution strategy, testing and debugging of strategies and reproduction of specific executions is very difficult.
- There is usually a large variety of suitable load distribution strategies. Since the implementation of different strategies always requires a considerable effort, a non-expert in load distribution will not use “the best” strategy.

- Observation of program behavior is essential for designing efficient parallel programs. If the programming environment does not directly support migration mechanisms, the corresponding observation tools cannot support migration too. As a result, the programmer has to use (and in most cases to implement) specific tools for observing migrations.

Summing up, load distribution support by the programming environment is desirable especially for applications requiring object migrations. Based on these observations, programming environments can be classified into three classes with respect to their load distribution support:

- The first class consists of environments which regard load distribution as a problem of the application and leave load distribution completely to the programmer. Most parallel programming libraries like PVM [20] and p4 [5] and parallel programming languages such as Emerald [4] and ARGUS [17] belong to this class; a survey may be found in [3]. (Although PVM manages the assignment of processes to nodes, it does not manage the assignment of subproblems to certain processes).

- The second class consists of environments supporting the task farming paradigm as for example Linda [7], DIB [12], and Dynamo [21]. Furthermore, many parallel object oriented programming languages, as for example Mentat [14] and Charm++ [16], and parallel functional programming languages [15] belong to this class.

- Environments which can *migrate* application objects form the third class. Casas et. al. [8] present three general concepts for load migration in PVM. Migration is either based on process migration among nodes, thread migration among processes, or data movement among processes. Furthermore, there are systems that support migration for objects with a specific implementation. For example, the ParForm environment [6] supports migration for numerical applications by automatically repartitioning the data domain with respect to the actual load.

Our new concept belongs to the third class. It combines thread migration and data movement, therefore it is comparable to the concepts presented by Casas et. al. [8]. Nevertheless, due to the arguments mentioned above, our system is based on a language approach instead of a library concept.

3 Programming Model

Our approach for realizing load distribution is to integrate an *agent* concept into a parallel programming language.

3.1 Agents

An agent is an object that provides a certain service, hopping from node to node. In this context, the agent concept can be described by the keywords *processes*, *tasks*, *suitcase*, *move task,* and *restart task*:

Processes Processes execute the program code associated with agents. Processes may contain several agents.

Tasks Tasks are computational requests that are performed by agents. An agent processes his tasks sequentially.

Suitcase Associated with an agent is its *suitcase*, containing data that the agent carries with it if it moves to other processes.

Move task An agent executes a *move task* to migrate to another process.

Restart task If an agent is sent to a process, it receives initialization data. A *restart task* is responsible for the initialization of further data structures corresponding to the agent using this initialization data.

3.2 Basic program execution model

ParMod [11] is a set of constructs for parallel programming in distributed systems. The ParMod programming model can be implemented by extending existing sequential programming languages like Pascal, Modula, and C by ParMod language constructs. To realize a language based environment supporting load distribution, we will extend the basic program execution model of ParMod. Although ParMod was developed about a decade ago (when nobody did talk about *agents*), it is excellently suited for integrating an agent concept without changing the basic execution model. In fact, all load distribution mechanisms can be realized *on top* of the existing programming model.

A ParMod program consists of a number of *processes*. Within each process, several *threads* may run concurrently (see figure 1). ParMod offers *conditional critical regions* for local synchronization of threads within a process. ParMod processes communicate via *global procedure calls* (GPCs). To call a global procedure, the programmer has to specify the name of the global procedure and the process in which the global procedure is executed. A global procedure may have three basic types of parameters: **in**, **out**, and **inout** parameters. When process A calls a global procedure in process B, the ParMod runtime system performs the following steps:

1. Values of the actual **in** and **inout** parameters are sent from process A to process B.

2. After receiving the parameter values, a new thread with these parameter values is started within process B.

3. After the termination of this thread, results (values of **inout** and **out** parameters) are implicitly returned via a call of a special *answer* procedure in process A.

A GPC is asynchronous, that means a thread proceeds immediately after calling a global procedure. Using an **await result** statement, it is possible for process A to wait for results from the GPC.

In the ParMod model, a program consists of several processes which may have different program codes. Some processes may be replicated, i.e, there may be several processes having identical program codes. In the context of this paper it is assumed that all processes of the program have identical program codes. The generalization to a system with different program codes is straightforward.

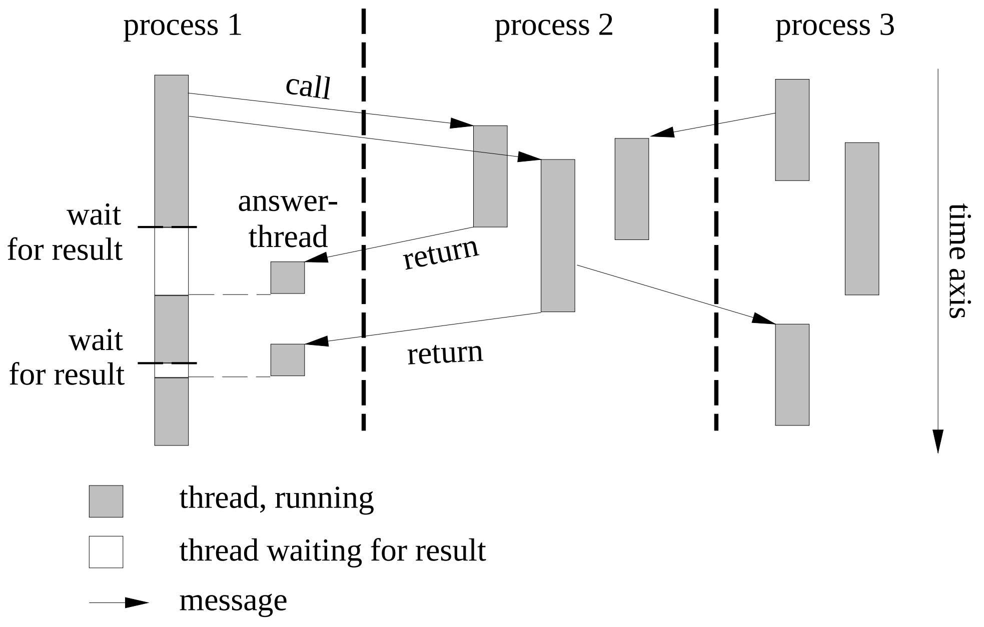

Figure 1: ParMod execution model

3.3 The PAT programming model

The integration of agents into ParMod is called the PAT (Process, Agent, Task) model. The PAT model supports two basic paradigms of load distribution:

1. Non-preemptive assignment: Tasks are assigned non-preemptively to agents.
2. Agent migration: Agents may be migrated among processes, that means, they are assigned preemptively to processes.

The integration of the agent model into ParMod is based on the following ideas:

- An agent is represented by an *elementary data structure* (EDS), similar to a record data structure.
- ParMod threads correspond to tasks. Processing of a task by an agent therefore means execution of the corresponding thread by the process that contains the agent.
- The load distribution strategy (which is assumed to be integrated into the runtime system) assigns a move task to an agent for migrating it to another process. Using the agent's EDS, the move task is responsible to "pack" and to move the agent's suitcase to another process.
- If an agent is assigned to a new process, the runtime system implicitly assigns the restart task to it. The restart task is responsible to "unpack" the suitcase.

Figure 2 illustrates the realization of the PAT model on top of the basic ParMod model. The example consists of three processes. There are four agents. Agent a_1 is assigned to process 2. The agents a_2, a_3, and a_4 are assigned to process 3. In the diagram, tasks are

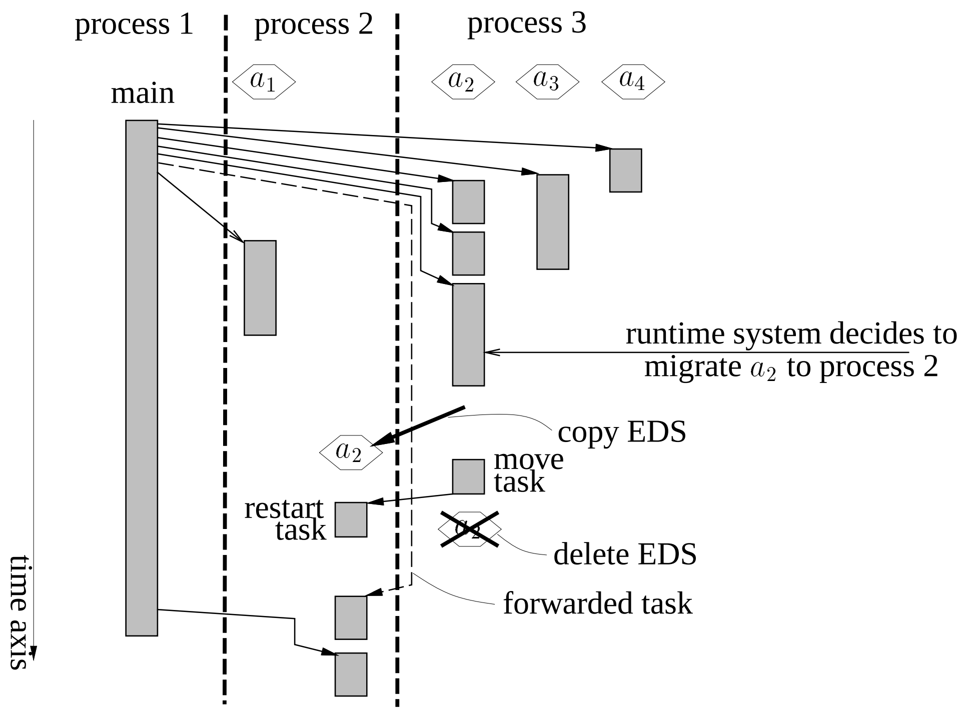

Figure 2: Agent migration using the basic ParMod model

placed in the column of the agent to which they are assigned. At the beginning, the procedure *main* in process 1 performs several GPCs, corresponding to the assignment of tasks to agents. Note that tasks for a certain agent are sequentialized. The most interesting phase of the example is the migration of agent a_2 from process 3 to process 2. When the load distribution strategy decides to migrate agent a_2, the following steps are performed:

1. The runtime system waits until the current task of agent a_2 is terminated.

2. The elementary data structure (EDS) of a_2 is copied implicitly to process 2. Since addresses are useless when they are sent to another process, it is not allowed that an EDS contains addresses. Nevertheless, the next steps make the migration of linked data structures or other data which are not directly connected to an agent feasible:

3. The load distribution system implicitly calls the *move* task which is assigned to a_2. The move task is implemented as global procedure by the programmer and has first to "pack" the agent's suitcase. For example, a linked list has to be packed into a linear array.

4. The move task sends the suitcase via a *restart* task to process 2. The *restart* task is also implemented as global procedure by the programmer.

5. The *restart* task "unpacks" the suitcase. For example, a linked list has to be constructed out of a linear array.

6. When the restart task terminates, the migration of the agent is complete. Now the load distribution system can assign further (conventional) tasks to the agent.

The assignment of agents to processes may be classified as *quasi-preemptive:* The load distribution strategy initializes an agent migration only if no task is assigned to the agent. Therefore, it is not necessary to interrupt tasks and to save intermediate task states. Agent migrations are just implemented by moving the suitcase of an agent to another process. Therefore, agent migrations can be implemented efficiently and also portable across heterogeneous platforms. Obviously, this is an significant advantage compared to preemptive assignment as used for example by Casas et. al. [8]. The price for the advantages of quasi-preemptive assignment is that the LDS probably has to wait for the termination of the agent's task. Therefore, the programmer should try to avoid "long-lived" tasks.

An interesting point is the automatic forwarding of tasks: When an agent is migrated, all tasks which have been assigned to this agent but which are not yet processed are forwarded automatically to the new location of the agent.

The combination of an EDS and a *move* task provides more flexibility compared to a *deep copy* concept (a deep copy implicitly follows the pointers in data structures, as used for example in Concert/C [2]).

3.4 Load distribution strategy

To realize load distribution mechanisms for a programming model like PAT, a load distribution strategy has to be provided by the runtime system of the programming language implementation. The ALDY load distribution system [18] directly supports the PAT programming model. ALDY is independent of the implementation of load distribution objects. Furthermore, ALDY uses the communication routines of the application for internal communication. Therefore, ALDY can be easily integrated into the ParMod runtime system.

Experiences with several types of applications have shown that the benefit of a specific load distribution strategy depends strongly on the characteristics of the application. Therefore, ALDY offers not a single strategy but a collection of several parameterized load distribution strategies. The strategy is specified in a separated specification file that is scanned by the runtime system at program start. Therefore, the application program is independent of the load distribution strategy and the programmer can easily experiment with several strategies.

A load distribution strategy for the PAT model consists of two components: The *assignment component* is responsible for the assignment of new agents to processes and the assignment of tasks to agents. The *migration component* is responsible for the migration of agents to other processes. For both components there is a large number of possible strategies. For example, the assignment component may be realized as central *queue* of tasks or as distributed object management (see for example [12]). The migration component may be realized using a sender-initiated or a receiver-initiated protocol [10].

An important factor for all load distribution strategies is the information about the actual state of the system. For using ALDY within the ParMod runtime system we can use the following approach:

- The strategy uses only information about the states of objects of the PAT model. System resources like CPU, interconnection network, main memory are not observed directly. Nevertheless the strategy reacts indirectly on the load of system resources by migrating agents from "busy" processes to "idle" processes. By using only application specific load information, the system remains portable and may be used in an heterogeneous environment since is does not depend on specific hardware oriented information.

- Object states are usually determined implicitly by the ParMod runtime system since the runtime system controls communication and synchronization. If necessary, the programmer may explicitly report object states to the load distribution system. This is useful for some special cases as for example user implemented busy waiting loops which are not detected by the runtime system.

4 Language Constructs

This section describes the new language constructs realizing the PAT model. The language constructs are an extension of the parallel and distributed programming language ParMod-C [22, 19]. ParMod-C is an extension of ANSI-C and realizes the basic ParMod program execution model (see section 3.1) by offering global procedures and language constructs for local synchronization. The following description does not require knowledge of the ParMod-C language constructs.

Agents are represented by an elementary data structure (EDS, see above) of a programmer-specified type. The declaration of an agent type *atype* consists of the declaration of local variables representing the the elementary data structure and the **init** and **move** procedures:

agent *atype* { *local variables* } **init** *init_proc* **move** *move_proc*;

The *local variables* may be defined similar to a C **struct**. If an agent is migrated, the values of the local variables are implicitly sent by the runtime system to the new process. Agents are generated by calling an initialization procedure. For that purpose, the **init** part declares a global procedure that has to be called to generate an agent of type *atype*. The **move** part declares the global procedure that is invoked if an agent of type *atype* is migrated to another process. This global procedure is started implicitly by the runtime system in the process that sends the agent (see figure 2).

There are two possibilities to define global procedures: Global procedures may be bound to an agent type or not bound to an agent type. A global procedure that is bound to an agent type is defined by

global *gp_bound* (*parameters*) **agent** *atype*
{ *statements* }

If a global procedure is bound to the agent type *atype*, GPCs of this procedure are always assigned to an agent of this type. A global procedure that is not bound to an agent type is defined by

global *gp_unbound* (*parameters*)
{ *statements* }

Global procedures that are specified as **init** or **move** procedures for an agent type have to be bound to this agent type. The syntax for the access to the local variables of an agent is similar to an access to the components of a C **struct** where the **struct** name is the name of the agent type.

The possibility to bind global procedures to agent types and the possibility to bind global procedures to certain process or agent instances implies four modes of GPCs:

- Process-bound calls: The programmer specifies the target process for a GPC, where the global procedure is not bound to an agent type. The corresponding task is not assigned to a certain agent but is started immediately within the specified process. This is the conventional mode in the ParMod model and the basic ParMod-C language.

- Unbound calls: The programmer specifies a global procedure that is not bound to an agent type: the load distribution system automatically selects a target process for the GPC and performs a process-bound GPC to that process. This mechanism directly realizes the task farming paradigm.

- Agent-bound calls: The programmer specifies an agent, to which the GPC is bound. The job of the load distribution system is to pass the GPC to the process actually owning the agent. The location (process) of an agent is transparent for the programmer. As mentioned above, agent-bound calls to the same agent are sequentialized.

- Agent-type-bound calls: The programmer specifies an agent type. The load distribution system selects an agent of this type and performs an agent-bound call to this agent.

The following statements illustrate the four modes of GPCs:

gp_unbound `(` *parameters* `);`
gp_unbound `[` *process instance* `]` `(` *parameters* `);`
gp_bound `(` *parameters* `);`
gp_bound `[` *agent instance* `]` `(` *parameters* `);`

Assuming that *gp_unbound* is not bound to an agent type (see above), the first GPC is unbound and the second GPC is process-bound. Assuming that the global procedure *gp_bound* is bound to an agent type *atype*, the third GPC is agent-type-bound. The fourth GPC specifies an agent-bound GPC to the agent *instance* of type *atype*.

To generate a new agent of type *atype*, the programmer has to call the global procedure that is specified as **init** procedure for this agent type. In addition, the programmer specifies an instance number for the new agent:

init_proc `[` *instance* `]` `(` *parameters* `);`

It is the responsibility of the programmer to use globally unique agent instance numbers. At first sight, this concept seems uncomfortable, but it enables the application to make a "global plan" of agent identifiers and therefore to be relieved from implementing protocols for the exchange of identifiers.

When a GPC is assigned by the runtime system to an agent, the corresponding task may need the instance number of this agent. Therefore, the expression

instance

returns the instance number of the actual agent. A *restart* GPC is used to restart an agent in the receiving process (see above). This GPC cannot be handled like an ordinary agent-bound GPC since the system can not assign ordinary tasks to the agent before the migration is complete. Therefore, the statement

restart *gp_bound* `(` *parameters* `);`

is used to specify a restart GPC. The **restart** statement can only be executed by a global procedure that is specified as **move** procedure. The target process of the restart GPC is implicitly determined by the load distribution system. To terminate an agent, a task which is assigned to the agent just has to execute the statement

endagent;

This statement terminates the executing task (similar to the **return** statement), releases memory for the local variables of the agent, and stops any load distribution mechanisms (task assignments or migrations) relating to this agent.

5 Example

The following example illustrates the new language constructs and shows how the migration mechanism can be realized on top of the four GPC modes mentioned above. The example shows a code fragment for a parallel program using the client-server paradigm (Figure 3). Keywords are printed in **boldface** letters.

```
int phi(...); /* represents mapping of requests to server-instances */

agent server { long address; } init init_server move move_server;
/* Declaration of the agent type server */

global init_server(void) agent server
{ Initialize server data; server.address= (long)address of server data; }

global restart_server(in int d[100]) agent server
{ rebuild data structure of server usind d and put address to server.address}

global move_server(void) agent server
{ int data[100];
  build data of type int[100] using server.address
  restart restart_server(data);
}

global server_task(inout int res) agent server
{ perform server call using data in server.address}

main(void)
{ int i, res;
  /* start four server agents locally: */
  for (i=1; i<=4; ++i)
     init_server[i+4*mynumber]();
  /* main now acts as a client performing server calls: */
  for (all server calls) {
     server_task[phi(this call)](& res);
     if necessary, wait for results using await result(res);
  }
}
```

Figure 3: Client-server example

There is one agent type `server`, representing a server type that handles some requests. It is assumed that there is a fixed mapping of server requests to server instances, implied by the data dependencies of the application. The function `phi` (line 1) represents this mapping of server requests to server instances. Assuming that all servers are generated at the beginning of the program, the only possibility for load distribution is to migrate servers to other processes. To illustrate the usage of the restart mechanism, we assume that the data corresponding to a server is not implemented as a collection of variables but as a complex data structure. If a server is migrated, this data structure has to be packed into an array by the sending process. The receiving process has to reconstruct the data from this array.

The declaration of the agent type `server` specifies one local variable `address` that holds the address of the agent data (line 2). The global procedure `init_server` is used to initialize agents of type `server`. To migrate an agent of type `server`, the global procedure `move_server` packs the server data into the array `data` and sends the data via a **restart** GPC of `restart_server` to the receiving process. The global procedure `restart_server` rebuilds the data structure of the server and assigns the address to the local variable `address`.

The `main` function shows the initialization of four servers using agent-bound GPCs of `init_server` (line 19). The ParMod-C expression **mynumber** which returns the actual process instance is used to compute globally unique instance numbers for agents. Server requests are implemented by agent-bound GPCs to the corresponding instance of `server` (line 22). The conventional ParMod-C language construct **await result** may be used to wait for results of server requests (line 23).

The example demonstrates that there is only few effort for the programmer to realize object migration, even if complex data structures are connected to the servers. The most difficult parts of the implementation such as global information about object locations, forwarding of GPCs, and the load distribution strategy are realized by the runtime system.

6 Summary and Conclusions

For several types of applications, the common *task farming* paradigm cannot be used to achieve dynamic load distribution due to data dependencies and synchronization among subproblems of the program. For these applications, migration of objects is necessary to realize load distribution.

This paper introduces a new load distribution concept based on cooperating agents. Agent migration is realized “quasi-preemptively” by data movement. This allows very easy and efficient implementation of the migration mechanism. User implemented tasks for moving and re-starting agents provide flexibility with respect to agent implementation. Furthermore, agents can hop across heterogeneous platforms, because the underlying runtime system can perform all necessary data marshalling. The load distribution strategy is completely separated from the application program and can be exchanged without having to recompile the application.

The paper presents an extension of the parallel and distributed programming language ParMod-C realizing the new programming model. The program example demonstrates that the programmer can limit implementation effort for object migration to object specific code.

Summing up, the new concept represents an interesting compromise between automatic object migration which is only useful for specific applications with a predefined implementation of migration objects, and systems which leave load distribution completely to the programmer.

References

[1] Gregory R. Andrews, “Paradigms for Process Interaction in Distributed Programs”, *ACM Computing Surveys*, 23(1):49–90, March 1991.

[2] Joshua S. Auerbach, Arthur P. Goldberg, German S. Goldszmidt, Ajei S. Gopal, Mark T. Kennedy, Josyula R. Rao, and James R. Russell, “Concert/C: A Language for Distributed Programming”, In *Proc. of the Winter 1994 USENIX Conference, San Francisco, California*, 1994.

[3] H.E. Bal, J.G. Steiner, and A.S. Tanenbaum, “Programming Languages for Distributed Computing Systems”, *ACM Computing Surveys*, 21(3):261–322, 1989.

[4] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter, “Distribution and Abstract Types in Emerald”, *IEEE Transactions on Software Engineering*, 13(1):65–76, 1987.

[5] R. Butler and E. Lusk, “Monitors, messages, and clusters: The p4 parallel programming system”, *Parallel Computing*, 20(4):547–564, 1994.

[6] C.H. Cap and V. Strumpen, “Efficient Parallel Computing in Distributed Workstation Environments”, *Parallel Computing*, 19(11):1221–1234, 1993.

[7] N.J. Carriero, D. Gelernter, T.G. Mattson, and A.H. Sherman, “The Linda alternative to message-passing systems”, *Parallel Computing*, 20(4):633–656, 1994.

[8] Jeremy Casas, Ravi Konuru, and Steve W. Otto, “Adaptive Load Migration systems for PVM”, In *Supercomputing’94, Washington D.C.*, pages 390–399, 1994.

[9] Zarka Cvetanovic, Edward G. Freedman, and Charles Nofsinger, “Efficient Decomposition and Performance of Parallel PDE, FFT, Monte Carlo Simulations, Simplex, and Sparse Solvers”, *The Journal of Supercomputing*, 5:219–238, 1991.

[10] Derek L. Eager, Edward D. Lazowska, and John Zahorjan, “A Comparison of Receiver-Initiated and Sender-Initiated Adaptive Load Sharing”, *Performance Evaluation*, (6):53–68, 1986.

[11] S. Eichholz, “Parallel Programming with ParMod”, In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 377–380, 1987.

[12] Raphael Finkel and Udi Manber, “DIB—A Distributed Implementation of Backtracking”, *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, 1987.

[13] Geoffrey C. Fox, Roy D. Williams, and Paul C. Messina, *Parallel Computing Works!*, Morgan Kaufmann Publishers, Inc., 1994.

[14] Andrew S. Grimshaw, “Easy-to-use Object-Oriented Parallel Processing with Mentat”, *Computer*, 26(5):39–51, 1993.

[15] Robert H. Halstead, “Parallel Computing Using Multilisp”, In J.S. Kowalik, editor, *Parallel Computation and computers for artificial intelligence*, pages 21–50, Kluwer Academic Publishers, 1988.

[16] L.V. Kalé and S. Krishnan, “Charm++: A portable concurrent object oriented system based on C++”, In *Conference on Object Oriented Programming Systems, Languages and Applications*, 1993.

[17] Barbara Liskov, “The ARGUS Language And System”, In M. Paul and H.J. Siegert, editors, *Distributed Systems, Methods and Tools for Specification*, chapter 7, pages 343–430, Springer Verlag, 1985.

[18] Thomas Schnekenburger, “The ALDY Load Distribution System”, Technical Report TUM-I9519 and SFB 342/11/95 A, Technische Universität München, May 1995, ftp ftp.informatik.tu-muenchen.de:/local/lehrstuhl/paul/parmod/doc/aldy-doc.ps.Z.

[19] Thomas Schnekenburger, Andreas Weininger, and Michael Friedrich, “Introduction to the Parallel and Distributed Programming Language ParMod-C”, Technical Report TUM-I9139 and SFB 342/27/91 A, Technische Universität München, October 1991.

[20] V.S. Sunderam, G.A. Geist, J. Dongarra, and R. Manchek, “The PVM concurrent computing system: Evolution, experiences, and trends”, *Parallel Computing*, 20(4):531–546, 1994.

[21] Erik Tärnvik, “Dynamo - A Portable Tool for Dynamic Load Balancing on Distributed Memory Multicomputers”, *Concurrency: Practice and Experience*, 6(8):613–639, 1994.

[22] A. Weininger, T. Schnekenburger, and M. Friedrich, “Parallel and Distributed Programming with ParMod-C”, In *First International Conference of the Austrian Center for Parallel Computation*, pages 115–126, Salzburg, September 1991, Springer, LNCS 591.