Distributed Event-driven Simulation- Scheduling Strategies and Resource Management

Overeinder, B.J.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 7

Dynamic Load Balancing: Automatic Control of Execution Threads

When one's ill or unhappy, one needs something outside oneself to hold one up. It is a good thing, I think, when one has been knocked out of one's balance, to have some external job or duty to hang on to.

—Aldous Leonard Huxley

7.1 Introduction

The progressive use of event-driven simulation techniques as an essential approach to problem solving in, for example, science, engineering, and economics, has urged the need for robust performance. Efforts to parallelize the discrete event simulation execution mechanism resulted in two different parallel discrete event simulation (PDES) protocol classes: conservative and optimistic. In the preceding chapters of this thesis, we have extensively studied the performance and execution behavior of the optimistic scheduling protocol Time Warp. The performance of parallel programs, and in particular of optimistic simulations, is (negatively) influenced by the appearance of load imbalance over the processing nodes of the parallel or distributed computing platform. Due the very complex execution patterns in optimistic simulation, the load imbalance cannot be predicted before the execution of the parallel simulation. Hence, we must solve the load imbalance dynamically. To this end, we need an execution environment that allows for the dynamic migration of execution threads (processes, threads, or objects) from overloaded processing nodes to "underloaded" or less loaded processing nodes.

The Polder project is an experiment framework for wide-area resource management, which deals with both resource allocation and job placement, and dynamic resource management in local clusters. The main contribution of this chapter is the presentation of the design and implementation of, and experimentation with, dynamic resource management local to a cluster. The dynamic resource management environment is named Dynamite, and incorporates provisions for transparent process migration that allow for efficient load balancing.
Dynamic Load Balancing of Execution Threads

of parallel jobs over the processing nodes of the cluster.

The development of Dynamite is the first step towards dynamic load balancing of execution threads in PDES. Dynamic load balancing of parallel jobs is a complex task and a research topic that is extensively studied. Dynamic load balancing of PDES can be even more complex, as the notion of workload must be redefined, see for example thrashing behavior that incurs tremendous amounts of work but no progress (or useful work). Also, we have to consider which class of PDES applications can potentially benefit from dynamic load balancing over the processing nodes of a cluster. By the amount of event messages in PDES, the communication latency is a considerable factor in the determination of the performance. In this respect, self-initiated simulation applications, where the simulation processes schedule most of their events to themselves, seems to offer a good opportunity. The remainder of the introduction presents the general setting of wide-area resource management and dynamic load balancing local to a cluster. The challenges of dynamic load balancing of PDES are not considered in this chapter, but some ideas are discussed in Section 7.7.

The current developments in clusters of workstations, and on a larger scale wide-area distributed computing, or “grid” technologies (Foster and Kesselman 1998), indicate the importance of resource management to determine the efficacy of a distributed computing environment. In distributed environments the typical set of jobs consists of interactive and batch jobs, which in turn can be sequential or parallel execution runs. By the diversity of the jobs offered to the distributed environment—interactive users start sequential and parallel jobs, and batch jobs arrive with some arrival probability distribution function—both the demand for, and the availability of resources are highly dynamic.

Resource management in distributed environments spans a variety of activities such as job scheduling, I/O scheduling, load balancing, etc. In order to optimize performance of applications, or the utilization of resources, the resource management system should be able to react on changes in the distributed computing environment. As a consequence, several facilities have to be made available to the distributed computing environment in order to interact with resources and applications. The term “metacomputing” was introduced by Smarr and Catlett (1992), as a reference to such a set of widely different computing resources that presents itself to the user as a single computing environment.

A serious problem hampering the development of metacomputing environments is the lack of a sound theoretical basis for resource management strategies to build upon. In order to break the impasse, we developed an experimental environment that provides a framework for the development and evaluation of the various components making up the metacomputer (van Halderen et al. 1998). The experimental environment is essentially a metacomputer in its functionality and characteristics, but allows to study, for example, different policies for resource management or test designs and implementations of scalable I/O libraries, and the validation of theories.
In this chapter we focus on issues concerned with dynamic load balancing of parallel applications within a local cluster in the metacomputing environment. With the availability of high-speed networks, clusters of workstations achieve the same scalable parallelism as the current massively parallel processor (MPP) architectures. Hence, we currently witness a shift of emphasis in high-performance computing from expensive, special-purpose monolithic systems to the use of clusters of workstations or PCs. When using time-shared workstation clusters as high-performance computing servers, however, one has to cope with the dynamical behavior of the compute nodes, the network load and the application tasks. These can lead to local load imbalances, which hamper the application's execution and the overall system performance.

One way to deal with this dynamically changing resource requirement would be an adaptive system that supports the migration of processes from overloaded to under-loaded processors at runtime, without interference from the programmer. In addition, the resulting adaptive system should hide the complexity of the load balancing from the programmer/end-user. These observations resulted in the design and implementation of an experimental adaptive system called Dynamite.

The chapter is outlined as follows. Section 7.2 describes the current hardware and software trends in cluster and metacomputing. In Section 7.3 the Polder metacomputer framework is introduced. This section gives a global perspective of the research goals we are aiming for. The next sections, Section 7.4 and Section 7.5, present the main contribution of this chapter, namely the design and implementation of a local-area load balancing facility incorporated within a message passing library. The ideas and design of a local-area scheduler, i.e., process monitoring and migration decision, are presented briefly. The design and implementation of parallel process migration and restart are described in detail. A series of experiments and results are presented in Section 7.6. Finally, Section 7.7 discusses the results and observations of the Dynamite environment, and concludes with suggestions for future work.

### 7.2 Background and Design Aspects

The current developments in high performance cluster computing and metacomputing are moving along two axis: hardware and software. The hardware development of parallel supercomputing and modern networks/clusters of workstations are directing to the same point on the horizon. The compute nodes in the parallel supercomputer are the same processors found in workstations, and the performances of the distinguished proprietary interconnection networks are attained by independently available network interfaces such as Gigabit Ethernet, Fibre Channel, HIPPI, or Myrinet. Progress in wide-area networking, e.g., SONET and ATM, motivated the development of software infrastructures that smoothly integrate distant distributed resources into a metacomputer that enables the coordinated implementation of high performance applications.
7.2.1 Trends in Hardware

The development of high speed networks, both for local-area and wide-area networks, has triggered a refocus on the hardware used in high performance computing, and in particular a refocus on distributed memory architectures. For example, the massively parallel processors (MPPs) that are used to solve large computational problems, are distinct by their proprietary message passing networks, i.e., communication backplanes specifically designed for a family of MPPs. With the advent of fast network interfaces that are generally available, like (switched) Gigabit Ethernet, Fibre Channel, HIPPI*, and Myrinet, the same large computational problems can be solved effectively on clusters of workstations connected by a local-area network (LAN). In particular Myrinet is an outstanding example of how technology used for communication and switching in MPPs has evolved to a high speed LAN.

The availability of high speed LAN has initiated a number of research projects to build parallel supercomputers made of “commodity off the shelf” (COTS) components. Although the projects described below also cover software issues, their main focus is the implementation of a parallel supercomputer.

The Beowulf project (Warren et al. 1997) aims to develop a parallel computer architecture based upon Pentium Pro processors and switched Fast Ethernet communication links (i.e., switched Fast Ethernet is not used as a broadcast medium, but rather as a point-to-point interconnection fabric giving the full 100 Mbit/s bandwidth). In addition with the availability of powerful, free operating systems (Linux, FreeBSD) and message passing interfaces (MPI), the Beowulf project realized a low-cost commodity parallel computer. With a 16-node parallel computer a sustained performance of one Gflop/s has been obtained on scientific applications. A number of Beowulf offsprings have been build, among which the 140-node DEC Alpha cluster Avalon, the 276-node DEC Alpha cluster Jet (interconnected with Myrinet), and the 56-node (dual processor) Pentium II cluster SWARM. The Avalon cluster achieves 12.83 Gflop/s running a 64 million particle molecular dynamics simulation (SPaSM) on 70 nodes (Warren et al. 1998).

An interesting initiative that combines both high speed LAN and WAN interconnections in the implementation of a high performance computing platform is the Distributed ASCI Supercomputer (DAS)*. The DAS is a 200-node wide-area distributed system built out of four Myrinet-based Pentium Pro clusters. The four clusters are located at four universities: Free University Amsterdam, University of Amsterdam, Delft University of Technology, and University of Leiden.

Each node contains a Pentium Pro, 128 MB RAM, a 2.5 GB local disk, a Myrinet interface card, and a Fast Ethernet interface card. The nodes within a local cluster are connected by a Myrinet SAN network (SAN stands for system area network), which is used as a high speed interconnection, mapped in user-space. Fast Ethernet is used as the operating system network for NFS services,

---


http://www.asci.tudelft.nl/das/das.shtml or http://www.cs.vu.nl/das/
etc. The four local clusters are connected by an ATM wide-area network, so the entire system can be used as a 200-node wide-area distributed cluster (see Fig. 7.1). The system runs the Linux operating system.

![Diagram of DAS Architecture](image)

Figure 7.1: Overview of the DAS Architecture. Four local-area Myrinet clusters are connected by an ATM wide-area network.

The DAS distributed supercomputer, with its high speed local-area and wide-area interconnections, can be regarded as a prototypical metacomputer architecture for the near future. In this respect, the DAS architecture provides a unique experimental testbed for research in metacomputer software infrastructures.

### 7.2.2 Trends in Software

New technologies in wide-area networks have resulted in a new impetus to research directed to provide coordinated network services. The feasibility of wide-area high speed network technology (e.g., ATM, but also HIPPI and IP over SONET) has been demonstrated by the implementation of network testbeds including BERKOM, CASA, Abilene, and vBNS (Abilene and vBNS take part in the Internet2 consortium). The aggregation of distributed and high performance resources on high speed networks will change the perspective on distributed computing and have an impact on the development of scientific applications. In a similar way as parallel computing enabled scientists to solve computational problems that could not be obtained efficiently by sequential computing, aggregated distributed resources can engage larger computational power to a single application.

Although the hardware developments in high speed networks are impressive, the services provided to use the aggregated distributed resources in a coordinated manner are still in their infancy. To fully exploit the potential of distributed resources on coordinated networks, a software infrastructure must be developed to provide easy to use and transparent access to the resources. This software infrastructure, the metacomputer (Smarr and Catlett 1992), manages the complexity of the underlying physical system for the user. The key observation in metacomputing environments is that with the current conceptual
model, interacting autonomous hosts are stretched into a regime for which they were not designed. This has resulted in a collection of partial solutions without coherence and scalability. The challenge is to provide an integrated foundation that hides the underlying physical infrastructure from users and from the majority of programmers. By seamlessly integrating the diverse computational resources, the metacomputer provides a platform that fulfills the requirements of a new class of resource-intensive applications.

Two projects that are exemplary for the current trends in metacomputing research are Legion and Globus. A prototype of the Legion metacomputer and preliminary versions of Globus components have been demonstrated successfully as part of the I-WAY network experiment (DeFanti et al. 1996). Globus and Legion are currently used to provide the infrastructure for the National Technology Grid (Stevens et al. 1997; Foster and Kesselman 1998).

Legion is a metacomputer project designed to provide users with a transparent interface to the available resources, both at the programming interface level and at the user level (Grimshaw and Wulf 1996). Legion uses an object-oriented framework that enables a coherent solution to problems like access support, location, fault transparency, inter-operability, security, etc. The objects, written in either an object-oriented language or other languages such as C, will interact with other objects via well-defined interfaces. The use of objects allows for substantial flexibility in the semantics of user applications; a user is able to select both the kind and level of functionality, and make their own trade-offs between function and cost (e.g., the level of security in authentication).

The Globus (Foster and Kesselman 1997) project addresses the metacomputing challenge by a vertically integrated treatment of application, middleware, and network. In the Globus perspective, metacomputing can build on distributed and parallel software technologies, but also requires significant advances in mechanisms, techniques, and tools. The metacomputing software problem is approached from the bottom up, by developing basic mechanisms such as communication, authentication, network information and data access. These low-level components define a metacomputing abstract machine on which can be constructed a range of alternative infrastructures, higher-level services, and applications.

The long term goal of the Globus project is to construct an integrated set of higher-level services that enable applications to adapt to heterogeneous and dynamically changing metacomputer environments. The adaptive applications are able to configure themselves to fit the execution environment and optimize the performance.

Essential to the success of metacomputing is careful scheduling. Generally, there are two performance optimization objectives in wide-area systems: high performance computing (reducing turnaround time of jobs) and high throughput computing (e.g., maximize the aggregate amount of work per time period). Given one of these two goals, the scheduling process must decide where a job and its constituent tasks will run. The objectives and issues that must be addressed by a wide-area scheduling system are more complex than in local clus-
The delicate interplay of the wide-area scheduler with the local site schedulers is one of the research interests in the Polder metacomputer project, which is presented in the next section.

### 7.3 The Polder Metacomputer Experimental Framework

The Polder metacomputer initiative (Overeinder and Sloot 1997; van Halderen et al. 1998) is an ambitions project that aims to provide an experimental framework for metacomputer design tradeoffs and gradually build a metacomputer environment that organizes heterogeneous distributed resources into one single computing environment with a uniform access. By its distributed nature, the resources are administered by local authorizing resource managers. Therefore, the Polder metacomputer must incorporate existing management software concerning resource control, access control, accounting and monitoring while supporting the multitude of hardware platforms present within the distributed system.

In the Polder metacomputer experiment different ways of use of metacomputing are addressed: high performance computing, high throughput computing, multi-site computing and automatic task balancing for dynamic resources. Each of these different usages of the metacomputing environment has its own requirements with respect to the services provided by the metacomputer. The underlying mechanisms should be flexible and generic in order to efficiently support these different requirements in services. To tackle these problems, a number of subprojects have been initiated to deal with issues like metacomputer access and job submission, wide-area and local scheduling, load balancing, and scalable I/O. These subprojects are performed by the different participants in the Polder initiative, among which the University of Amsterdam, NIKHEF (Amsterdam), Delft University of Technology, University of Wisconsin–Madison, and Paderborn Center for Parallel Computing.

Some of the issues concerning metacomputer access and job submission, wide-area scheduling, and local-area load balancing are discussed in the next section. Within the MOL partner project, the PLUS lightweight communica-
tion interface (Brune et al. 1997b) addresses inter-operability between heterogeneous platforms and different message passing layers. PLUS encapsulates message passing specific communication primitives (e.g., \texttt{MPI\_Send}, \texttt{pvm\_send}) and enables inter-operability between MPI and PVM applications.

### 7.3.1 Resource Management in the Polder Metacomputer

The efficiency of a metacomputing system can be viewed in two different ways. For high performance computing (HPC) a parallel job perspective is taken. The system performance is defined in terms of the turnaround time of highly demanding parallel jobs. In the view taken by high throughput computing (HTC), the performance of the system is mainly defined in terms of the number of jobs that are processed within a certain period of time.

The global resource management structure of the Polder metacomputer model is depicted in Fig. 7.2. The structure determines how the heterogeneous distributed resources are presented to the metacomputer user or application. On the base-level there are resources (e.g., workstations, MPPs, or I/O devices) administered by a local resource manager (e.g., Condor, Codine, or LSF). The aggregated local resources (that is, at the base-level the resources administered by the local resource manager) are represented by self-describing active agents. These agents (in Fig. 7.2 the entities in the shaded area) describe the type of resources, amount of memory, disk space, connectivity, etc.—the agent is essentially not limited in its descriptive plurality. The agents can be aggregated into a new agent, and hence represent a larger set of distributed resources. The aggregation of agents and the information advertised by the agents can reflect local authorization decisions. Although the organization of the agents is hierarchical, the perspective to resources is one-dimensional; that is, a unified view to the heterogeneous distributed resources on a coordinated network.

![Figure 7.2: The Polder metacomputer global resource management model. The software infrastructure (active agents) organizes the distributed resources to a metacomputer.](image-url)
The Polder metacompeter access interface is distributed and WWW-based to allow for a scalable, flexible and generic interface that interacts with the resource agents. The wide-area resource management actually takes place at the metacompeter access interface. Upon job submission via the access interface—with the job requirements being specified—the agents start with bidding on the job. In accordance with the wide-area scheduling policy, one of the agents offers the best fit on the job requirements. The job and its constituent tasks are allocated to the resources in coordination with the local resource manager. In this top-down approach, the wide-area scheduler determines the resources assigned to a job, and direct the local resource managers to actually allocate these resources.

Wide-area scheduling is a complex problem and subject of various research projects. Within the Polder metacompeter project a simulation model of the resource management infrastructure has been developed to allow for rapid prototyping and evaluation of scheduling strategies (Santoso et al. 2000). Experiments with scheduling strategies under strict conditions can be instrumented on top of the resource management simulation model, which is essential for validation with theoretical models. After a scheduling algorithm has been thoroughly evaluated, it can be integrated within the metacompetering environment.

The previous discussion did not mention the scheduler for dynamic load balancing in a local-area cluster. This subject is presented in Section 7.5.1, where the Dynamite local-area scheduler is introduced.

### 7.3.2 The Curse of Dynamics

In general the resources in the metacompetering environment are not exclusively allocated to one user or application, that is, resources are often shared among users and applications. Consequently, changes in the distributed system such as variation in demand of processor power, variation in number of available resources, or dynamic changes in the runtime behavior of the application, hamper the efficient use of the metacompetering environment.

Consider, for example, an application that after a straightforward domain decomposition, is mapped onto the processors of a parallel architecture. If the hardware system is homogeneous and allocated to only one application program, then the execution will run balanced until completion: we have mapped a static resource problem to a static resource system. However, if the underlying hardware system is a cluster of multi-user workstations we run into problems because the available processing capacity per node may change: in this case the static resource problem is mapped to a system with dynamic resources, resulting in a potentially unbalanced execution. Things can get even more complicated if we consider the execution of an application with a dynamic runtime behavior on a metacompeter environment, i.e., the mapping of a dynamic resource problem onto a dynamic resource machine. The notion of redundant decomposition has been posed by de Ronde et al. (1996) to introduce sufficient richness in parallel tasks to make a balanced workload in such a dynamic resource machine possible.
One way of dealing with this dynamically changing resource requirement would be to dynamically rebalance a job and its (parallel) constituents by migration of processes from overloaded to under-loaded resources at runtime. If the dynamic load balancing occurs locally, the wide-area scheduler does not participate. However, the local resource manager might request the wide-area scheduler that a job be re-scheduled elsewhere. The next section describes the design and implementation of these functionalities that are needed for dynamic load balancing, i.e., process migration of running (parallel) jobs.

![Diagram showing task allocation of static and dynamic applications and resources.](image)

**Figure 7.3:** Task allocation of static and dynamic applications and resources.

### 7.4 Dynamite: Process Migration in Message Passing Environments

Process migration support can be incorporated at two operation levels: *operating system level* and *user level*. In operating system level implementations the resource management facilities are supported by the OS kernel. Examples of such systems are Mach (Milojicic et al. 1993), Sprite (Dougatis and Oosterhout 1991), and MOSIX (Barak and La'aden 1998). User level designs and implementations of adaptive systems include dynamic resource management facilities by providing their own dynamic load balancing runtime support. Examples of user level designs are Condor (Litzkow et al. 1988) for sequential, and MPVM (Casas et al. 1995) for parallel application systems.

In our project we have the following design constraints for the process migration facility:

- since we assume that the major computational resource is a scalable cluster environment, the application programming model must be based on message passing;
- it is essential we support a platform independent operating system, therefore the operating system should be Unix;
- by hiding the complexity in libraries, the dynamic load balance runtime support system must be incorporated at user level.
Furthermore, the design of a self-contained experimental environment for dynamic load balancing of parallel application systems should include at least the following three components: (i) parallel programming environment, (ii) parallel runtime support system, and (iii) checkpointing/migration facility. The parallel programming environment enables the programmer to decompose the application problem into parallel subtasks. The parallel runtime support system allows for the parallel execution of the parallel application system; and the checkpoint/migration facility extends the runtime support system with functionality necessary for dynamic load balancing.

The first two facilities are provided by the PVM system (Sunderam et al. 1994). The PVM system includes an application programming interface for parallel program development and a runtime support system to allow for parallel execution of the application. The task checkpoint/migration functionality extension must be integrated with the PVM runtime support. The choice to use PVM as the basic parallel programming environment is motivated by the free availability of the source code and the extendibility of the runtime support. The application programming interface incorporates the dynamic addition and deletion of hosts (resources) and processes.

These design constraints have motivated the development of Dynamic PVM or DPVM for short (Dikken et al. 1994; Overeinder et al. 1996). The development of DPVM is now continued in the Dynamite project (van Albada et al. 1999; Iskra et al. 2000). Dynamite is an acronym for Dynamic Task Migration Environment, and currently supports PVM-based programs only. However, the principles of Dynamite should be easily portable to MPI (MPI Forum 1998). Both message passing environments are generally available on many different platforms and allow for the extension of process migration into their libraries. Although the checkpoint/migration design considerations are equal for PVM and MPI, there are some differences in the implementation. PVM (as basis for DPVM/Dynamite) is a message passing environment that also includes process creation and termination, and other resource management functionalities such as primitives for the allocation and deallocation of resources. The MPI-1.1 definition however, does not include any hooks for resource management functionalities required with process migration. This has to be included in the MPI runtime support system, but must be transparent to the application programmer. The MPI-2 definition includes process management, but no resource control as in PVM. MPI-2 assumes that resource control is provided externally—probably by computer vendors, in the case of tightly coupled systems, or by a third party software package when the environment is a cluster of workstations.

In the following discussion we briefly outline aspects of the PVM system and present the design issues to incorporate checkpoint/migration facilities in Dynamite.
7.4.1 The PVM System

The PVM (Parallel Virtual Machine) system presents an integrated environment for heterogeneous concurrent computing on a network of workstations. The computational model is process-based, that is, the unit of parallelism in PVM is an independent sequential thread of control, called a task. A collection of tasks constituting the parallel application, cooperate by explicitly sending and receiving messages to one another. The support for heterogeneity permits the exchange of any data type between machines having different data representations.

The PVM system consists of two parts: a daemon, called pvmd, and a library of PVM interface routines, the pvmlib. The PVM daemon and library enable a uniform view of the network of workstations, called hosts in PVM, as a parallel virtual machine.

Each host in the virtual machine is represented by a daemon that takes care of task creation and dynamic (re-)configuration of the parallel virtual machine. PVM tasks are assigned to the available hosts using a round-robin allocation scheme. Once a task is started, it runs on the assigned host until completion, i.e., the task is statically allocated.

The PVM library implements the application programming interface that includes primitives for process creation and termination, host addition and deletion, coordinating tasks, and message-passing primitives. The underlying communication model can be classified as asynchronous message-passing, where the messages are buffered at the receiving end. An important aspect of the communication model is that the message order from each sender to each receiver in the system is preserved. The PVM message-passing interface supplies both point-to-point communication primitives and global communication

![Diagram](image)  

Figure 7.4: The PVM system composed of daemons and tasks.
7.4 Dynamite: Process Migration in Message Passing Environments

primitives based on dynamic process groups. To enable the use of heterogeneous host pools, messages can be encoded using an external data representation (see XDR (Sun Microsystems, Inc. 1987)).

A relevant issue in the context of the forthcoming discussion, is message routing. PVM supports two routing mechanism for messages, namely indirect and direct routing. By default, the messages exchanged between tasks are indirectly routed via the PVM daemon. With indirect routing, a task sends the messages first to the local PVM daemon. The local daemon determines the host on which the destination task resides, and sends the message over the User Datagram Protocol (UDP) transport-layer to the responsible daemon. This daemon eventually delivers the message to the destination task. For example in Fig. 7.4, an indirect path from task \( a_1 \) to \( b_2 \) goes via pvmd A and pvmd B. Direct message routing allows a task to send messages to another task directly over a Transmission Control Protocol (TCP) link, without interference of the PVM daemons and thereby enhancing communication performance (see for example the TCP connection between tasks \( a_1 \) and \( c_1 \) in Fig. 7.4).

7.4.2 Design Aspects of Process Migration in Dynamite

Process migration (operating system level and user level) is realized by the movement of an active process from one machine to another in a parallel or distributed computing system. The process is suspended and detached from its environment, its state and data (the checkpoint) transferred to the destination host, where it is restarted and attached to the destination environment. The major requirement for providing a migration facility is transparency: the execution of a process should proceed as if the migration never took place. In parallel application systems like PVM applications, this transparency should hold also for the migrated process's communication partners. The application programs then do not have to take into account for possible complications of checkpointing and migration.

From the requirements defined above, it follows that Dynamite must incorporate a checkpoint/migration facility and location independent task identifiers, in order to support transparent process migration. The checkpoint/migration functionality in Dynamite is based on the ideas of the facility provided by the Condor system. Dynamite extended the checkpoint protocol to safely checkpoint communicating parallel tasks without loss of messages. The location independent task identifiers, or virtual task identifiers, guarantee a unique name space for tasks independent of their location. Thus the same task can be addressed with the same task identifier after migration. Compare the virtual task identifiers with virtual memory addresses: the virtual memory address can be mapped to different physical addresses during the execution of a program.
7.5 Implementation Aspects of the Dynamite Environment

This section describes the extensions to PVM that are necessary to support dynamic load balancing within the runtime support system. In order to implement task migration, see Section 7.5.3, functionalities in the PVM daemon *pvm* and library *pvm* need to be enhanced with checkpoint/migration mechanisms.

It is essential to note that the intertask communication, viz., message routing by the *pvm*, is strongly affected by the added functionality of task migration. Therefore, we need to develop a methodology to guarantee the transparency and correctness of this intertask communication.

The extensions to the *pvm* and *pvm* must not change the PVM programming interface and semantics, such that source code portability is guaranteed. The packet routing by the *pvm* ensures migration transparency. With this approach, any standard PVM application can be linked and executed with the Dynamite system without a modification to the source code of this application, thus hiding the complexity for the end-user.

7.5.1 The Scheduler

Although the scheduler is not considered an integral part of Dynamite, its role and interface is mentioned here. In line with the top-down perspective of the wide-area and local site scheduler, the Dynamite scheduler resides beneath the local site scheduler. The local site resource manager is the authority that allocates the resources for the Dynamite cluster. The Dynamite scheduler acts as a resource manager within this Dynamite cluster, that is, it decides when to migrate a task and to which host it is moved. In addition, the Dynamite scheduler can request or relinquish resources in interaction with the the local resource manager.

In this scenario, the Dynamite scheduler largely determines the efficacy of the Dynamite system in its aim for load balancing. The development of good algorithms or heuristics for load balancing is a study in itself and is beyond the scope of this thesis. The current scheduler decides on (re-)allocation of processors for tasks, based on gathered load information of the workstation pool. The scheduler was developed in collaboration with the University of Paderborn within the Dynamite project (van Albada et al. 1999).

The scheduler consists of two functional components: *resource monitoring* and a *migration decision*. The resource monitors and migration decision are implemented as normal PVM tasks. This approach makes the incorporation of new scheduling strategies flexible and provides for a flexible experimental platform for studying the effectiveness of the different load balancing disciplines. A consequence of implementing the scheduling components as PVM tasks, is that an additional interface must be provided to enable the migration decider to interact with the Dynamite system, in particular with the PVM daemons.
To this end, the *pvmlib* is extended with an interface routine, `pvm_move(tid, host)`, that initiates the migration of task `tid` to the specified host `host`.

**Resource Monitoring**

The monitoring subsystem keeps track of the load on the hosts and the communication between the tasks. In order to make migration decisions, the following information is administered:

- nominal capacity on each node (CPU, memory, disk space);
- current load on each node;
- required capacity for each task;
- network capacity for each task;
- communication pattern for each task.

Each of these items can be measured at execution time by the monitoring subsystem, but we assume that node capacity and network properties are sufficiently stable that they can best be specified beforehand by the system administrator (see Brune et al. (1997a) for further details).

Because of the assumed dynamic behavior of the application and the system load, the other items need to be obtained by the monitoring subsystem. Information about load and capacity must be collected from all nodes of the cluster, also those where currently no task of the parallel application is running. This is accomplished by running a small monitoring slave on each node (Fig 7.5).

The statistics obtained by the monitor slaves are sent to the monitor master process that is not only responsible for maintaining the whole cluster statistics, but also has to make migration decisions. The information on communication patterns is obtained directly from the DPVM environment. Therefore, DPVM has been enhanced by a message monitoring thread. This thread keeps track of each message sent and received. These communication statistics are also sent to the monitor master process.

**Migration Decider**

The migration decider is the main part of the scheduler thread that is executed periodically by the monitor master process. Based on the monitored data, the migration decider has to judge where and when to migrate a task from an overloaded node. Additionally, the task to be moved causes some constraints on the migration decision. Therefore, the master load monitor has to supply some normalized values about the attributes CPU, memory, and disk swap space of each node and additionally the available network capacity.

The increasing interest in distributed computing has lead to intensive scientific research in load balancing schemes for distributed memory systems, see for example Decker et al. (1998). Because not every load balancing scheme is
applicable to every application, the migration decider has been designed in a flexible manner to support a broad range of applications. For the first prototype we have implemented a straightforward solution with a greedy-like algorithm and a constraints list. The details of the algorithm are beyond the scope of the thesis, and can be found in van Albada et al. (1999).

### 7.5.2 Consistent Checkpointing Through Critical Sections

To implement dynamic load balancing by task migration, the runtime support system must be able to create an image of the running process, the so-called *checkpoint*. A checkpoint of an active process consists of the state and data of the process, together with some additional information to recreate the process. To incorporate file I/O migration, the state vector also includes information about open files together with their modes, file descriptors, etc.

A complication with checkpointing communicating PVM tasks, is that the state of the process also includes the communication status of the socket connections. Thus, to save the state of the process, the interprocess communication must also be in a well-defined state. Since suspension of the related communicating task is not desirable, the task should not be communicating with another task at the moment a checkpoint is created. To prohibit the creation of process checkpoints during communication, we apply the notion of *critical sections* and embed all interprocess communication operations in such sections. Checkpointing can only take place outside a critical section. When a checkpoint signal arrives during the execution of a critical section, the checkpointing is deferred.
The checkpointing functionality is implemented in the dynamic loader, to which the following changes have been made:

- it can handle a checkpoint signal (SIGUSR1);
- it can treat a checkpoint file just like any other executable;
- it wraps certain system and library calls:
  - for open files (e.g., open, write, creat),
  - for memory allocation (mmap, munmap, and mremap†);
- cross-checkpoint data is stored separately.

When a checkpoint signal is sent to the process, control is passed to the checkpoint handler. First of all the current signal status and the contents of the processor registers are saved. Next, the name and location of the checkpoint file is determined. The checkpoint file is placed in a directory which must be accessible from all the nodes in the cluster. After saving the signal mask and the status of open files, the checkpoint itself is created. Basically, the checkpoint handler saves the address space of the process: the text segment, the data segment, the stack, the dynamically allocated pages, and the shared libraries used.

Restarting the checkpoint is realized by running the checkpoint binary. When the binary is run, the dynamic loader is executed first. As soon as the dynamic loader is finished, control is passed to the actual program. The Dynamite dynamic loader has some extra functionality included. One of the first things this loader tries to locate is the special section containing the name of the checkpoint file. If such a section is present, it knows that it is restoring form a checkpoint, and specialized subroutines take care of a proper handling of the process’ segments. The signal status and processor registers are restored, and the process resumes its execution at the point where it was checkpointed.

The wrapped system calls enables the checkpointing/restoring facility to deal with open files. Basically, these wrapper routines invoke the original C-library calls, doing some extra administration, which allow the open file connections to be restored properly. The system call mmap is wrapped as the memory allocated by this system call must be restored too. This implies that all the memory allocations done by mmap have to be monitored as well. The cross-checkpoint storage is used to preserve data structures across a checkpoint/restart, such as the mapping of the shared libraries used by the process or the status of the open files.

### 7.5.3 The Migration Protocol

The main objective of the Dynamite migration facility is transparency of the migration protocol, i.e., to allow for the movement of tasks without affecting

†Linux specific.
the operation of other tasks in the system. With respect to the individual task
selected for migration this implies transparent suspension and resumption of
execution: the task has no notion that it is migrated to another host, and the
communication can be delayed without failure, triggered by migration of one of
the tasks.

In the task migration protocol we distinguish four phases:

1. create new process context at destination host;

2. the new routing information is broadcasted;

3. disconnect task from its local pvmd and checkpoint task;

4. move task to its new host, and restart and reconnect the task to its new
   pvmd.

The first step in the migration protocol is the creation of a new process
context at the destination host by sending a message to the pvmd representing
that host. A new PVM task context is created, so that the PVM daemon can
accept any messages addressed to the migrating task and temporarily store
them.

Next, all the PVM daemons but the source and the destination one are noti-
fied that a migration is about to take place. The daemons update their routing
information, so that messages sent via the daemons to the migrating task are
sent to the destination node, see also Section 7.5.4.

The checkpoint phase (the third step) is executed on the source node, i.e.,
the node the task runs on before the migration takes place. First, routing
information is updated, so that any messages sent to the migrating task via the
PVM daemon are forwarded to the destination node instead of being delivered
locally. Finally, the task finds out that it is to be migrated. A SIGUSR1 signal
is sent to the task by the PVM daemon. Control is passed to the checkpoint
signal handler in the Dynamite dynamic loader. However, before the actual
checkpointing takes place, the communication between this task and its PVM
daemon and the other tasks has to be flushed. The signal handler invokes a
DPVM function that reads all the available data from all the connections, closes
the task connections and sends the final TM_MIG migration message to the local
PVM daemon. Subsequently, the checkpoint handler creates the checkpoint file
and terminates the process.

In the final restart stage, executed on the destination node, the task is
restarted at the new location using the spawn_task function. In the process
of restarting the task from the checkpoint file, the dynamic loader invokes a
DPVM function that reconnects the restored task to the PVM daemon on the
destination node. Control is passed back to the application code, and the PVM
daemon can finally deliver all the messages addressed to the migrating task
which it had to store during the migration.
7.5.4 Packet Routing and Direct Connections

In message passing environments like PVM, the process identifier or task identifier, task id for short, is a unique identifier which serves as the task’s address and therefore may be distributed to other PVM tasks for communication purposes. For this reason the task id must remain unchanged during the lifetime of a task, even when the task is migrated.

**Indirect Connections**

By default, PVM tasks use indirect connections to communicate with each other. In this mode, messages between tasks are routed through two PVM daemons, local to the source and destination tasks. As a consequence, PVM application tasks do not have any remote network connections open, their only communication channel is with the daemon.

As the task ids must remain unchanged over migrations, this has implications for the packet routing of messages. The task id contains the host identifier at which the task is enrolled and a task sequence number (local to the host). This information is used by the PVM daemon to route packets to their destination, i.e., to the appropriate PVM daemon and task. When a task is migrated to another host, this routing information is not correct anymore. Therefore, an additional routing functionality must be incorporated in the PVM daemon routing software in order to support the migration of tasks.

An important design constraint is that the routing facility must be highly efficient and should not impose additional limitations on the scalability. This is accomplished by maintaining in the PVM daemons the routing tables for

![Figure 7.6: Routing tables keep track of the migrated tasks.](image-url)
migrated tasks, which contains the current locations of migrated tasks (see Fig. 7.6). These routing tables are consulted for all inter-task communication. Upon migration of a task, the routing table of the PVM daemons are updated to reflect the change in location of the migrated task. How this is accomplished was described in the previous section. Figure 7.6 depicts the migration of a task attached to pvmdb and the subsequent routing table update.

Direct Connections

To improve efficiency, an alternative direct communication mode is available on application request. In this mode, tasks that wish to communicate with each other can establish a direct TCP/IP network connection between themselves.

Special care must be taken when migrating a task that has direct connections with other tasks, or else messages that are being processed or are cached in the kernel buffers will be lost during the migration.

In the first two stages of the migration protocol, along with updating the routing information, DPVM notifies all the PVM tasks that a migration is about to take place. Because it is important that the tasks reply in a timely manner, PVM daemons also send SIGURG signal along with the migration notification messages. It is the responsibility of the asynchronously invoked signal handler function to handle this message.

In the checkpoint stage, the checkpoint signal handler of the migrating task sends an end-of-connection (TC_EOC) message via all the open direct connections. The remote tasks read all the data from the connection until they receive TC_EOC, at which point they send the TC_EOC message back. The migrating task reads all the data on its side of the connection, and closes the connection upon receipt of TC_EOC. As a result of the close on the migrating side, the remote tasks receive EOF at this point, and can close the connection on their side.

Any messages that were only partially sent by the migrating task are fully resent after the task is restarted. Any messages that were partially sent by the remote tasks are fully resent via PVM daemons, i.e., indirectly. The direct connection is reestablished as soon as the migrating task restarts and there are new messages to be sent.

7.6 Performance Evaluation

Originally, DPVM was developed on a network of IBM AIX/32 machines (Dikken et al. 1994). Further development of DPVM and later Dynamite has been accomplished on Sun workstations operating under Solaris and Solaris (Vesseur et al. 1995), and PCs running Linux. Dynamite currently supports applications written for PVM 3.3.x, running under Solaris/ UltraSPARC 2.5.1 and 2.6 and Linux/i386 2.0 and 2.2 (Iskra et al. 2000). From the user's perspective, all that is needed is to relink the application with Dynamite's version of the PVM libraries and with the Dynamite dynamic loader.
The stability of the Dynamite environment has been assessed by a series of tests under Solaris and Linux. Dynamite has been used to make over 2500 successful migrations of large processes (over 20 MB of memory image size) of a commercial PVM application PAM-CRASH (Clincemallie et al. 1997) using direct connections, after which the application finished normally.

The performance evaluation of Dynamite is accomplished by four different experiments: (i) communication performance evaluation, (ii) migration overhead evaluation, (iii) evaluation by the NAS Parallel Benchmark (NPB) suite, and (iv) the GRAIL simulation application. The focus of the first two experiments is on the increased overhead of Dynamite compared to standard PVM. The third series of experiments explores the efficacy of the Dynamite environment (including scheduler) for a number of parallel benchmark kernels with different computation and communication behavior. The last experiment measures the performance of Dynamite for a large scientific finite-element model simulation, called GRAIL (de Ronde et al. 1997b).

### 7.6.1 Measuring DPVM Communication Overhead

The basic communication properties of a message-passing system, such as latency time and throughput bandwidth, can be measured by the well-known ping-pong experiment. With the ping-pong experiment, series of messages of different sizes are sent between two tasks. The initiating task sends a message to the second task, the second task receives the message into a buffer, and immediately returns it to the initiating task. Half the time of this message ping-pong is recorded as the time $t$ to send a message of length $n$.

The ping-pong experiments are performed for both the standard PVM implementation as well as the DPVM implementation, with message size ranging from 1 byte to 100 KB. The experimental results obtained for Solaris and Linux are presented in Fig. 7.7 and Table 7.1. The Solaris network consists of Sun Ultra 5/10 workstations interconnected by switched 100 Mb/s Fast Ethernet. The Linux cluster is equipped with Pentium Pro 200 MHz nodes and is also interconnected by switched 100 Mb/s Fast Ethernet.

From Fig. 7.7, we can see that in all cases DPVM has (little) increased communication costs. This stems from two factors:

- signal blocking/unblocking on entry and exit from PVM functions (function call overhead);
- extra header in message fragments (communication overhead).

The first factor adds a fixed amount of time for every PVM communication call, whereas the second one increases the communication time by a constant percentage. For small message sizes, the signal blocking/unblocking factor dominates over the extra header information factor, since there is little communication and the message is not fragmented into multiple packets. The DPVM overhead for 1 byte messages ranges from 25% for direct communication under Linux to 4% for indirect communication under Solaris. Although the DPVM
direct communication overhead is significant, we must point out that it represents the worst-case scenario, as the relatively fast direct communication is hurdled by a fixed signal blocking/unblocking overhead, resulting in a large overhead percentage.

As the message size increases, the overhead of extra message header information in message fragments becomes more dominant. The DPVM overhead
7.6 Performance Evaluation

<table>
<thead>
<tr>
<th></th>
<th>PVM</th>
<th>DVM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>latency (msec)</td>
<td>throughput (MB/s)</td>
</tr>
<tr>
<td>Solaris</td>
<td></td>
<td></td>
</tr>
<tr>
<td>indirect</td>
<td>1.57</td>
<td>1.64</td>
</tr>
<tr>
<td>direct</td>
<td>0.59</td>
<td>4.01</td>
</tr>
<tr>
<td>Linux</td>
<td></td>
<td></td>
</tr>
<tr>
<td>indirect</td>
<td>1.01</td>
<td>1.73</td>
</tr>
<tr>
<td>direct</td>
<td>0.42</td>
<td>3.42</td>
</tr>
</tbody>
</table>

Table 7.1: Latency and throughput performance of PVM and DPVM for Solaris and Linux.

for 100 KB messages eventually becomes 2% for indirect communication and 8% for direct communication under Solaris, and is 4% for both direct and indirect communication under Linux. The extra message header overhead can be tuned by changing the PVM/DPVM message fragment size. By increasing the packet size, a smaller number of fragments are necessary per message sent. However, the overhead for small messages increases, as the packets sent between the tasks will be largely empty.

7.6.2 Checkpoint and Migration Overhead

The checkpoint and migration experiments measure the DPVM load balance overhead for various process sizes. A simple ping-pong type program communicating once a few seconds via direct connection was used, process size was set with a single large malloc call. The experiments measure the executing time of each of the four migration stages (see Section 7.5.3). The average results over five experiments for Solaris and Linux are shown in Fig. 7.8. The standard deviation σ is smaller than 10%.

The results from the experiments show that the major part of the load balancing overhead is due to checkpointing and restarting. The migration protocol and connection flushing amount together to approximately 0.01–0.03 seconds, and are not depicted in Fig. 7.8. The checkpoint and restart times are limited by the speed of the shared file system. On the two platforms used, the Solaris network and Linux cluster, the bandwidth of the NFS system is 4–5 MB/s over the 100 Mb/s network. From the figure we can see that the checkpoint times under Solaris and Linux approximately increase linearly with process size. However, the restoring phase in Linux takes more or less a constant amount of time, while it grows with the process size under Solaris.

The difference in restore times between Solaris and Linux is due to differences in the implemented memory allocation strategy in malloc. For large memory allocations, Linux creates a new memory segment (separate from the heap) using mmap, whereas Solaris always allocates from the heap with sbrk. When restoring, the head and stack are restored with read (see Section 7.5.3),
which forces an immediate data transfer. However, for the other segments the Linux implementation takes advantage of mmap, which uses a more advanced page on demand technique, delaying network transfer until the data is actually needed. Since the allocated memory region is not needed to reconnect to the PVM daemon, the time it takes to restart the task is constant under Linux. Clearly, delays may be incurred later, when the mmaped memory is accessed and loaded.
7.6 Performance Evaluation

7.6.3 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) is a suite of applications used by the Numerical Aerodynamic Simulation (NAS) Program at NASA for the performance analysis of parallel computers. The benchmark suite consists of five "kernels" and three simulated applications which mimic the computational behavior of large scale computational fluid dynamics applications. A unique property of the NPB is that the applications are specified algorithmically. The implementation of the NPB kernels used in the experiments with DPVM are described in White et al. (1995).

The characteristic behavior of the NPB kernels used in the performance analysis of PVM and DPVM are:

CG The communication patterns in the conjugate gradient kernel are long-distance and unstructured.

EP The embarrassingly parallel kernel is based on a trivial partitionable problem requiring little or no communication between processors.

FT In the 3-D Fast Fourier Transformation, the communication patterns are structured and long distance.

IS Integer sort performs rankings of equally distributed integer keys; the communication is frequent and relatively low-volume, and the pattern of communication is a fully connected graph.

MG The 3-D multigrid solver is characterized by highly structured short- and long-distance communication patterns.

The NPB experiments are performed on eight nodes of the DAS Linux cluster (Pentium Pro 200 MHz and 100 MB/s Fast Ethernet). The nodes are exclusively reserved for the experiments. The NPB kernels are configured to use four computation tasks each, running for approximately 30 minutes in the optimal situation without background load. The number of available nodes exceeds the number of tasks of the NPB kernel. Thus, during the execution of the benchmarks some of the nodes are idle, which allows the Dynamite scheduler to migrate PVM tasks from overloaded nodes to idle nodes.

The NPB kernels are executed and timed for three situations: (i) no background load, (ii) with background load, and (iii) with background load and Dynamite scheduling and migration. Without background load, the compute nodes are fully dedicated to the NPB kernels, and the timings are used as a reference performance. The external background load is generated by running a single computationally intensive process for five minutes on each node used by the benchmark kernel. In this way, one node at a time is overloaded using a round-robin schedule. The performance results of the NPB kernels for the three situations are shown in Table 7.2.

http://www.nas.nasa.gov/NAS/NPB/
Table 7.2: Execution times of the NAS Parallel Benchmarks (in seconds).

<table>
<thead>
<tr>
<th></th>
<th>no load</th>
<th>load</th>
<th>ckpt. size</th>
<th>migrations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>no Dynamite</td>
<td>Dynamite</td>
<td></td>
</tr>
<tr>
<td>CG</td>
<td>1795</td>
<td>3352 (+87%)</td>
<td>2226 (+24%)</td>
<td>19 MB</td>
</tr>
<tr>
<td>EP</td>
<td>1620</td>
<td>1919 (+18%)</td>
<td>1773 (+9%)</td>
<td>14 MB</td>
</tr>
<tr>
<td>FT</td>
<td>1859</td>
<td>2693 (+45%)</td>
<td>2237 (+20%)</td>
<td>31 MB</td>
</tr>
<tr>
<td>IS</td>
<td>1511</td>
<td>1688 (+12%)</td>
<td>1758 (+16%)</td>
<td>41 MB</td>
</tr>
<tr>
<td>MG</td>
<td>1756</td>
<td>2466 (+40%)</td>
<td>1863 (+6%)</td>
<td>17 MB</td>
</tr>
</tbody>
</table>

The performance results in Table 7.2 show that the execution times with background load increase for both situations, whether with or without Dynamite. However, the performance results with Dynamite scheduling and migration significantly improve with respect to the results without Dynamite, reducing the percentage of slowdown by a factor two to six. One notable exception is the IS kernel, for which Dynamite scheduling and migration diminishes the performance results compared to the results without Dynamite. The relative difference in slowdown percentages of the NPB kernels depends on the computation and communication characteristics of the respective benchmark kernel.

Figure 7.9 presents the execution progress of three NPB kernels: conjugate gradient, embarrassing parallel, and integer sort. These three benchmarks each show different computation and communication characteristics, resulting in significant different performance slowdown figures. The figures show the data for one of the tasks of the parallel benchmark. The left column presents the time spent on executing each individual step (ideally, this should be constant), whereas the right column presents the accumulative execution time.

The results for the CG kernel are shown in Fig 7.9(a). The execution of the CG benchmark slows down 87% when subjected to external background load. The considerable slowdown indicates that a substantial amount of execution time must be spent on computation, thus the CG kernel and the external background load compete for computing resources (CPU time). Furthermore, the communication pattern of the benchmark (loosely synchronous, global communication) forces all the other processes to wait for the one lagging behind. The computational intensiveness and the loosely synchronous global communication pattern of the CG kernel makes that the overall performance of the benchmark severely drops. The scheduling capability in Dynamite is able to migrate the CG task from an overloaded to an idle processor; see also the short periods of increased execution time in the left figure of Fig 7.9(a), while the results without Dynamite show constantly increased execution time per step.

The performance results for the EP kernel in Fig 7.9(b) show a different picture. The "embarrassingly parallel" benchmark is based on trivial partitionability of the problem, while incurring no data or functional dependencies, and requiring little or no communication between processors. The external background load significantly hampers the performance of the affected task,
7.6 Performance Evaluation

(a) Conjugate gradient (CG) kernel.

(b) Embarrassing parallel (EP) kernel.

(c) Integer sort (IS) kernel.

Figure 7.9: Execution progress of NAS parallel benchmarks: the time to execute one step (left) and the accumulative time (right).

but has almost no influence on the other tasks (the execution times per step coincidence for the “no load” and “load” figures in Fig. 7.9(b) where other tasks of
Dynamic Load Balancing of Execution Threads

The benchmark are affected by the external background load). The gain of Dynamite scheduling is limited, as the overall performance of the EP benchmark is only partially determined by the individual tasks.

The IS benchmark is in particular a communication intensive application. In Fig. 7.9(c), we see that the IS performance is only slightly affected by the external background load, as most of the execution time is spent on communication: frequent and in large-volume. The pattern of communication is a fully connected graph. The performance of Dynamite is slightly worse than the results without any scheduling and migration. Although the migration decisions of Dynamite’s scheduler are not unreasonable, there is little performance gain that fails to exceed the migration costs, which is rather high in the IS benchmark because of large process size (41 MB).

The large process size of the FT kernel (31 MB) also limits the attainable performance of Dynamite over no scheduling and migration. The slowdown reduction from 45% to 20% would be significant larger if the processes to be migrated were smaller.

7.6.4 The GRAIL Finite-Element Model Simulation

The GRAIL simulation application is a finite element model of a gravitational radiation antenna (de Ronde et al. 1997a). The finite element model is parallelized by decomposition of a finite-element mesh. The parallel finite-element simulation is time-driven, using loosely synchronous communication to exchange the subdomain boundaries between the neighbors. The computation/communication behavior of the GRAIL simulation kernel is characterized as computational intensive with regular communication patterns of large messages.

A series of GRAIL simulation experiments is executed on the DAS cluster to evaluate the effectiveness of Dynamite for a real-world large scientific simulation application. The series of experiments is carried out without external background load for PVM, DPVM, and DPVM with scheduler; and with external background load for DPVM and DPVM with scheduler. The parallel simulation consists of 3 tasks running on 4 nodes. The checkpoint size of the GRAIL tasks is 8.5 MB. The results of the experiments are presented in Table 7.3. The individual results are the average over three experiments, with standard deviation \( \sigma \) smaller than 10%.

<table>
<thead>
<tr>
<th>parallel environment</th>
<th>exec. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 PVM</td>
<td>1854</td>
</tr>
<tr>
<td>2 DPVM</td>
<td>1880</td>
</tr>
<tr>
<td>3 DPVM + sched.</td>
<td>1914</td>
</tr>
<tr>
<td>4 DPVM + load</td>
<td>3286</td>
</tr>
<tr>
<td>5 DPVM + sched. + load</td>
<td>2564</td>
</tr>
</tbody>
</table>

Table 7.3: Execution time of the GRAIL application (in seconds).
The first three rows in Table 7.3 show the increased costs of DPVM and DPVM with scheduler over PVM, if there is no external background load. The execution time increased with 1.5% for DPVM, and is accounted to the critical section locking and the extra communication overhead that is necessary for transparent message routing to migrated tasks. On top of this overhead, DPVM with scheduler and monitoring, i.e., the complete Dynamite environment, accounts for another 2% overhead. As there is no migration of the tasks, this 2% overhead is interpreted as the monitoring overhead.

The fourth and fifth set of experiments include an artificial, external background load. The external background load is a single, CPU-intensive process that runs for 600 seconds on each node in turn, using a round-robin schedule. In the fourth set, the monitoring and scheduling subsystem is not running, and DPVM does not migrate tasks from overloaded nodes. A considerable slowdown of 75% over DPVM without background load is observed. The computational intensive GRAIL simulation is fairly sensitive to the presence of external background load. Furthermore, the loosely synchronous communication between the simulation tasks results in an overall performance of the GRAIL simulation that suffers from the background load, and results in the significant slowdown. The fifth set of experiments combines the external background load with monitoring and scheduling; thus the complete Dynamite environment, including the presence of external background load. The results in Table 7.3 show that Dynamite manages to reduce the slowdown percentage from 75% to 34%. This remaining 34% slowdown is contributed to the following factors:

- the delay before the monitor notices increase in load on the node, and to make the migration decision;
- the non-zero costs of the migration; and
- the master task, which is started directly from the shell, cannot be migrated; when the round-robin schedule of the external background load skips the node with the master task, the slowdown decreases further by 10%.

Figure 7.10 depicts the execution progress of the GRAIL simulation for three sets of experiments: (i) PVM, (ii) DPVM with background load, and (iii) DPVM with monitor and scheduler, and background load. The figure shows the progress in iterations of the GRAIL simulation versus the execution time. The performance figure for PVM is a straight line with the largest angle. The other two lines are the performance figures for DPVM with background load, with and without scheduling, and show some discontinuities due to background load and task migration. Initially, the progress of both experiments is slower than the PVM experiment—as the load is initially applied to the node with the master task, no migrations take place. After approximately 600 seconds, the background load moves to another node. Subsequently, in the case where the monitoring and scheduling subsystem runs, the scheduler migrates the application task from the overloaded node, and the progress improves significantly,
Dynamic Load Balancing of Execution Threads

Figure 7.10: Execution progress of GRAIL for three cases. Note that the plain PVM run was made without an external background load, whereas both DPVM runs were done with such a background load.

...coming close to the performance of PVM without any background load (performance line of DPVM with scheduling shows same angle). For the experiment with DPVM without the monitoring and scheduling subsystem running, there is no observable change in the performance at this point. However, DPVM without scheduler does improve between 1800 and 2400 seconds from the start, that is when the idle node is overloaded. After 2400 seconds from the start, the node with the master task is overloaded again, so the performance deteriorates in both DPVM cases.

Figure 7.11 shows two typical Dynamite execution runs of the GRAIL simulation in a heterogeneous environment. The execution time of both simulation runs is approximately 1:15 hours. The two figures depict the CPU percentage (the CPU time divided by the elapsed time) of the DPVM tasks during the GRAIL simulation. The results are taken from simulation runs with four tasks on a network of workstations (Solaris/UltraSPARC). The host pool consists of ten workstations with different relative speed\(^5\), varying from 500, 900, 1000, to 1150. One can see from the figures that for most of the time, the four tasks are assigned different CPU percentages. As the GRAIL simulation has a loosely synchronous communication pattern, the tasks on the fastest workstation experience longer idle times than tasks on the slower workstations. This translates to lower CPU percentages for the tasks on the faster workstations, and vice versa.

The external background load consists of two components. First, the regular background load generated by other users working at their workstations. Sec-

\(^5\)The relative speed is determined by running a small computation kernel.
Figure 7.11: Two typical DPVM execution runs of the GRAIL simulation with four tasks and a host pool of ten processors. The relative speed of the processors in the host pool varies from 500, 900, 1000, to 1150. The migrations are denoted by the vertical lines.

Second, there is an artificial background load imposed on each workstation in turn for 300 seconds, in a round-robin schedule. These external background load factors bring about the large fluctuations in the CPU percentage in Fig. 7.11. But even in the ideal situation without background load, we observe fluctuations...
in the GRAIL simulation progress due to the heterogeneity of the workstation pool. By this heterogeneity, it is difficult to interpret the figures directly. The scheduler relates the relative speed of the processor with the measured CPU percentage. In this way the scheduler is successful in making sensible migration decisions. The migrations are denoted by vertical lines in the figures. During the GRAIL simulation run in Fig 7.11(a), there were 11 migrations. The first migration was almost at the start of the execution. The second migration around iteration 1900 in Fig 7.11(a) is more prominent. It demarcates the migration of task 2 from the overloaded node to an lightly loaded node, resulting in a significant overall performance improvement. The concerning node was overloaded two minutes before the scheduler made the decision to migrate task 2 (see the period of reduced performance in Fig 7.11(a). A similar situation appears just before the third migration, where task 3 is migrated. Here however, task 3 is moved from a relative fast, overloaded workstation to a moderate fast, slightly loaded workstation, so there is less global performance impact. Other migrations with a global performance impact are migrations five, seven, eight, nine, and eleven. Figure 7.11(b) gives a similar impression of the GRAIL simulation execution. Although twelve migration are shown in the figure, actually fifteen migrations are made. Three of the migrations were double migrations, where two tasks were migrated at the same time (namely migrations one, five, and twelve).

### 7.7 Summary and Discussion

The Dynamite environment provides a robust framework for load balancing, where the runtime support system migrates tasks from a parallel program when necessary. The overhead incorporated with dynamic load balancing is small compared to the possible costs of load imbalance. Experiments show a slight performance penalty in a well-balanced system (less than 5%), but significant performance gains can be obtained for task migration in an unbalanced system. The communication and checkpoint overhead experiments in Sections 7.6.1 and 7.6.2 indicate that the Dynamite system provides efficient task migration support. The experiments in a controlled highly dynamic cluster environment (see Sections 7.6.3 and 7.6.4) exposes the ability of Dynamite to react to changes in the cluster environment and reduce the turnaround time of the applications. The eventual success of Dynamite depends on the scheduling strategy.

The concept of implementing the checkpoint/restart facilities in the dynamic loader and using it to migrate PVM tasks has been proven to work in practice. The architecture of Dynamite is modular so that it can easily be adapted to specific application requirements. For example, it is possible to use just the dynamic loader of Dynamite and get checkpoint/restart facilities for sequential jobs that do not use PVM. Also, in the development phase this modularity is used for experimentation with various migration policies. It is even not required to use the Dynamite monitor/scheduler: the user can migrate tasks
manually from the PVM console (using the new move command) or from custom programs (using the new pvm_move function call).

The future development of Dynamite aims to provide a complete integrated solution for dynamic load balancing of parallel jobs on networks of workstations or clusters. To realize such an environment, a number of enhancements have to be included, such as support for MPI and generic support for the migration of TCP/IP sockets. The need for MPI support is motivated by the large user base, whereas support for migration of TCP/IP sockets is motivated by the potential large application base. With support for the migration of TCP/IP sockets, a large number of parallel applications, either developed with PVM, MPI, or another parallel programming environment based on TCP/IP communication, can benefit from the virtues of dynamic load balancing by task migration. Besides message-passing parallel programs, support for shared-memory parallel programs is projected. The definition of the OpenMP standard (Dagum and Menon 1998) for shared-memory parallel programs allows for an excellent opportunity to include support for dynamic task migration of shared-memory parallel programs in Dynamite.

Another research line is the integration of Dynamite into the Polder metacomputer. The key issue in the integration is the interplay of the Dynamite scheduler with the local resource manager. The design and implementation of more advanced scheduling strategies will be directed by experimental validation of the strategies in the resource management simulation model, which is developed within the Polder metacomputer framework (Santoso et al. 2000). The DAS distributed supercomputer provides an excellent experimental platform to implement and validate different the designs of components in the Polder metacomputer. The DAS architecture with high speed local-area and wide-area network, grasps the characteristics of the prototypical metacomputer of the near future. In this respect, the various issues in wide-area scheduling and local site scheduling have to be included in the DAS resource management.

Finally, the application of Dynamite as a runtime support system for the APSIS Time Warp simulation environment is a challenging research topic. The dynamic computational requirements of the optimistic Time Warp simulation protocol seems to perfectly match the ability of Dynamite to dynamically rebalance the computational load over the processors. However, there are some different considerations for optimistic Time Warp simulations than for parallel applications in general. First, the notion of progress or useful work in optimistic simulation is not easily translated to CPU percentage. Consider for example a simulation process that inhibits trashing behavior, i.e., spends more time on protocol overhead than on forward simulation. Such a process can have a high CPU percentage, while the amount of progress is almost zero. Thus the monitor/scheduler subsystem must be provided with the internal Time Warp kernel statistics such as event rate, rollback rate, and committed event rate. Another consideration is the simulation class that can benefit from dynamic load balancing on a network of workstations. The potentially advantageous simulation application has irregular internal workloads and is an instance of a
class of applications that are *self-initiated* (Nicol 1991). In this self-initiated application class, the simulation processes typically schedule most of their events to themselves, which leads to relatively few remote messages making this class of applications well suited for networks of workstations, which are known to have high communication overheads.