Technical University of Gdańsk

Faculty of Electronics, Telecommunications and Informatics
Dept. of Computer Systems Architecture

Specialty: Parallel and Distributed Processing

Pawel Czarnul
60983/ETI
email: Pawel.Czarnul@ask.eti.pg.gda.pl
http://www.ask.eti.pg.gda.pl/~pczarnul

MASTER’S THESIS

Professor conferring a MSc degree: Professor Henryk Krawczyk
Consultant: Bartosz Krysztop MSc

Dynamic assignments of applications in distributed environments

Abstract:
This master’s thesis addresses the problems of dynamic process allocation and migration in distributed systems. The author proposed a processing model and implemented the extension to the well known PVM environment – DAMPVM (Dynamic Allocation and Migration Parallel Virtual Machine). This work contains the main concepts of DAMPVM, its construction and implementation details, the description of all its features as well as the experiments performed with DAMPVM and their results. Apart from that a few other available solutions similar to DAMPVM are presented. A few lemmas and tendencies are discussed which were either proved or confirmed by the experiments. DAMPVM uses groups and new PVM 3.4 contexts to solve the problem of process identification in the network.

Gdańsk, 27th August 1999
Copyright ©1999, Paweł Czarnul
# Contents

List of notation ......................................................... i

Preface ...................................................................... iii

1 Introduction ......................................................... 1
   1.1 Concept of the work .............................................. 1
   1.2 Main problems .................................................... 2
   1.3 Basic results ...................................................... 3
       1.3.1 New techniques and methods developed in this work  3
   1.4 Related papers ................................................... 4

2 Task allocation problem .......................................... 9
   2.1 Definitions ........................................................ 9
   2.2 Definition of an assignment ................................. 10
       2.2.1 Decomposition vs assignment .......................... 10
       2.2.2 Static vs dynamic assignment .......................... 10
       2.2.3 Task allocation problem definition .................. 10
   2.3 Task allocation criteria ...................................... 13
   2.4 Speed-up – the goal of good task allocation .......... 15
   2.5 Basic strategies ................................................ 17

3 Allocation functions in other systems ....................... 19
   3.1 UNIX ............................................................ 19
   3.2 PVM ............................................................. 20
   3.3 MPI .............................................................. 21
   3.4 Condor .......................................................... 22
   3.5 CORBA ........................................................ 23
   3.6 MPVM ........................................................... 25
   3.7 MpPVM .......................................................... 26
   3.8 Dynamic PVM .................................................. 27
   3.9 MIST ............................................................. 27
3.10 PVM-related systems comparison ........................................ 28

4 Proposition of a new system: DAMPVM .................................. 29
  4.1 Motivations .......................................................................... 29
  4.2 General description of the system ........................................ 29
  4.3 Model of processing ............................................................ 30
    4.3.1 Computing environment model ....................................... 30
    4.3.2 Application model ........................................................ 33
    4.3.3 Need for instrumentation of application A ....................... 36
    4.3.4 DAMPVM goals ............................................................ 36
  4.4 Requirements and limitations .............................................. 37
    4.4.1 Tools and methods used ............................................... 37
    4.4.2 Implementation requirements ......................................... 38

5 Architecture of DAMPVM ......................................................... 39
  5.1 DAMPVM structure ............................................................. 39
    5.1.1 DAMPVM vs PVM and operating system ......................... 39
    5.1.2 System components ..................................................... 40
  5.2 DAMPVM processing concept .............................................. 41
    5.2.1 Migration and allocation algorithms ............................... 43
    5.2.2 Reassignment initiation ............................................... 44
    5.2.3 Reassignment rules ..................................................... 45
    5.2.4 Migration criteria ....................................................... 46
    5.2.5 Pseudocode ............................................................... 46
    5.2.6 Other variations of the basic algorithm ........................... 48
    5.2.7 Internal DAMPVM messages ......................................... 49
  5.3 Using DAMPVM ................................................................. 52

6 Process identification in DAMPVM ........................................... 53
  6.1 Process identification solution ........................................... 53
    6.1.1 Identification in a migration context .............................. 53
    6.1.2 Process identification mechanisms used in DAMPVM .......... 54
    6.1.3 Migration in a process ................................................ 57
    6.1.4 Advantages and disadvantages of the solution ................ 59
  6.2 How to get rid of message forwarding? ................................ 60
    6.2.1 Limitations ............................................................... 65
  6.3 Visualization of data in DAMPVM ....................................... 66

7 Experiments & analysis .......................................................... 69
  7.1 Migration need ............................................................... 69
    7.1.1 Migration adapts the assignment to other users ............... 70
CONTENTS

7.1.2 Optimal assignment without migration ≠ no idle time . 71
7.2 Description of experiments ........................................... 72
  7.2.1 Analyzed parameters ............................................. 72
  7.2.2 Experiments environment ....................................... 73
  7.2.3 Analyzed application ........................................... 74
  7.2.4 Experiments definitions ....................................... 77
  7.2.5 Experiments results ........................................... 77
  7.2.6 Exemplary assignments ........................................ 80

8 Final remarks ......................................................... 83
  8.1 Experiments discussion ........................................... 83
  8.2 Final conclusions ................................................ 87
  8.3 Tendencies .......................................................... 87
    8.3.1 Future work .................................................. 87
    8.3.2 Possible extensions ......................................... 88
  8.4 Implemented extensions ......................................... 89

A DAMPVM Installation & Programming ................................ 91
  A.1 DAMPVM Installation Manual .................................... 91
    A.1.1 Running DAMPVM ............................................ 92
    A.1.2 Configuration file .......................................... 92
  A.2 DAMPVM Programming Tutorial ................................... 93
  A.3 Problems found ................................................... 96

B List of DAMPVM functions ........................................... 97

C Example source code ................................................. 101

D DAMPVM console screenshot ....................................... 107
List of notation

$A$ – the user application considered in this master’s thesis

$|X|$ – the number of elements in set $X$

$p_{ij}(t)$ – the $j$-th process of application $A$ running on node $N_i$ at moment $t$

$N_i$ – the $i$-th node in the network

$s_{pi}$ – the speed of node $N_i$ defined as the inverse of time needed to perform a selected set of instructions

$L_i(t)$ – the current total load of node $N_i$ (all processes at node $N_i$) at moment $t$, expressed in percentages

$ohL_i(t)$ – the current load of node $N_i$ at moment $t$ except the processes of considered application $A$, expressed in percentages

$ohL_i^k(t)$ – the latest value of $ohL_i(t)$ which is kept on node $N_k$ at moment $t$

$CP(i,t)$ – the set of all application $A$ processes currently running on node $N_i$ at moment $t$

$NP(i,t)$ – the set of new processes of application $A$ to be started at moment $t$ i.e. node $N_i$ gets a message at moment $t$ that it has to start (assign to the nodes) the processes from $NP(i,t)$

$estt_i(t)$ – the estimated execution time of all the processes of application $A$ currently running on node $N_i$ from moment $t$ till they are completed

$estt_i^k(t)$ – the latest value of $estt_i(t)$ which is kept on node $N_k$

$instr_{ij}(t)$ – the estimated number of elementary instructions (or some other metric representing work) to be executed by process $p_{ij}(t)$ on node $N_i$ from moment $t$ till the end of its execution; this can represent the estimated, left work for this process
\( \Gamma(t) \) – the communication costs matrix which defines the costs of sending a unit of information between every pair of the nodes

\( \gamma_{pq}(t) \) – the cost of sending a unit of information via the physical communication link between nodes \( N_p \) and \( N_q \) at moment \( t \)

\( N_b \) – the neighbor relation matrix which defines which node is allowed to spawn or migrate tasks on/to which nodes

\[
\text{nb}_{pq} = \begin{cases} 
1 & \text{if node } N_q \text{ is allowed to spawn new tasks on node } N_p \text{ and/or node } \\
 N_q & \text{is allowed to migrate some}
\end{cases}
\]

\[
\text{processes from node } N_q \text{ to } N_p \\
0 & \text{otherwise}
\]

\( PS_{ij}(t) \) – the size of the process \( p_{ij}(t) \) state at moment \( t \) measured as the size of all the necessary variables and data needed to identify the state of process \( p_{ij}(t) \) on node \( N_i \) at moment \( t \), can be expressed e.g. in KBs

\( CP(i,t)_{\text{active}} \) – the number of all active processes of application \( A \) on node \( N_i \) at moment \( t \)

\( CP(i,t)_{\text{waiting}} \) – the number of all waiting processes of application \( A \) on node \( N_i \) at moment \( t \)
Preface

This master’s thesis has been written at the end of the author’s studies. The author’s specialty is *Parallel and Distributed Processing*. He has created an extension to the well known distributed environment – PVM\(^1\). PVM and other message passing environments and libraries were studied on courses at the technical university. The extension created by the author is called DAMPVM\(^2\) and enables using some extra features not available in the PVM environment. These include dynamic allocation of processes to the machines in the network as well as dynamic and transparent heterogeneous migration both performed dynamically by DAMPVM. This work contains the main concepts of DAMPVM, its implementation details, the description of all its features as well as the experiments performed with DAMPVM and their results. Apart from that a few of other available solutions similar to DAMPVM are presented. A few lemmas and tendencies are discussed which were either proved or confirmed by the performed experiments. Future work including possible improvements and extensions are discussed at the end of this work.

The author has published the main results concerning dynamic assignments and DAMPVM in paper [39]: Paweł Czarnul, Henryk Krzaczek “Dynamic Assignment with Process Migration in Distributed Environments” which has been accepted for the EuroPVM/MPI’99 conference in Barcelona, Spain, which will be presented in the “Lecture Notes in Computer Science” Springer series.

DAMPVM was created mainly to support a flexible environment for time-consuming applications which would adapt dynamically to unpredictable changes of the environment and would make the most of it. One can write any multi-process applications in DAMPVM, can start and get the results. DAMPVM performs such allocation of processes to the nodes in a network that the least possible execution time is achieved. This is done mainly by performing heterogeneous and fast migration. DAMPVM can be downloaded from the author’s Web page: http://www.ask.eti.pg.gda.pl/~pczarnul.

\(^1\)Parallel Virtual Machine
\(^2\)Dynamic Allocation and Migration PVM
Also the latest information including DAMPVM manuals can be found there. DAMPVM uses the latest features of the PVM environment i.e. contexts as well as the group mechanism to achieve process identifiers transparency.

At this point, I would like to thank professor Henryk Krawczyk who is in charge of the Department of Computer Systems Architecture for his encouragement, technical discussions and valuable remarks. I also want to thank my friends from the university, especially Łukasz Garstecki who has shared many courses and created many interesting programs with me and Jerzy Proficz, the local network administrator, who enabled me to perform the experiments.

Now thanks go to my mother for everything and to my father for his valuable advice as well as all the discussions about both humane and technical worlds. I would also like to give thanks to my sisters for supporting energy while writing this master’s thesis. At the end of these acknowledgements special thanks to my girlfriend Honorata for her patience and standing millions of words about computers which must have been very boring and annoying for a different area specialist.
Chapter 1

Introduction

1.1 Concept of the work

In the history of computers the never-ending progress in the performance of processors is still visible and does not seem to face its end soon. Running programs or applications which require fast processors can be speeded up by executing them on faster and faster computers. Apart from the rapid technological changes in the parameters of processors like clock speeds, the architecture (RISC, CISC), cache memory sizes (L1, L2, L3 level cache) as well as memory access time, memory types (EDO, SDRAM, RAMbus, etc.) – [12], [43], [51], one can use other methods to shorten execution time of applications. These are program parallelization and its execution on many processors at the same time. Depending on the kind of a problem, the possibilities of its parallelization (i.e. dividing into processes which can be executed in parallel) and the available hardware the total gain can vary greatly. 

What is more, usually a specific problem requires a dedicated hardware architecture e.g. a parallel system with a shared memory. It means that the execution time of the program solving such a problem can be much shorter when performed on a dedicated architecture rather than a general one.

Using many processors to execute an application can give some gain in a few aspects. In practice, we have the following: execution time (in the face of the system with one processor only) and using some specific resources of some computers in the network by the application. In general, the application which consists of many processes can perform various computations and tasks on data like matrix multiplication or inversion. Particular computers in the network can be dedicated to some tasks i.e. they can do only some tasks or be able to do everything but perform some tasks faster than others. It means that in general finding an algorithm which assigns processes of an application
to processors in the system is a difficult task. It can be complicated by some external factors for the application like failures of some computers or links between them, the change of links' bandwidth or the change of processors' workloads (by other users of the system). All these factors will be formulated and discussed later very carefully.

This work consists of eight chapters. Chapter 1 presents the characteristics of considered problems and discusses some already published solutions and the results concerning the problems or closely related to them. Chapter 2 defines a mathematical model of the problem proposed by the author, distinguishes important parameters and mutual relations among them. It also defines the criteria to be optimized. Chapter 3 presents a few other systems concerning allocation of processes in parallel or distributed environments and/or migration mechanisms. A few important features of these systems are compared. Chapter 4 includes the processing model proposed by the author and used in DAMPVM. This contains the environment model (environment parameters) as well as the application model (which important process parameters are crucial to the performance of DAMPVM). Chapter 5 presents the construction and implementation solutions created by the author and a few code details of DAMPVM. Some variants when applied in different situations and under various conditions are discussed. It explains the real solution versus the theoretical model presented in the previous chapter. Chapter 6 includes the solution to the process identification problem i.e how process identifiers are maintained in the system after migration occurs. The idea how to eliminate a message forwarding necessity is presented. Chapter 7 includes the descriptions of performed experiments and the discussion of the results comparing to the others which can be found in the literature. On the base of these results, a few interesting theorems are formulated and proved by the experiments and mathematical reasoning in chapter 8.

The problem considered in this work focuses on the minimization of the total execution time of an application. It means the time from the start of the application till all the processes of this application finish their execution.

1.2 Main problems

The main problems to be solved in this work include:

- creating the environment which would support:
  - dynamic process allocation to the computers of a local network
  - adapting to the changes of the environment using heterogeneous migration of processes
1.3. BASIC RESULTS

- developing scalable algorithms for process allocation and migration
- defining and executing experiments in the implemented environment to check its performance (mainly scalability)

This includes finding solutions to many problems. They are as follows:

- process identification problem in message passing environments with migration; this includes:
  - maintaining the same identifiers of processes after migration
  - message delivery problem during migration of processes (possible lost or duplicated messages)

1.3 Basic results

This work describes the system created by the author which performs dynamic allocation of processes in a computer network. It consists of the processes which assign processes to machines and a programming library which is analogous to the well known PVM library, while it is extended with functions connected with a dynamic assignment. The system performs a dynamic assignment of processes when they are created and, if it is profitable, migrates processes between pairs of the nodes. The user of the system writes the code of an application which consists of processes and then runs it on the set of computers connected via communication links. The application can be fully dynamic i.e. during its execution, it can create new processes without limitations. The execution of some processes may depend on the others and this does not have to be known before. The algorithm hidden in the system tries to perform such allocation that the total execution time of the whole application is the least possible. Moreover, it can migrate processes from one node to another to minimize this criterion. It is also assumed that the computers captured by the application can be used by other users without any limitations. This is in contrast with specialized, dedicated multiprocessor systems for huge, numeric applications. The proposed system (DAMPVM) detects the presence of other users and the loads of nodes to optimize the assignment.

1.3.1 New techniques and methods developed in this work

The idea of migration developed in the system is quite different from the ones in the currently available systems with migration like Condor, Dynamic
PVM or MPVM\textsuperscript{1}. The proposed migration mechanisms are performed at the code level instead of the system level. Unlike Condor or MPVM special functions for packing and unpacking a process state should be supported by a programmer in the proposed system. Similarly the user program ought to be written in such a way that it can continue from any interrupted point. This means that the packed state should identify the last reached point in the process before migration occurred. These requirements result in the small size of a process state and enable heterogeneous migration. This idea has been implemented as an extension of the PVM environment.

The author uses group and context mechanisms to achieve the transparency of process migration in the network. Processes keep their first identifiers even if they have been migrated, so communication is possible. This is explained in detail in chapter 6. This has been implemented in DAMPVM.

A new idea of decomposing processes into smaller ones is proposed by the author in 6.2. This makes message forwarding redundant and could improve the scalability.

1.4 Related papers

A formalized definition of the problem as well as its solution will be stated in the next chapters. Here the main variants of the assignment problem from literature are presented. This problem can be formulated in the following way: how to allocate processes to the nodes in the network so that some criterion is minimized or maximized?

This section describes theoretical papers referring to the allocation algorithms and methods used in parallel and distributed systems. Real environments are presented in chapter 3.

The first class of assignments was the static assignment class. In this case the set of processes \((P)\) and the set of nodes – processors \((N)\) were considered as the sets known a priori. As far as the application is concerned there is matrix \(W\) which defines the cost of process execution on a certain node in the system. Apart from that there is the communication costs matrix when the cost of communication a portion of data is the same regardless the link one takes. Otherwise two matrices \(C\) and \(F\) are considered which define the amount of information to be sent between each pair of processes and the cost of sending a portion of data between processors respectively. The problem of finding the minimum of the summary execution cost was solved by Stone in 1977 for \(|N| = 2\).

\textsuperscript{1}Migratable Parallel Virtual Machine
1.4. RELATED PAPERS

In the case of more than 2 processors the problem of finding this minimum becomes NP-complete and then one has to use heuristic algorithms as in [28].

The above matrices are often presented as so-called interaction graphs (TIG – Task Interaction Graph). TIG is a digraph \( G = (P, E) \) which is the graph of a program \( P \) – processes, \( E \) – communication between processes) where every node has normalized execution time of the process attached to it. Every edge \((N_i, N_j)\) has two numbers \( t_{ij}, t'_{ij} \) where \( t_{ij}, t'_{ij} \) represent the communication time between processes if they are attached to different nodes or the same node respectively.

In [45] another model is introduced. In classic solutions it is assumed that the amount of data sent between each pair of processes is known before and can be presented as a graph like TIG. Here the allocation of data to processes are optimized. Each process \( P_i \) stores its input and output data on processor \( N_j \) to which it has been assigned. We can consider two situations:

- **data-reuse** – processes \( P_i \) and \( P_j \) assigned to processor \( N_B \) need data from process \( P_k \) on processor \( N_A \). When the processes are executed in order \( k, i, j \), then process \( P_j \) does not need to wait for the completion of the transfer from processor \( N_A \) from process \( P_k \) because the required data is already on processor \( N_B \) (used earlier by process \( P_i \))

- **multiple data-copies** – as above, processes \( P_i \) and \( P_j \) need data from process \( P_k \). All of them are assigned to processors \( N_A, N_B \) and \( N_C \) respectively. Assuming the execution order \( k, i, j \) we can see that process \( P_j \) can get data from two different processors and it can be determined by the communication times to these nodes.

The examples stated above are, of course, ignored by the classic model described before and taking such factors into account can considerably decrease the execution time of the whole application.

When solving problems with static assignments, all the parameters of the system like those defined by the matrices above as well as the other ones (like the interference matrix in [28]) are known before the execution. In problems solved by dynamic assignments these parameters can change in time in the way that cannot be predicted before. For instance, the loads of nodes and thereby their computing power can change depending on the time of a day, the number of users on a certain computer, the sort of tasks executed by other users etc. Similarly the communication links can break down or matrix \( \mathbf{T} \) can change depending on the above factors.

Of course, in this case some assumptions must be stated, which would specify this model more carefully i.e. to assume some known and constant
parameters or assume a probabilistic model where, for instance, there is a
probability one in ten that an event occurs. This can concern communication
events as well as probability distributions of data amounts which are to be
sent.

In paper [33] it is assumed that from time to time some new nodes of
the program are created or some of them must be deleted i.e. we have the
creation of new processes or the termination of their work respectively. A
correction of the previous assignment must be performed then. The loads of
all the nodes should remain approximately the same (they are regarded as the
numbers of processes on the nodes) while the communication costs between
the nodes are minimized. This task is solved using the linear programming
method. Similarly stated problems could also be solved making use of, for
instance, genetic algorithms.

Let us notice that the algorithm which performs the assignment (or a
correction of the previous one) must have low computational complexity in
the face of the number of processes and processors (preferably polynomial).
Sometimes it could not be enough. One can think of an algorithm that
would do an assignment from scratch every time the number of processes
changes. If the application consists of many processes, this solution (in spite
of being optimal considering complexity) could be more costly (taking the
execution time into account) than heuristic algorithms. In the case of a
dynamic assignment procedure its time has to be counted as a part of the
application’s execution time. The same fact can occur with migration. It
can be unprofitable since the overhead when finding processes to migrate as
well as the cost of migration itself which can be too high.

There is a similar system to the one created by the author described in
[18]. It is based on the same environment as the system described in this
master’s thesis, i.e. PVM. Paper [18] does not deal with processes’ migration
(what is often regarded as too costly) but focuses on data redistribution
instead. Data redistribution is forwarding some data to a process from one
node to another to achieve some criterion e.g. equalize the execution times
on all nodes or, in other words, load balancing. As a matter of fact, it is the
same problem as considered in the next chapters in this thesis but the model
proposed by the author takes much more parameters into account as well as
prefers migration to data redistribution.

The algorithm used in this paper can be compared to other neighbor
algorithms which can be found in the literature. Since a heavily-loaded node
tries to balance its load with its neighbors and sends its load to lightly-
loaded neighbors it is in fact a gradient algorithm. The basic version of
such an algorithm was presented in [26]. There were a few classes of nodes
distinguished like heavily-loaded, middle-loaded and lightly-loaded nodes. A
special surface was created where every node counted its distance to lightly-loaded nodes. After that migration was performed. Such a surface could change reasonably so it had to be obtained many times.

On the other hand, so-called neighbor algorithms try to reach a global, good assignment by successive exchanges of the loads between neighbors. A few classes can be distinguished and compared as it was done in [49]. There are diffusion algorithms where a node exchanges its load with many neighbors at the same time as one balancing operation and dimension exchange algorithms where an exchange is performed node by node with the neighbors. The algorithm used here belongs to the second class.

Many strategies have been described in [29] including gradient methods and were compared to the results of experiments. Extended Gradient Model is presented in [23].

The goal of this work was to build a system which would adapt to a non-predictable environment and would make the most of it minimizing the execution time of an application. In the algorithm used in DAMPVM other extra factors not analyzed in other papers are considered. They include the activities of other users who can perform their tasks at the same time. Apart from that there are no distinct classes of nodes. There are estimated execution times for the nodes which represent their loads. One innovative feature implemented is dynamic process state detection (active, waiting) and dynamic prediction of process and node execution times. The experiments were to show when migration is needed and how much can be gained using the mechanisms developed in this work. Of course, also other, more complex algorithms could be used here together with these mechanisms. DAMPVM focuses both on the algorithm and the above mechanisms.

The system can be applied mostly in situations where there is a need for performing time-consuming tasks and the available environment consists of interconnected workstations. These are the examples of such tasks:

- multiplication of huge matrices
- inversion of matrices
- numeric computations in physics, mathematics, chemistry etc.
- image processing i.e. data compression (digital pictures, films, sound)
- data processing in medicine i.e. data in tomography

These tasks can be very time-consuming and proper decomposition into subtasks and using the proposed system give high scalability and thereby significant reduction of the execution time. It is assumed that the same network
is used by other users in the meantime. The destination uses of this system are computational tasks. As it is explained in chapter 6, a considerable overhead of some mechanisms can be reduced if a computational paradigm is assumed. In contrast to Condor or MPVM, DAMPVM tries to minimize the total execution time of DAMPVM tasks started by a certain user. If they run many independent applications the total execution time of all of them will be minimized.
Chapter 2

Task allocation problem

2.1 Definitions

In this section basic definitions are introduced and defined as they will be used throughout this master’s thesis.

Definition 1 A computer system or a system – the network of computers (each equipped with its own processor and local memory) connected with each other with physical links which enable them to send data to each other

Definition 2 A communication link – a physical connection between two computers; it enables them to send data between each other in two directions

Definition 3 A process or a task – a sequential program which can be run on one computer of the system; it can communicate with other processes or create new tasks

Definition 4 Problem decomposition or parallelization – the process of constructing a program which solves a certain problem such that this program consists of many processes which can be executed on different machines at the same time

Definition 5 An assignment of an application to a system – attaching application processes to computers so that every process runs on one computer; this refers to a certain moment of time and thus the assignment of an application to the system can change in time

Definition 6 Migration of a process from node $N_i$ to node $N_j$ ($i \neq j$) – the action of stopping the process running on node $N_i$ and restarting on node $N_j$ so that it carries on its execution with the same data, variables and the environment, if possible
2.2 Definition of an assignment

2.2.1 Decomposition vs assignment

At the beginning, it is very important to distinguish between problem decomposition and the assignment of a program consisting of many processes to this system which is to be solved manually or automatically by some software.

In this work, it is assumed that problem decomposition to processes which can be executed in parallel is done by a programmer who decides about the granularity of a program. The granularity means the sizes of processes versus the size of the whole application. This decision is made in the design phase. The programmer also chooses some other solutions like communication protocols between processes, synchronization solutions, the way processes access some data etc. This work will not take this problem into account assuming that we already have a parallelized application and the system to use and the only problem is to assign the application to this system.

2.2.2 Static vs dynamic assignment

As stated above, the assignment of an application to a system will be considered. Two kinds of assignments can be distinguished:

1. A static assignment – the assignment is known in the compilation phase of the application; it can be done by counting the execution times of processes and the communication times between every pair of them; a precedence tree can be obtained from the analysis of the problem

2. A dynamic assignment – the assignment is performed during the working time of the application; the decision where to place a new process is made just before starting it; much less information about the application and its structure is known in this case

Considering the dynamic assignment we have subclasses of it. The assignment can be constant in time i.e. after starting a process on a computer it runs on this processor till the termination. On the other hand, one can enable migration of a process. In this case the work of a process can be interrupted and continued on a different computer. The solution of the author addresses the second case.

2.2.3 Task allocation problem definition

Let $G_p$ be the graph representing a program (an application) which is to be assigned to a given system. The program consists of a certain number of
processes which are included in set $P$. The processes exchange information between themselves (they send messages). This is represented in the graph by its edges i.e. set $E$. So the following graph represents the application:

$$G_p = (P, E)$$

As an example the graph in Figure 2.1 is shown.

![Diagram](image)

Figure 2.1: An exemplary application graph

Similarly the hardware system can be represented as graph $G_s = (N, L)$ where:

- $N$ – the set of the nodes (processors) in the system
- $L$ – the set of physical links between the nodes (processors)

The following pair of functions is the assignment of the program to the distributed system:

$$\alpha : P \rightarrow N$$
$$\beta : E \rightarrow L,$$
minimizing some definite criteria e.g.:

- the summary cost of processes execution
- the summary cost of communication between the nodes in the system
- the summary cost of communication between the nodes in the system under some conditions on them e.g. keeping approximately the same loads of the nodes

or realizing some confinement put on the system e.g.:

- allowing only some tasks to run on certain processors
- doing routing of messages between some nodes via strictly definite nodes on the way

The last condition can ease testing of distributed applications.

In this work assignment a is emphasized mainly. It means that the problem is to assign a process at its start to a processor as well as re-balancing using migration. Sending messages and therefore the routing of messages is not considered although the basic formula, which is to be optimized, takes this time into account. The programming solution uses the communication functions of PVM and PVM resolves the routing of messages inside itself.

The general definition of an assignment written above requires a few essential parameters which usually determine the concrete assignment:

**an execution costs matrix** – the matrix defining the execution cost of process $P_i$ on processor $N_j$
2.3. **TASK ALLOCATION CRITERIA**

\[
W = \begin{bmatrix}
    w_{11} & w_{12} & \cdots & w_{1|N|} \\
    w_{21} & w_{22} & \cdots & w_{2|N|} \\
    \vdots & \vdots & \ddots & \vdots \\
    w_{|P|1} & \cdots & \cdots & w_{|P||N|}
\end{bmatrix},
\]

where \( w_{ji} \) is the execution cost of process \( P_j \) on processor \( N_i \)

**A communication sizes matrix** – the matrix defining the amount of information which has to be sent between processes

\[
C = \begin{bmatrix}
    c_{11} & c_{12} & \cdots & c_{1|P|} \\
    c_{21} & c_{22} & \cdots & c_{2|P|} \\
    \vdots & \vdots & \ddots & \vdots \\
    c_{|P|1} & \cdots & \cdots & c_{|P||P|}
\end{bmatrix},
\]

where \( c_{ij} \) is the amount of information which has to be sent from node \( N_i \) to \( N_j \)

**A communication costs matrix** – the matrix defining the costs of sending a unit of information between every pair of the nodes:

\[
\Gamma = \begin{bmatrix}
    \gamma_{11} & \gamma_{12} & \cdots & \gamma_{1|N|} \\
    \gamma_{21} & \gamma_{22} & \cdots & \gamma_{2|N|} \\
    \vdots & \vdots & \ddots & \vdots \\
    \gamma_{|N|1} & \cdots & \cdots & \gamma_{|N||N|}
\end{bmatrix},
\]

where \( \gamma_{pq} \) is the cost of sending a unit of information via the communication link from node \( N_p \) to \( N_q \).

One can take notice of the fact, that it may be \( \gamma_{pq} \neq \gamma_{qp} \) for \( p \neq q \).

Obviously, the above formulation of the problem is general and can be applied when using both static and dynamic assignment models.

As the system created by the author is dynamic it looks at the above parameters as well as the ones distinguished later. The model proposed by the author is presented in section 4.3 in detail.

### 2.3 Task allocation criteria

In this thesis the message passing paradigm is analyzed. It means that the analyzed application consists of processes which can communicate between each other, send messages with data to each other. This means they can be synchronized between each other i.e. a process has to wait for other processes, etc. It is assumed that the speed at which a node manages outgoing and coming messages is directly proportional to its processing speed. This ratio is denoted as \( \zeta_i \) for node \( N_i \) and allows for processing and communication
units at the same time. In other words, $\zeta_i$ is the ratio of the communication cost versus the computations cost for machine $N_i$.

As mentioned before, the task allocation problem can be formulated as follows: find such an assignment that a certain criterion is achieved. If the goal is to minimize the total execution time of an application it amounts to minimizing the following quantity:

$$\text{min execution cost} = \max_{i \in \{1, 2, \ldots, |N|\}} \left\{ \sum_{\alpha(P_j) = i} w_{ji} + \zeta_i \sum_{\alpha(P_j) = i, \alpha(P_l) = l, l \neq i} (c_{ji} \gamma_{jl} + c_{lj} \gamma_{li}) \right\}$$

(2.1)

In this formula the execution costs can be regarded as respective execution times (matrix $W$) and the communication times of information units (i.e. bytes or basic data types (int, char etc.)) between nodes (matrix $\Gamma$). In the above formula the latency has been neglected. Migration was not considered in the above formula. It is very difficult to predict how many migrations occur during a program run. Actually it is also difficult to count communication costs when there is migration working because processes can change their locations quite often. One can notice that minimizing the above value leads to the load balancing problem i.e. to allocate processes in such a way that there is the same (ideally) amount of work on every node. Sometimes, however, if migration costs are high, balancing work would result in higher execution time than before performing migration.

A different but similar criterion could be as follows:

$$\text{min communication cost} = \sum_{i=1}^{|N|} \left( \sum_{\alpha(P_j) = i, \alpha(P_l) = l, l \neq i} (c_{ji} \gamma_{jl} + c_{lj} \gamma_{li}) \right)$$

(2.2)

This minimizes the traffic in the network rather than expects the shortest execution time.

The problem is to assign execution processes to the nodes of a system in such a way that value 2.1 or 2.2 is the least possible. The above formulas refer to the static assignment. Since the parameters of this formula can change it is possible that new reallocation is needed. This leads to the dynamic assignment problem. A much more complex model is proposed by the author in chapter 4.
2.4 Speed-up – the goal of good task allocation

Before the parameters used in DAMPVM are distinguished, it is extremely important to understand the goals of processing using many processors instead of one. Two of them were given in section 1.1.

Parallelization of a sequential program can give only definite reduction of the execution time of an application what is specified by speed-up S.

Definition 7 Assuming the execution of application A, speed-up S is defined as:

\[ S_A = \frac{t(1)}{t(N)} \]  

where:
- \( t(1) \) – the execution time of application A running on the parallel machine which is composed of one processor
- \( t(N) \) – the execution time of application A running on the parallel machine which is composed of N processors

Now, let us assume that:
- \( s_i \) – the \( i \)-th part of the program’s instructions to be executed sequentially
- \( r_j \) – the \( j \)-th part of the program’s instructions to be executed in parallel on many processors

This is shown in Figure 2.3. Let us define:

\[ s = \sum_{i=0}^{k} s_i \]

and

\[ r = \sum_{j=0}^{l} r_j \]

Now it can be formulated:

Law 1 The Amdahl law tells:

\[ S = \frac{s + r}{s + \frac{r}{N}} = \frac{1}{s + \frac{r}{N}} \]  

Without losing generality one can assume that the total execution time in the system with only one processor is equal to one unit of time:

\[ s + r = 1 \]
It is visible now that $S$ reaches its maximum value equal to $S = N$, when it is possible to execute the whole program on all the processors i.e. $s = 0$.

Reaching high speed-up values depends to a high degree on the way of program decomposition into the parts which could be executed in parallel and the sort of hardware which can be used.

The above formula is true if there are $N$ identical processors (i.e. having the same computational speeds). Equation 2.4 can be generalized to $N$ processors but of different speeds. This situation is much more common in practice, especially in a heterogenous computer network considered in this work.

Let us assume there are $N$ processors of speeds:

$$s_{pi} \text{ for } i = \{1, 2, ..., N\}$$

where $s_{pi}$ is the speed of processor $N_i$.

In this case the program will take the least time to execute when each available processor is busy without any breaks while executing the program. It means that the working time on each processor will be the same and together with the execution of sequential part $s$ it is the total execution time in the computer system. Let $t_s(N)$ be the execution time of the parallel part on
2.5. BASIC STRATEGIES

each processor. Let \( r^i \) mean the part of instructions which can be executed in parallel and performed by processor \( N_i \). Then:

\[
\frac{r^i}{r} = \frac{s p_i}{\sum_{i=1}^{N} s p_i}
\]  \hspace{1cm} (2.5)

because each processor gets the number of instructions directly proportional to its speed.

Besides, it is visible from this that the execution time of the parallel part \( r^i \) on processor \( N_i \) is constant and is equal to:

\[
t_e (N) = \frac{r^i}{s p_i} = \frac{r}{\sum_{i=1}^{N} s p_i}
\]  \hspace{1cm} (2.6)

Now the speed-up of such a computer system can be estimated in the face of the system with one, the fastest available processor. The exact formula can be written in this way. Also the maximum achievable speed-up is visible after a while.

\[
S = \frac{s + r}{\max s p_i} + \frac{r}{\sum_{i=1}^{N} s p_i}
\]  \hspace{1cm} (2.7)

The maximum speed-up equals:

\[
S_{max} = \frac{\sum_{i=1}^{N} s p_i}{\max s p_i}
\]  \hspace{1cm} (2.8)

In this work a distributed environment is considered. Computers can be heterogeneous and the costs of communication can be considerable. In such a system the processors can devote much time to communication when executing a multi-process application which decreases speed-up \( S \).

The above formulas will be used for calculating the ideal speed-ups for the experiments in chapter 8.

2.5 Basic strategies

From the implementation point of view many different strategies of migration and suitable mechanisms can be distinguished. They include:

- systems implemented at the operating system level e.g.:
  - Sprite ([11])
  - Chorus ([11])
- Mach ([11])

- tools which pack a process state outside the kernel of an operating system; these are architecture-dependent mechanisms so they give only homogeneous migration; the examples of such systems are:
  - Condor ([5])
  - MPVM ([7])
  - Dynamic PVM ([16])
  - MIST ([6])

- tools which pack a process state at the code level; in this case heterogeneous migration becomes possible; DAMPVM is the example of such a system; another example is MpPVM ([9]) in which migration steps are performed on a programming language level so heterogeneity is achieved.
Chapter 3

Allocation functions and strategies in distributed systems

This chapter presents a view on allocation functions and corresponding mechanisms developed in some of the currently available systems and programming platforms.

3.1 UNIX

UNIX ([20], [1], [34], [19]) is the name for the whole family of operating systems like Sun-OS, LINUX, HP-UX and many others. Like the majority of operating systems it allows to start programs simply by typing their names in the terminal or console windows. As UNIX is a multiprocessing system, it can execute many tasks at the same time. In a C code one can use the following functions to start a process:

- `system("process_name")` - `system(name)` executes a shell command given as a parameter by calling `/bin/sh -c name` - one can start task `process_name`

- `exec(*)` family of functions:
  - `int execl( const char *path, const char *arg, ...);`
  - `int execlp( const char *file, const char *arg, ...);`
  - `int execlpe( const char *path, const char *arg, ..., char * const envp[]);`
- `int execv(const char *path, char **argv[])`;
- `int execvp(const char *file, char **argv[])`;

- **pid_t** `fork(void);` - creates a child process

Unfortunately the above calls let start tasks only on the same machine they are invoked. In order to start tasks on a different machine to shorten the execution time or by reason of something else, the calls should be executed by hand on many machines. Of course, it is neither convenient nor flexible if any changes should be introduced. Fortunately one can start a process on a different machine by using the remote shell in the following way:

- **system**("rsh host_name process_name \"") - starts process process\_name on machine host\_name
- `fp = fopen("rsh host_name process_name \","r")`  
  (FILE *fopen(const char *command, const char *type)); - starts process process\_name on machine host\_name and receives message from process\_name using descriptor fp
- **rexec(...)**

Of course, executing time-consuming processes on machines which are already heavily-loaded is not very sensible. The load could be checked using `top` or `ps` commands. This way of starting processes could be useful for batch execution but gives little control over processes after starting them.

### 3.2 PVM

PVM (Parallel Virtual Machine - [35], http://www.epm.ornl.gov/pvm) is the environment in which one can execute applications consisting of many processes. A special library of functions is supported for creating, managing tasks and communicating with other tasks (=processes). This virtual machine consists of some number of hosts one can dynamically attach or detach from the virtual machine. There is a function a process can use to start its children:

- `int numt = pvm_spawn( char *task, char **argv, int flag, char *where, int ntask, int *tids )` in a C code and
- `pvmfspawn(task, flag, where, ntask, tids, numt )` call in a Fortran code
The above function enables a process to start ntask number of processes with the names pointed by task giving them some command-line arguments (argv). The identifiers of the spawned tasks are returned in tids in order to communicate with them later. Tasks can be started on a specified computer with its name pointed by where or on any of the machines of a certain architecture. Finally one can leave the decision where to spawn to PVM which decides where to start a new task.

Unfortunately, as in the previous section, PVM does not trace the loads of the machines which belong to the virtual machine so some solutions like load balancing are not trivial to implement. On the other hand PVM supports easy communication between processes as well as process management.

In other words, PVM enables users to write and execute applications consisting of many processes but does not support dynamic allocation based on the changes of the environment and its current state.

Communication functions include:

- pvm_send(destination_tid, tag) – send
- pvm_bcast(group_name, tag) – broadcast send
- pvm_recv(sender_tid, tag) – blocking receive
- pvm_nrecv(sender_tid, tag) – non-blocking receive

There is also a number of functions for controlling the virtual machine.

### 3.3 MPI

MPI\(^1\) ([32]) is the standard which defines the communication interface between processes of an application. On the contrary to PVM, MPI focuses on the performance of processing and communication rather than on the flexibility of the solution. This is why MPI does not include dynamic process creation. The latest standard MPI-2 is nearer PVM as far the flexibility is concerned and includes dynamic process creation. Functionally MPI is similar to PVM but there are far more communication functions. Just like PVM, MPI enables users to execute applications on many nodes but does not support features or tools to adapt the allocation of processes to the states of the machines in the network. Thus it works best in isolated networks or on multiprocessor computers.

\(^1\)Message Passing Interface
There are many MPI implementations available. One of the best known
are MPICH and LAM MPI. LAM MPI 6.1 includes MPI-2 dynamic process
creation ([32]). Similarly to PVM, there is a special function for creating
processes. In MPI a group of processes is created. Since the group of created
processes is different to the group of the parent, the parent and its child
have to communicate using an intercommunicator. The following function
lets create child processes:

\[
\text{\textbullet\ MPI\_ Spawn(char program[]), char *argv[], int maxprocs, MPI\_Info}
\text{\textbullet\ info, int root, MPI\_Comm parents, MPI\_Comm *children,}
\text{\textbullet\ int errs[])}
\]

The above function lets create maxprocs number of program copies. argv
are the parameters to the spawned tasks and the results are returned in errs.
children is the intercommunicator for communication whose remote group
contains children. Children get the intercommunicator whose remote group
contains the parents. This enables the communication between the parents
and the children. In MPI the above operation is collective. program, argv
and maxprocs are only valid in the process whose rank is given as root.

3.4 Condor

Condor ([5]) is a very successful implementation of the system which tries
to make profit of all available machines in the network. Assuming some
tasks (processes) to execute in the network, Condor can relocate processes
to use lightly-loaded nodes instead of overloaded ones. There is a network
which consists of a certain number of computers supported by various users
for executing the applications of others when the owners are out. The only
requirement is to free the workstation immediately when its owner comes
back to start their jobs. Condor monitors the activities performed on the
workstations and migrates processes if necessary. This is adapting to the
changes of the environment. Three “types” of users are distinguished ([5]):

1. users who come only to check their emails and to reply them
2. users who edit, compile and execute their programs on end
3. users who do many simulations, time-consuming tasks (huge matrix op-
erations, searching, genetic simulations, artificial life simulations, etc.)

Condor tries to allocate the tasks of “type 3” users to the idle machines
of “type 1” and “type 2” users. When a machine is recaptured by its owner
it is relieved immediately by Condor. In this way the owners lose nothing but the others can only gain.

Condor supports single standard UNIX processes. It does not require any changes in the code to relocate them. Condor finds idle machines itself and performs necessary reallocation itself.

Condor works very well but does not support multi-process applications unless processes do not communicate with each other. Unfortunately many computational applications divide their data into a few parts each of which is executed by a separate process. Communication is not maintained by Condor.

Relocating UNIX processes is done by checkpointing them, transferring to another machine and restarting.

3.5 CORBA

CORBA (36), (21), (41) is the specification which enables programmers to integrate various operating systems, programming languages and hardware to work together.

![CORBA architecture](image)

Figure 3.1: CORBA architecture

In CORBA, cooperation between a client and an object implementation

---

2 Common Object Request Broker Architecture
can be distinguished. It is shown in Figure 3.1. The details of communication are hidden to both the client and the object implementation. ORB\(^3\) is responsible for the details of communication. It is important that the codes of the client and the object implementation are isolated from each other. Their interfaces must be expressed using OMG/ISO IDL\(^4\). The client sees only the interface of the object it wants to communicate with. The client can refer to objects located on different machines.

![Diagram of object referencing](image)

**Figure 3.2: Intermachine object referencing**

CORBA supports scalability and migration of objects but this is also limited. The instance of the object is visible to the client as a reference (IOR - Interoperable Object Reference). There can be transient and persistent references. Because object movements are possible, at first the client sends requests to the implementation repository which has a special table of the servers available in its local domain (Figure 3.2). When everything goes all right the client finds out the current address of the server which holds the object, it wants to communicate with and communicates with it. Servers can change the hosts they reside on if a new host belongs to the same local domain because the implementation repository the client knows has the links to the servers. This means that all the objects from a server have to be moved at once with the server. Alternatively the implementation repository can hold also single objects together with servers but it would result in lower

---

\(^3\)Object Request Broker  
\(^4\)Interface Definition Language
3.6. MPVM

MPVM\(^5\) ([7]) is the extension of the PVM environment which supports dynamic process migration. The idea of MPVM is to allocate a regular PVM application dynamically and perform migration, if necessary. In this way PVM processes can use idle cycles of free machines in the local network and relieve overloaded ones. This makes MPVM fully adaptable to the changes of the environment.

MPVM is transparent i.e. it can utilize a regular PVM code. No changes for a program are necessary to be used by MPVM. On the contrary to Condor, MPVM supports interprocess communication. In order to relocate a process, MPVM transfers its state to another machine and restarts the process.

The following mechanisms had to be used to maintain the correct delivery sequence of messages when processes migrate:

1. virtual tids
2. home maps
3. hint maps

Virtual tids enable processes to identify themselves by the same tids during their lives regardless they have been migrated or not. A home map is stored on every machine and stores tid-to-host tuples for all the processes which have been started on this node. Of course, these processes could have been migrated in the meantime to some other nodes.

Figure 3.3 shows a possible scenario in the system. When task T1 migrates from host H1 to H2 and then H3 it needs to inform the host it was born on i.e. host H1. If task T2 wants to send a message to task T1 it sends a request to host H1 and finds out the current location of T1. Now it is able to send the message. A hint map is a cache which stores tid-to-host tuples received during previous requests to home maps. This reduces intermachine communication. A task can often get to know the location of another task from its own machine.

\(^5\)Migratable PVM
3.7 MpPVM

MpPVM\textsuperscript{6} ([9]) is a design and a software system which supports process migration in the PVM environment. The idea of using idle workstations in a non-dedicated (often heterogeneous) network is presented. In fact, it is a modified version of PVM where process migration is performed on a high level by executing programming language statements when migration occurs. It is in contrast to e.g. MPVM where stack, data segments need to be transferred. MpPVM consists of three parts: MCL (the MpPVM precompiler), Mpd (the MpPVM daemon - a modified version of pvmd) and Mlibpvm (a modified version of pvmlib). The above migration mechanism guarantees migration heterogeneity. In a MpPVM process, migration can be done only in certain points of the code. The MCL precompiler analyzes the code and inserts migration points at which migration can be performed.

\textsuperscript{6}Migration-point based PVM
MCL analyzes instructions sequences in a program to insert migration points in the appropriate points of the code and obtains the minimum set of data that needs to be transferred at every migration point (explained in [8]). Mpts store MPTs (Migration Process Table) which has the current locations of the processes in the virtual machine. This is to guarantee reliable point-to-point communication between processes which can migrate in the meantime. As in MPVM, the message forwarding mechanism is developed to make sure that all messages reach their destination processes even if they have been migrated in the meantime. There is also a scheduler (schedulers - depends on the implementation) which looks for idle computers in the network to which tasks could be migrated. It also initializes migration processes (the one which is to be migrated and its successor) before it is performed. The scheduler must inform all Mpts about processes which has communicated or will do it with migrated processes. In [9] and [8] it is assumed that there can be two version of a scheduler instance: a global one or distributed scheduler processes. Experimental results and the comparison to PVM are also given in [9].

3.8 Dynamic PVM

Dynamic PVM (DPVM – [16]) combines two well known environments: PVM and Condor with its checkpointing and migration facilities. The idea was to add process migration to the PVM environment. A global scheduler starts the migration procedure. Similarly to Condor, a checkpoint is created (based on a task core-dump), it is transferred to a new machine and restarted. Since in DPVM checkpointing is not allowed during communication, critical sections are introduced which cover the “communication parts” of a process code. When a new task is created on a remote machine, its PVM daemon is initialized for the new process. After restarting the process, communication state is initiated as in the old copy and the daemon delivers all the messages that has come in the meantime.

3.9 MIST

MIST\(^7\) ([6]) is the project which focuses on resource management, a multi-user support as well as profiling and debugging the PVM environment. Task checkpointing/migration facility is proposed as MMPVM (Multi-user, Migratable PVM), a MIST kernel, which is the extended version of PVM. MM-

\(^7\)Migration and Integrated Scheduling Tools
PVM proposes message forwarding and sequencing to save the correct order of PVM messages (as promised by PVM). Since checkpointing is done at the same level as in Condor or MpPVM, it results in homogeneous migration only. There is also the Global Scheduler implemented as a resource manager in PVM which captures `pvm_spawn()` calls from processes and implements its own dynamic allocation algorithm. The Load Monitor traces the load of a machine it resides on – there is one LM copy on each machine.

### 3.10 PVM-related systems comparison

Table 3.1 shows a general PVM-related systems comparison concerning migration and dynamic allocation features.

<table>
<thead>
<tr>
<th>Environment</th>
<th>Dynamic allocation</th>
<th>Migration</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>PVM</td>
<td>+</td>
<td>-</td>
<td>a very flexible environment, a virtual machine can be extended dynamically</td>
<td>dynamic allocation does not adapt to the environment</td>
</tr>
<tr>
<td>Condor</td>
<td>+</td>
<td>+</td>
<td>no special changes in a code required</td>
<td>only single processes supported</td>
</tr>
<tr>
<td>MPVM</td>
<td>+</td>
<td>+</td>
<td>asynchronous migration, no changes in a PVM code required</td>
<td>homogeneous migration only, possible big size of a process state</td>
</tr>
<tr>
<td>MpPVM</td>
<td>+</td>
<td>+</td>
<td>heterogeneous migration, necessary checkpoint data analysis</td>
<td>migration points only, not fully synchronous migration</td>
</tr>
<tr>
<td>DPVM</td>
<td>+</td>
<td>+</td>
<td>asynchronous migration</td>
<td>homogeneous migration only, possible big checkpoint size</td>
</tr>
<tr>
<td>MIST</td>
<td>+</td>
<td>+</td>
<td>a general approach, asynchronous migration</td>
<td>homogeneous migration only, possible big checkpoint size</td>
</tr>
</tbody>
</table>

Table 3.1: PVM-related systems comparison
Chapter 4

Proposition of a new system: DAMPVM

4.1 Motivations

PVM is a well known platform with a message passing paradigm which enables programmers to write distributed applications quite easily. Unfortunately PVM does not detect the changes of the environment to allocate the tasks perfectly. The execution of time-consuming computational tasks can be shortened considerably by reallocating tasks dynamically. The reason for this are other users who come and leave the local network and do their own tasks which load the network. Migration can balance the work which results in the perfect (theoretically) assignment. Since PVM is well known and widely used, the idea was to extend it with dynamic allocation and heterogeneous, fast process migration. The implemented system should enable testing the performance of various allocation and migration algorithms.

4.2 General description of the system

The author of this work has created an environment which enables users to execute distributed applications like in the PVM environment. An application consists of many processes (1 or more) and is dynamically assigned to the set of computers. The allocation and migration algorithms, their features and capabilities will be shown in the next chapters.

The environment is based, as PVM, on the idea of a virtual machine that is a big virtual computer which consists of many nodes each of which having its own processor (an execution unit) and a memory. As far as a program model is concerned it is an application which consists of many processes which
can communicate between each other by sending/receiving messages to/from specific processes. The environment is dynamic itself because processes can create new children during their execution.

### 4.3 Model of processing

To describe the principles of the proposed dynamic assignment algorithm two models are considered.

#### 4.3.1 Computing environment model

PVM model represents a virtual machine which consists of many processors interconnected via physical links where each processor has its own memory. From an application’s point of view this machine is seen as many nodes (computers) on which the processes of this application can be executed. In practice, it is often a local network where processes of an application can run on a subset of nodes in parallel. A certain speed-up can be achieved in comparison to sequential processing of these processes.

Unfortunately, this model, while being very general, does not allow for other users executing their processes in the same network at the same time.

![Figure 4.1: Network computing environment](image)

A computing environment considered in this paper is shown in Figure...
4.3. MODEL OF PROCESSING

4.1. It can be described by graph $G_s(N, Nb)$ where $N$ is the set of nodes ($N = \{N_1, N_2, \ldots, N_n\}$) and $Nb$ describes the neighbor relation between the nodes which is defined below.

**Neighbor relations**

A pair of nodes can be neighbors which can be defined by the user of the system. This was introduced to decrease interprocessor communication. Actually, if node $N_i$ is a neighbor of node $N_j$ it means that node $N_j$ can place a new process on node $N_i$ (if it is profitable) and node $N_j$ can migrate the processes running on it to node $N_i$.

A node can migrate tasks only from itself to its neighbors and assign new tasks only to itself or its neighbors. We introduce a matrix to denote this:

$$
Nb = \begin{bmatrix}
0 & nb_{12} & \ldots & nb_{1n} \\
\vdots & \vdots & \ddots & \vdots \\
 nb_{n1} & \ldots & \ldots & 0
\end{bmatrix},
$$

where:

$$

b_{pq} = \begin{cases}
1 & \text{if node $N_q$ is allowed to spawn} \\
& \text{new tasks on node $N_p$ and/or node} \\
& \text{$N_q$ is allowed to migrate} \\
0 & \text{otherwise}
\end{cases}

$$

This matrix does not have to be symmetrical. It can be defined by the user of DAMPVM (using a configuration file). It means that the topology of spawning and migration is fully flexible but fixed during the execution of an application.

**Node characteristics**

Let us consider the execution of application $A$. Then two classes of processes running on each system node are distinguished:

1. Application $A$ processes – referring only to application $A$ which execution times are minimized

2. Processes referring to both other users processes and system processes
This is illustrated in Figure 4.2. The percentages of a processor captured by other users processes and the system are monitored to determine the fraction of the processor available for application \(A\).

Each node is characterized by the following parameters (variable in time):

- \(s_{pi}\) - the speed of node \(N_i\) defined as the inverse of time needed to perform a selected set of instructions – these set should be similar to the instructions used in application \(A\) because e.g. node \(N_i\) can be faster than \(N_j\) in numerical operations but slower in accessing memory

- \(L_i(t)\) – the current total load of node \(N_i\) (=processor at node \(N_i\)) at moment \(t\) expressed in percentages

- \(othL_i(t)\) – the current load of node \(N_i\) by processes different from the processes of application \(A\) at moment \(t\) expressed in percentages; this variable is measured using the UNIX ps command; since it is always calculated on node \(N_i\), the neighbors of \(N_i\) are notified about its changes and keep their own copies denoted as \(othL_k(t)\) for node \(N_k\)

- \(CP(i, t)\) – the set of all application \(A\) processes currently running on node \(N_i\) at moment \(t\)

- \(NP(i, t)\) – the set of new processes of application \(A\) to be started at moment \(t\) i.e. node \(N_i\) gets a message at moment \(t\) that it has to assign the processes from \(NP(i, t)\) to some nodes

- \(estt_i(t)\) – the estimated execution time of the processes of application \(A\) currently running on node \(N_i\) from moment \(t\) till they are completed

**Communication costs**

Now the parameters introduced in subsection 2.2.3 will be extended as they are used in practice by DAMPVM.
4.3. MODEL OF PROCESSING

Let matrix $\Gamma(t)$ be the communication costs matrix which defines the costs of sending a unit of information between every pair of the nodes:

$$
\Gamma(t) = \begin{bmatrix}
\gamma_{11}(t) & \gamma_{12}(t) & \cdots & \gamma_{1n}(t) \\
\gamma_{21}(t) & \gamma_{22}(t) & \cdots & \vdots \\
\vdots & \vdots & \ddots & \vdots \\
\gamma_{n1}(t) & \cdots & \cdots & \gamma_{nn}(t)
\end{bmatrix},
$$

where $\gamma_{pq}(t)$ is the cost of sending a unit of information via the physical communication link from node $N_p$ to $N_q$ at moment $t$.

Let $e_{ij}(t)$ be the amount of information which is to be sent from process $p_{ij}(t)$ to $p_{ij}(t)$ from moment $t$ till the end of the process $p_{ij}(t)$ execution.

4.3.2 Application model

Apart from the system model, the model of an application must be assumed as well.

Static description

A static view on the application is represented by graph $G_A = (P, E)$ where $P$ is the set of the processes of application $A$ and $E$ is the communication between the pairs of processes.

Dynamic creation tree

Moreover, application $A$ consists of many processes which can be created any time without any limitations. New processes are created by the processes which have already been started before. Actually this forms a dynamic tree in which the root process is the first one and it creates its children, they do the same and so on. An exemplary tree is shown in Figure 4.3.

Migration

Processes of application $A$ can be migrated from node to node. Such decisions can be taken by DAMPVM automatically. When a process is to be migrated, its execution is interrupted, its state is packed, a new copy on a different machine is started and this copy receives the state of its predecessor and carries on the execution. There is one parameter connected with migration: $PS_{ij}(t)$ – the size of the process $p_{ij}(t)$ state at moment $t$ – the size can be expressed in kilobytes (the state includes all the necessary variables and
data which are needed to carry on the execution from the interrupted point). Sometimes a state can be composed of only a few important variables. This is in contrast with other systems e.g. Condor or MPVM where memory, stack segments and many others need to be taken and packed.

**Forward execution time prediction and estimation**

DAMPVM is able to detect the current number of instructions/the amount of work a process has to do.

This is represented by parameter $\text{instr}_{ij}(t)$ – the estimated number of elementary instructions to be executed by process $p_{ij}(t)$ on node $N_i$ from moment $t$ till the end of its execution. This parameter can represent the number of instructions, the number of loop runs in the process or the value of some metric representing the work which remained for the process.

The same as with node parameters $\text{instr}$ and $PS$ can change in time.

The sense of parameters $\text{instr}_{ij}(t)$ and $\text{estt}_i(t)$ is shown in Figure 4.4. They refer to estimated values of work required for process $p_{ij}(t)$ on node $N_i$ and the execution time of the application $A$ processes for node $N_i$ (from moment $t$) respectively. Let us consider the situation illustrated in Figure 4.4. There are three processors and some processes which belong to application $A$ as well as some other processes (which belong to the system or other users). There were some actions performed in the past (moment $t_x$ is considered) including migration. But no past events are considered at moment $t_x$. Every process of application $A$ has its own predicted number of instructions – $\text{instr}_{ij}(t_x)$. Then the estimated execution time $\text{estt}_i(t_x)$ for node $N_i$ can be calculated as the function of the $\text{instr}_{ij}(t_x)$ values:
4.3. MODEL OF PROCESSING

\[ \forall i \in \{1, 2, \ldots, |N|\} \quad estt_i(t_x) = \frac{1}{sp_i(1 - \frac{\theta h L_i(t_x)}{100})} \sum_{j=1}^{CP(i,t_x)} instr_{ij}(t_x) \quad (4.1) \]

Figure 4.4: Definition of the \textit{instr} and \textit{estt} values

It is important for the performance to detect the current state of a process. It is possible to distinguish two different states of a process as follows:

- **active** - performing computations
- **waiting** - waiting for the results of computations from other processes or for access to some resources (controlled by monitors or other resource sharing control units)

\[ CP(i,t) = CP(i,t)_{active} + CP(i,t)_{waiting}, \]

where \( CP(i,t)_{active} \) means the number of active processes on node \( N_i \) at moment \( t \) and \( CP(i,t)_{waiting} \) the number of waiting processes. When process \( p_{ij}(t) \) enters its waiting state it can inform DAMPVM about it by calling special library functions. When DAMPVM receives this message it sets temporarily for this process \( instr_{ij}(t) = 0 \). It can enable other processes from other nodes to migrate to the temporarily relieved node \( N_i \).

The reasons for introducing parameter \textit{instr} are the following:
• it tells in what state a process is – it solves the problems when in spite of perfect work balance idle time appears – discussed in subsection 7.1.2

• the execution path of a process may not be known a priori – it can depend on e.g. the results of different processes computations

4.3.3 Need for instrumentation of application $A$

The above problems can be easily solved by putting special instructions in appropriate points of a code to tell DAMPVM how much work is left as far as the process execution is concerned. A programmer who writes the code of a process can estimate this and put a special function $\text{PC Instructions}()$ (see appendix B for the description of the function and appendix C for the real example) in the code. If a process wants to inform that it is in an idle state it sets $in\text{str} = 0$. In DAMPVM this can be done by calling function $\text{PC Wait}()$ (see appendices B and C). This is extremely useful in computational processes before receive functions which are usually blocking and can take much time (like in the example in appendix C). Similarly setting $in\text{str}$ to appropriate values after conditional instructions informs that the estimated, left amount of work for a process has changed. The exemplary instrumentation based on this reasoning is shown in Figure 4.5. One can notice that parameter $in\text{str}$ does not have to represent real instructions (e.g. multiplication) as its name says but also a predicted amount of work expressed in numbers. For DAMPVM, the relative proportions of these numbers for processes are important, not their absolute values. A programmer can guess these values and support them. If not, DAMPVM treats all processes as of the same size. The experiments show that especially detecting a process state in the way described above gives a considerable gain. The concrete instrumentation for the application used in experiments (appendix C) is shown in Figure 7.4.

4.3.4 DAMPVM goals

Formula 2.1 for moment $t$ can be written (using the additional introduced parameters) as follows:

$$min \ time = \ max_{i \in \{1,2,...,|N|\}} \ \{ \text{est} \times i(t) \} =$$

$$\max_{i \in \{1,2,...,|N|\}} \ \left\{ \frac{1}{sp_i (1 - \frac{oh_i(t)}{100})} \ \sum_{j=1}^{[CP(i,t)]} \ \text{instr}_{ij}(t) \right\} +$$
4.4. REQUIREMENTS AND LIMITATIONS

\[ \zeta_i \sum_{i \neq i} (c_{ij}(t) \gamma_{ii}(t) + c_{fij}(t) \gamma_{ii}(t)) \] \hspace{1cm} (4.2)

Since \( s_{pi} \) is counted for a selected set of instructions, \( \zeta_i \) should be selected so that proper relations between processing data and communication for processor \( N_i \) are met. If any of the additional parameters changes substantially then the value of the above formula changes and the assignment must be updated by the algorithm. As one can see the above formula represents the total execution time of an application and should be minimized. Sometimes it is worth to move a task from one node to another to reduce the value of \( \text{min time} \). One has to consider the cost of migration then.

4.4 Requirements and limitations

4.4.1 Tools and methods used

The user of DAMPVM is supported with a library of functions analogous to PVM ones. It is described in detail in chapter 5. Particular processes are identified by identifiers PCTids (analogous to tids\(^1\) in PVM). Similar functions to operate on buffers and for sending and receiving messages are supported.

The most important difference is that processes can migrate from one node to

\( ^1 \)tid task identifier
another during their work and these operations are transparent to the user. It means that after migration they are still identified by the same identifier as before the migration and all the other processes can use the same identifier for receiving and sending (here there is some confinement - it is explained in section section: How to get rid of message forwarding?). A programmer, if wants to enable migration for a process, must support the following functions in its code:

- **PackState()** – the function which packs the current state of a process
- **UnPackState()** – the function which unpacks the previously packed state

A process can carry on its work from the interrupted point. Of course, the code of the process as well as the functions PackState() and UnPackState() has to be written optimally to enable the process to continue its computations from the interrupted point.

This solution has both advantages and disadvantages and they will be discussed in detail in the next chapters.

A big advantage of this solution is a great conceptual analogy to the well known PVM environment. It enables DAMPVM users to import old PVM programs to the new environment with a new library very easily. Various dynamic assignment algorithms can be put into DAMPVM to test their qualities and usefulness in different conditions. The library also contains mechanisms which enable to make profit from additional information about processes e.g. their migration sizes. In the future other information may be used e.g. the set of processes a certain process communicates with and another set of processes it would never communicate with.

### 4.4.2 Implementation requirements

These are the necessary technical requirements needed for successful compilation and installation of DAMPVM:

1. Operating system in which PVM version 3.4+ works properly
2. PVM 3.4+ successfully installed
   (available from [http://www.ornl.gov/pvm](http://www.ornl.gov/pvm))
3. V GUI version 1.16+ working in this system (the provided interface of DAMPVM uses this library) – as an option the system will work in a *quiet* mode without an interface (this library is not needed then), V GUI is available from [http://objectcentral.com](http://objectcentral.com)
Chapter 5

Architecture & implementation of DAMPVM

5.1 DAMPVM structure

5.1.1 DAMPVM vs PVM and operating system

The implementation of DAMPVM, the whole idea, in particular the solution to the problem of process identification after migration occurs (described in chapter 6) are the original author’s solutions.

DAMPVM consists of kernel processes which run on all the nodes where application’s processes are to be executed. These kernel processes are named PCSchedulers. These processes, during the system’s work, communicate between each other in order to exchange information useful for good allocation of newly created processes or migration of previously running ones. These are the exemplary sorts of information exchanged:

- the change of the current load of a processor by other users processes
- the change of the estimated processing time of processes on a certain node in the network which can be caused by:
  - the termination of a process
  - a new process on a node
  - the migration of a certain process from a node to another one
- messages which measure the current link bandwidth between two nodes

Mutual positions and relations among DAMPVM, an application, PVM and the operating system are shown in Figure 5.1.

39
No matter what kind of a network is used (LAN, WAN, etc.) and its structure (token bus, token ring or others) a DAMPVM user can define logical links between particular nodes. In order to decrease the overhead of DAMPVM, nodes communicate only with their logical neighbors. Similarly, the migration of a process can be done only from a certain node to one of its logical neighbors. Defining the logical structure of the network (matrix $N_b$) i.e. logical links is possible through a special configuration file in which the user can enter any structure they want. Defining a specific structure can even help in finding bugs in an application containing many processes.

One can notice that in particular the logical network can form a complete graph or structures like: a bus, a ring or others. Apart from that the user can define an incoherent graph as in Figure 5.2.

### 5.1.2 System components

As mentioned above, the processes of an application have mechanisms for communication and process creation analogous to PVM processes. A kernel process on every node possesses full and up-to-date information about the processes currently running on this node. When a process wants to create a new process its request is directed to the kernel process running on the same machine. This kernel process makes a decision (using information owned by itself and from its logical neighbors) where to place a new task (or tasks) and starts it (them).

The structure of the system is shown in Figure 5.3. There are kernel processes – **PCSchedulers** (one on every machine) which exchanges messages between each other, application processes on the nodes which communicate
5.2. DAMPVM PROCESSING CONCEPT

between each other (communication implemented by a programmer). There is also communication between application processes and DAMPVM kernels which is hidden in the DAMPVM functions. The most important messages of this kind as well as between kernel processes are listed in subsection 5.2.7. Now the system elements can be put into Figure 5.1 which is shown in Figure 5.4.

5.2 DAMPVM processing concept

DAMPVM tries to balance the estimated processing times of the nodes known at the moment \( estti(t) \). This results in minimizing the total execution time of a considered application. If external conditions change or any of the \( instr_{ij} \) values changes or new tasks appear (new set \( NP(i, t) \)) DAMPVM tries to balance the work to achieve a good assignment. Similarly to [26], processes are transferred from more loaded (a higher \( estt \) value) nodes to their lightly-loaded (a lower \( estt \) value) neighbors. If the current state of the application (indicated by the \( instr \) values – result in \( estt \)) is constant, then no improvements are performed unless the environment changes (the \( ohL_i \) values).
process creation request
application process
DAMPVM kernel

communication between DAMPVM kernel processes
communication between DAMPVM application and kernel processes
communication between DAMPVM application processes

Figure 5.3: DAMPVM structure
5.2. DAMPVM PROCESSING CONCEPT

5.2.1 Migration and allocation algorithms

The system manages two algorithms at the same time:

- a migration algorithm – where to move a task or tasks and to which machines they should be transferred
- an allocation algorithm – where (on which machine) to start a new task (tasks)

These two algorithms are managed by the kernel processes each of which resides on a separate computer. They communicate between each other (only with logical neighbors) and make all the decisions for these algorithms. The goal of these algorithms is to minimize the value given by formula 4.2. This value refers to a specific moment of time so decisions are made taking the current settings and parameters into account. If something changes, this has to be updated.

Every kernel process deals with the problem where to place a new process which is started by a certain process running on the same node as the kernel process. In other words, handles all the requests from the processes on this node. It can place a new process either on any of neighboring nodes or on itself. This is the allocation algorithm part. The same kernel process decides whether to migrate any of the processes running on its node to the
logical neighbors computers. It is important that it can make decisions only for the processes from its node. This means that kernel processes work independently (they do not lock themselves) but they use information from their logical neighbors as well.

The diagram shown in Figure 5.5 presents the processing concept of the DAMPVM algorithm. The current assignment is the state of the whole system.

![Algorithm concept diagram](image)

**Figure 5.5: Algorithm concept**

5.2.2 Reassignment initiation

DAMPVM is event-driven. It means that two classes of events can cause the change of the current assignment:

1. new *spawn* requests – new tasks are started on nodes using the *Spawn* function – the pseudocode shown in Figure 5.6

2. changes of *instr<sub>ij</sub>(t)* or *othL<sub>i</sub>(t)* parameters – function *Schedule* checks if there is a migration need and, if so, performs migration – the pseudocode shown in Figure 5.7

These two event classes are shown in Figure 5.5.

Each node (the kernel process) accepts *spawn* messages from the processes which have already been started before on it. After receiving a *spawn* request,
it can start new processes either on itself or its neighbors (indicated by matrix \( N_b \)). The stream of tasks (a probabilistic model) is not known a priori so these messages can come anytime.

5.2.3 Reassignment rules

The migration algorithm tries to move some tasks towards a lower \( estt \) value i.e. to the neighbors which have lower \( estt \) values. There are no distinct classes of nodes like: \{lighty-loaded, middle-loaded or heavily-loaded\} like in the classical gradient algorithm ([26]). Here the load is represented by \( estt \) and \( estt \) is calculated using formula 4.1.

If there are new tasks to start i.e. new set \( NP(i, t) \), the assignment of these tasks is performed which results in new set(s) \( CP(\ast, t) \). One can notice that for a new set \( NP(i, t) \) only \( CP \) sets for nodes \( N_i \) and its neighbors can change. In other words the algorithm which performs a new assignment is coded as function \( \text{Spawn} \) (Figure 6.1) such that:

\[
\forall i \in \{1, 2, \ldots, |N|\} \\
\quad CP(i, t) = \text{Spawn}(CP(k, t), NP(k, t), othL_k(t), s_{pk} : nb_{pk} = 1 \lor k = i) \\
\quad CP(i, t) = \text{Schedule}(CP(k, t), othL_k(t), PS_{kj}(t), \Gamma(t), s_{pk} : (nb_{pk} = 1 \lor k = i) \land j \in \{1, 2, \ldots, |CP(k, t)|\})
\] (5.1)

Function \( \text{Spawn}() \) starts new tasks one by one on the nodes which have the smallest \( estt \) values.

If \( instr_{ij}(t) \) changes, it is possible that a new assignment is required.

\[
\forall i \in \{1, 2, \ldots, |N|\} \\
\quad CP(i, t) = \text{Schedule}(CP(k, t), othL_k(t), PS_{kj}(t), \Gamma(t), s_{pk} : (nb_{pk} = 1 \lor k = i) \land j \in \{1, 2, \ldots, |CP(k, t)|\})
\] (5.2)

Function \( \text{Schedule}() \) checks for each neighbor \( N_k \) for each task \( p_{ij}(t) \) if migration to \( N_k \) is profitable. If succeeded, migrates the selected task to \( N_k \).
5.2.4 Migration criteria

The migration criterion used in the current version of DAMPVM is quite simple but it works well. It can be compared to pressure and wind in the nature. It means that the load flows from node to node from higher to lower estt values. A node checks if the current maximum of estimated times on itself and its candidate neighbor is bigger than the maximum of these times after the migration of a candidate task to this node. This is clearly visible in Figure 5.7 as a pseudocode and is shown in Figure 5.8. If it is so, the node migrates this task to its neighbor. Other variations of this algorithm and criteria are suggested in subsection 5.2.6.

5.2.5 Pseudocode

In this section general ideas of the algorithms used in DAMPVM are presented as pseudocodes.

Since estt\(_i\)(t) is always calculated on node \(N_i\) the neighbors of \(N_i\) are informed about its changes. They keep their own copies denoted for node \(N_k\) as estt\(_k\)(t). This is also the case with othL\(_i\)(t).

The changes of estt\(_i\)(t) and othL\(_i\)(t) are propagated to the neighbors of node \(N_i\) so that neighbor \(N_k\) updates its estt\(_k\)(t) = estt\(_i\)(t) and othL\(_k\)(t) = othL\(_i\)(t).

The changes of parameter othL\(_i\)(t) are not known a priori so the migration algorithm should reassign processes after considerable changes of this and it does so.

The proposed algorithms for node \(N_i\) can be written as pseudocodes (Figures 5.6, 5.7 and 5.10).
5.2. DAMPVM PROCESSING CONCEPT

\[
\text{Schedule()} \{
\text{for (every logical neighbor } N_k) \\
\text{for (every task } p_{ij}(t) \text{) } \{
\text{estti}(t_m) = estti(t) - \frac{1}{s_{pi}(1-\frac{1}{m ij(t)})} \times \text{instri}(t) + \zeta_i \gamma_{ik} \text{PS}_{ij}(t); \\
\text{esttl}_k(t_m) = esttl_k(t) + \frac{1}{s_{pk}(1-\frac{1}{m jl(t)})} \times \text{instrl}_j(t) + \zeta_j \gamma_{jk} \text{PS}_{lj}(t); \\
\text{min}(t) = \min(\text{estti}(t), \text{esttl}_k(t)); \\
\text{min}(t_m) = \min(\text{estti}(t_m), \text{esttl}_k(t_m)); \\
\text{if (min}(t_m) < \text{min}(t)) \{ \\
\text{migrate task } p_{ij}(t) \text{ to node } N_k; \\
\text{esttl}_k(t) = \text{esttl}_k(t_m); \\
\text{break}; \\
\} \\
\}
\}
\]

Figure 5.7: Migration – function Schedule()

The real algorithm implemented in DAMPVM does not take communication between processes into account (\(\forall i, j, l, f, t \ c_{ijf}(t) = 0\)). Communication time between nodes is used only for estimating migration costs. The ratio \(\zeta\) is the same for every node and it equals 0.035. It was optimized experimentally. It is possible that another value would be better for a different system or application. By manipulating this parameter one can also increase or decrease the migration occurrence (see Figure 5.7). This is also connected with the speeds of the nodes which are measured automatically by DAMPVM. DAMPVM executes a selected set of instructions to obtain these speeds. If this set was changed, this ratio would have to be changed as well to save the appropriate proportions between the processing and communication speeds.

The above algorithms balance the remaining work of an application rather than the loads of nodes like many other algorithms. As one can see, reaching the minimum value of equation 4.2 is possible if and only if all the nodes work non-stop without any breaks and they finish their parts of work (parts of an application) at the same time. In this optimal case communication times are very important because they determine how close to optimal the scalability is. The same criteria are used in [18]. It is visible that the allocation algorithm starts a new task on the node the request has arisen or one of its logical neighbors having the smallest value of \(\text{estti}_k(t)\).
5.2.6 Other variations of the basic algorithm

You can see that there are two versions of function Schedule() - the one shown in Figure 5.7 is used in DAMPVM. To prevent many nodes from overloading their common neighbor like in Figure 5.11, a locking version was also tested. The idea was that a node locks another node to which it wants to migrate a task and unlocks it after the whole operation. This was to prevent other nodes from migrating other tasks to the same node at the same time. Please note that this algorithm causes that some nodes need to resign from migrating a task to the node which is locked – they search for another candidate. Finally, the experiments showed that for the environment that was available (up to 8-9 machines) this “improvement” gave worse results than the “standard” algorithm. Sometimes it forced migrating processes to the best available but unlocked nodes. It is possible that other more complex variations of this idea would be useful – it is a good research area. The author proposes the following extended migration criterion. Let \( H(N_i)(t) \) be the number of the \( N_i \) neighbors with \( estt(t) \) (estimated execution time from moment \( t \)) higher than \( estt_i(t) \). Now node \( N_i \) can migrate a task to node \( N_k \) on which \( H(N_i)(t)estt_k(t) \) is the least if, of course, the maximum of estimated execution times of nodes \( N_i \) and \( N_k \) is lower after migration than before. In this case the idea is to migrate a task to the node with a reasonably low \( estt \) value which has not so many neighbors which could also migrate their processes to it.

Moreover, another version of a “standard” and locking function Schedule()
5.2. DAMPVM PROCESSING CONCEPT

Schedule() {
    for (every logical neighbor \( N_k \))
        for (every task \( p_{ij}(t) \)) {
            \[ estt_i(t_m) = estt_i(t) - \frac{1}{\alpha_k(t)} instr_i(t) + \zeta_i \gamma_k P S_i(t); \]
            \[ estt_k(t_m) = estt_k(t) + \frac{1}{\alpha_k(t)} instr_i(t) + \zeta_k \gamma_i P S_i(t); \]
            \[ min(t) = \min(estt_i(t), estt_k(t)); \]
            \[ min(t_m) = \min(estt_i(t_m), estt_k(t_m)); \]
            if \( (min(t_m) < min(t)) \) {
                if (node \( k \) is not locked) {
                    lock node \( k \);
                    migrate task \( p_{ij}(t) \) to node \( N_k \);
                    unlock node \( k \);
                    \[ estt_k(t) = estt_k(t_m); \]
                    break;
                }
            }
        }
    }
}

Figure 5.9: Migration – function Schedule() – a locking version

was used in the experiments. In the “standard” function Schedule() a node checks for every task if migration is profitable. If it is, it exits the loop. The variant in which the loop was continued until all the tasks were checked was also examined and tested. It resulted in too heavy traffic in the network and sometimes the assignment got unstable and it took much time to reach a stable state. The same case was analyzed with the “standard” function Schedule() (Figure 5.7). One can notice that there is only one possible migration for one neighbor in function Schedule() and migration is performed to the first profitable node not the one with the smallest estt value as in function Spawn().

5.2.7 Internal DAMPVM messages

A kernel process sends the following kinds of messages to its logical neighbors when a respective event occurs (these are the most important messages only):

- PC_COMM_TIME_RESP – the echo–response to the message request PC_COMM_TIME_REQ which is supposed to measure the bandwidth
while (true) {
    switch (event) {
        case NEW_SET_NP(i, t):
            Spawn();
            break;
        case NEW_TASK_Pi СтаrтЕd_оn_ТhIs_NоDе:
            register task p as \( p_{i+1} = \text{CP}(i, t) \) and update \( \text{CP}(i, t) \);
            store instr\( i_{\text{CP}(i, t)}(t) \) and \( \text{PS}(i, t)_{\text{CP}(i, t)}(t) \) of the new task;
            update \( \text{estt}_{i}(t) = \text{estt}_{i}(t-1) + \)
            \( \frac{1}{1 - \alpha_{\text{CP}(i, t)}} \) \( \text{instr}_{i_{\text{CP}(i, t)}}(t) \)
            send \( \text{estt}_{i}(t) \) UP to the logical neighbors;
            break;
        case TASK_Pi(t)_FINIshЕd_оn_ТhIs_NоDе:
            remove task \( p_{ij}(t) \) from the database;
            update \( \text{estt}_{i}(t) = \text{estt}_{i}(t-1) - \)
            \( \frac{1}{1 - \alpha_{\text{CP}(i, t)}} \) \( \text{instr}_{i_{\text{CP}(i, t)}}(t) \)
            send \( \text{estt}_{i}(t) \) DOWN to the logical neighbors;
            break;
        case \( \text{estt}_{k}(t) \) UP received:
            update \( \text{estt}_{k}(t) \);
            break;
        case \( \text{estt}_{k}(t) \) DOWN received:
            update \( \text{estt}_{k}(t) \);
            Schedule(); // try to migrate tasks
            break;
        case NEW_оthL_k(t) received:
            update \( \text{othL}_{k}(t) \);
            update \( \text{estt}_{k}(t) = \text{estt}_{k}(t-1) - \)
            \( \frac{1}{1 - \alpha_{\text{CP}(i, t)}} \) \( \text{estt}_{k}(t-1) \)
            for neighbor \( N_k \);
            if \( (\text{othL}_{k}(t) < \text{othL}_{k}(t-1)) \) Schedule();
            break;
        case instr\( i_{\text{CP}(i, t)} \)_CHANGED:
            update instr\( i_{\text{CP}(i, t)} \);
            update \( \text{estt}_{i}(t) \);
            if \( (\text{instr}_{i_{\text{CP}(i, t)}}(t) < \text{instr}_{i_{\text{CP}(i, t)}}(t-1)) \)
            send \( \text{estt}_{i}(t) \) DOWN to the logical neighbors;
            else {
                send \( \text{estt}_{i}(t) \) UP to the logical neighbors;
                Schedule();
            }
            break;
    }
}

Figure 5.10: Migration and allocation algorithm for node \( N_i \)
5.2. DAMPVM PROCESSING CONCEPT

![Diagram of a link between two nodes.

- **PC_PROCESSING_TIME_UP** – estimated execution time of processes on the node which has sent this message has increased – new estimated time is sent; sending this kind of a message can be caused by:
  - the creation of a new process on the node which has sent this message
  - the situation when a migrated copy from a different node appears on the node which has sent this message
  - the situation when a certain process on the node which has sent this message informed its kernel process that the estimated number of its instructions had increased

- **PC_PROCESSING_TIME_DOWN** – estimated execution time of processes on the node which has sent this message has decreased – new estimated time is sent

- **PC_NEW_OTHERS_CPU_LOAD** – the load of the node (by other users or system processes) which has sent this message has changed considerably (DAMPVM uses the 10% threshold) – a new load is sent

Apart from that, a kernel process stores up-to-date information concerning the state and some parameters of its logical neighbors, as stated before. It also knows the relative processing speeds of these nodes. After the start of the whole system kernel processes measure the speeds of all the nodes and every kernel process finds out the speeds of itself and its logical neighbors. They are used in the allocation and migration algorithms.

Additional messages are exchanged between processes of an application and the kernel processes. These messages are sent by application processes to their respective kernel processes. They are hidden in the code of the DAMPVM library. These are the most important ones:
• PC\_SPAWN\_REQ – a process wants to start a new task(s)
• PC\_NEW\_TASK – a new task wants to register in the system
• PC\_TASK\_FINISHED – a task has finished its execution and wants to logout from the system
• PC\_MIGRATION\_ENABLE – a process enters the state of enabled migration: from this moment DAMPVM is allowed to migrate this process
• PC\_MIGRATION\_DISABLE – a process enters the state of disabled migration: from this moment DAMPVM is not allowed to migrate this process
• PC\_WAIT – a process enters the idle state: it is waiting for messages from other processes
• PC\_INSTRUCTIONS – a process tells the system the number of instructions left for it

5.3 Using DAMPVM

Installation steps needed to start DAMPVM as well as a short programming tutorial are presented in appendix A. In general, programming applications for use with DAMPVM is as easy as programming PVM except the proper construction of the application for migration (including functions PackState() and UnPackState()). The example of a real application used for testing DAMPVM is shown in appendix C. The list of functions which can be used in DAMPVM applications is presented and explained in appendix B.

In general, the idea is the following: one writes an application code (analogous to the PVM code), then starts DAMPVM (appendix A) and runs the application. The user “starts and forgets”. DAMPVM tries to allocate the application processes optimally and performs migration if necessary. This can be caused by e.g. another user who started their tasks on a certain machine(s) or termination of some application processes. DAMPVM was created to adapt to the changes of the environment in order to execute its applications as fast as possible.

DAMPVM is heterogeneous just like PVM but it uses a few extra lines of code which are machine dependent. Now it works on Linux, Solaris and HP-UX. In the future other versions should be also available.
Chapter 6

Process identification in DAMPVM

6.1 Process identification solution

The solution described in this chapter has been implemented in DAMPVM and it gives good results as shown in chapters 7 and 8. It is general so it could be implemented easily in other message passing environments which support group and context mechanisms.

Moreover, the author proposes an idea how to eliminate the message forwarding mechanism (see section 6.2). In general, processes can be decomposed in such a way that message forwarding becomes redundant which can result in better performance of the system.

6.1.1 Identification in a migration context

In message passing environments like PVM processes are identified by their unique identifiers. They use them whenever they want to send a message to a different process or get a message from a certain process. In PVM there are "tids". Every process has its own "tid" which is unique in the scope of the whole virtual machine. In the MPI specification instead, which is also based on the processes model, processes are identified by successive integer numbers. In such environments processes can communicate with each other:

- one-to-one – a message is sent to one specified receiver
- one-to-many – a single message can be sent to a definite group of receivers by the "broadcast" or "multicast" operations
The point of a migration mechanism is that a working process is interrupted and restarted on a different machine from the point at which it was interrupted. Actually, a new process should be created on another computer which would start its execution from the final state of its predecessor. The solution to this problem proposed by the author differs from others which can be found in the literature.

A newly created process which is supposed to be the migrated process is, of course, known in the system by a different identifier than its predecessor. This is because in environments like PVM a new process has its own "tid" and a programmer is not able to assign a specific identifier to a process. They are set by PVM itself. The same is with MPI and others. All it means that other processes which would like to communicate with the process which has just been migrated can see its old identifier. Actually they can see the identifier of a non-existing process because after a new copy is created the old one is useless and should not receive or send any messages any more. Usually, so-called "virtual tids" are introduced to solve this problem like in MPVM. Actually, there are the maps of identifiers which translate the identifiers of migrated and currently non-existing processes to the identifiers of their migrated successors. Such a solution is a bit difficult and inconvenient to create and manage. One of the following mechanisms can be used to manage this problem (every one is costly):

1. sending the identifiers of migrated copies to all existing processes which can be too costly in the case of a big number of processes and frequent migration

2. introducing separate processes which would deal with translating identifiers – they must be notified about every migration occurrence

3. creating processes–mailboxes; already migrated copies would act as mailboxes – they would send messages further to their descendants (migrated copies) – this could result in many such processes holding too many resources of the system and too much communication in the system

6.1.2 Process identification mechanisms used in DAM-PVM

The proposed solution is based on two mechanisms (both of them present in PVM 3.4):

1. the group mechanism
6.1. PROCESS IDENTIFICATION SOLUTION

- processes can dynamically create new groups and enter them
- a process can send messages to all the members of a group even if
  it is not a member of this group

2. contexts – this element has been introduced in PVM 3.4

- a context is the additional parameter of a message like a tag
- contexts can be dynamically created and released by processes
- a message sent in a certain context can be received exclusively in
  the same context by a receiver process

The point of the solution is that every process of an application has its
own unique PCTid which is the following structure:

typedef struct {
    int pcschedulertid;
    int spawnnumber;
    int context;
} PCTid;

In fact, a process is uniquely identified by the first two elements of this
structure that is the PVM identifier of the kernel process running on the same
machine (PCScheduler – there is only one copy of this process on every node)
and a successive number of a process started on this node (spawnnumber)
(actually this is the number of the spawned processes on this node). All the
processes started on a certain node get these numbers from the kernel process
on this node. This process gives successive integer numbers spawnnumber to
started processes.

Consequently a new process (not migrated) starts its life in the following
way:

1. finds out what the "tid" of the kernel process on the same node is –
   pcschedulertid
2. gets its spawnnumber from this kernel process (as well as other infor-
   mation)
3. creates its own context identified by the value of variable context – at
   this point it already has the complete structure PCTid
4. sets its current context to the newly created context context
5. creates a group with name "\textless pcschedulertid\textgreater _x\textless spawnnumber\textgreater ", where \textless pcschedulertid\textgreater is the value of variable pcschedulertid written as a string and similarly \textless spawnnumber\textgreater is the value of variable spawnnumber written in the same way — the group with such a name is surely unique in the whole PVM machine

6. enters the created group

A migrated copy of a process starts its life in the following way:

1. gets the values of pcschedulertid and spawnnumber (and other information) from the previous copy and sets them as its own values

2. gets context context from the previous copy — at this moment it has the complete structure PCTid which is identical to the structure of the previous copy so other processes do not see many copies — they see only one process with the constant PCTid

3. sets its own context to the received context context

4. creates a group with name "\textless pcschedulertid\textgreater _x\textless spawnnumber\textgreater ", where \textless pcschedulertid\textgreater is the value of variable pcschedulertid written as a string and similarly \textless spawnnumber\textgreater is the value of variable spawnnumber written in the same way — this group has, of course, the same name as the group of the previous copy

5. enters the created group

The following interaction diagrams show these operations: Figure 6.1 — the creation of a new process and Figure 6.2 — process migration.

Now communication operations will be described in detail. If a process wants to send a message to a process having a certain PCTid it sends a "broadcast" to the group with the name from the receiver's PCTid, as described above. One can notice that it is not very costly because this group consists of only one member. Only for short time during a migration process there are two processes in this group i.e.: a migrating process and its new successor — a new copy. In this way a message is sent to the receiver process.

What is very important a message is sent in context PCTid.context of the sender. Receiving a message from the sender with a certain PCTid has to be performed as receiving all messages from anybody in context PCTid.context of the required sender. It will result in getting a message only from the required sender because every process sends messages in its own unique context so receiving in a certain context is receiving from one process only. Like in
6.1. PROCESS IDENTIFICATION SOLUTION

![Diagram showing process creation]

Figure 6.1: Process creation

PVM a process can also wait for messages from anybody. It can be done by receiving in an indeterminate context (-1).

The functions for sending and receiving has been implemented as shown in Figures 6.3 and 6.4.

In this way migration transparency has been achieved – this algorithm is transparent to all other processes. As a matter of fact, many other initiating operations are performed. Here only the main idea of the solution to the migration problem was presented.

6.1.3 Migration in a process

Transferring the state of a process to its migrated copy to enable the execution from the interrupted point also differs from the other known solutions like MPVM or Condor. As mentioned before, in order to enable the migration of a certain process the programmer who writes the code of a process has to write two extra functions for this process:

- **PackState()** – the function which packs the current state of a process
- **UnPackState()** – the function which unpacks the previously packed state by function PackState()
Figure 6.2: Process migration

DAMPVM uses the PVM message handler mechanism to interrupt a process and execute function \texttt{PackState()} asynchronously. The code of the process and functions \texttt{PackState()} and \texttt{UnPackState()} have to be written properly to enable a process to continue its computations from the interrupted point. After unpacking the state using function \texttt{UnPackState()} a process should skip to the interrupted point (it knows that from the variables in the packed state) and carry on its work from this point. As an example you can look at the code of an exemplary program which is shown in appendix C. This program was used for the experiments and tests of DAMPVM.

One can notice that packing a coherent state of a process (function \texttt{PackState()} is called asynchronously) can be easy or more difficult. A more detailed analysis of this problem shows that it is easy when a process creates new output data (different variables than input data) e.g. during matrices multiplication. It can be harder if input data are modified i.e. during vector sorting like in the exemplary application in appendix C.

In DAMPVM, a process can dynamically allow or forbid to be migrated. It can decide, if DAMPVM is allowed to migrate it. This is useful if there are very complex instructions in a process code and migration should not interrupt them because the process state in this section would be very difficult to pack. If a programmer does not want to support functions \texttt{PackState()}
int PC_Send(PCTid pctid, int msgtag) {
    char tempgroupname[30]; // the name of receiver's group
    sprintf (tempgroupname, "%dx%d", 
              pctid.pcschedulertid, 
              pctid.spawnnumber); // converts the pctid to its corresponding group name
    pvm_setcontext(mypctid.context);
    pvm_bcast(tempgroupname, msgtag);
}

Figure 6.3: function PC_Send()

and UnPackState(), processes will not be migrated at all. Functions PC_MigrationEnable() and PC_MigrationDisable() can be put in the code to allow or forbid migration of the calling process respectively. If function PC_MigrationEnable() was called somewhere in the code, it is valid till the PC_MigrationDisable() occurrence which disables migration. If not activated, migration of a process is disabled and should be activated by PC_MigrationEnable().

Similarly to critical sections in Dynamic PVM, migration should be disabled when communication functions are executed. This means there should be function PC_MigrationDisable() before a communication section and function PC_MigrationEnable() after that. Concrete solutions and optimizations of these depend entirely on a programmer.

In other solutions the state of a process is packed as visible from the operating system point of view. In such a case after the interruption of a process the following elements of the process must be saved: open files, allocated resources like the memory, various segments, buffers, semaphores, shared memory content etc.

6.1.4 Advantages and disadvantages of the solution
The solution of the author has three advantages over known strategies:

1. enables migration in a heterogeneous environment i.e. between com-
int PC_Recv(PCTid ptid, int msgtag) {

  int oldcontext =
    pvm_setcontext(ptid.context);
  // set the context to the context
  of the sender or its migrated copy
  (they all set the same contexts)

  pvm_recv(-1, msgtag);

  pvm_setcontext(oldcontext);
}

Figure 6.4: function PC_Recv()

puters of different architectures (like i386 and Sun)

2. the size of a process state is often much smaller than in other solutions:
in MPVM, for instance, a process state during migration is often much
bigger than necessary

3. possible various modes of work – described in detail in subsection 8.3.1

While manipulating on matrix $Nb$ bottleneck links in the network can
appear. They could stop the flow of processes from a certain part of the net-
work to another as shown in Figure 6.5. Fortunately, one can add additional
links as also shown in Figure 6.5 to increase the flow.

The disadvantage is that currently DAMPVM does not support forward-
ing messages which could be sent just before or during a migration process.
This can result in lost messages. Fortunately, as explained below, it is not a
problem, especially for computational tasks. There is a general idea given in
6.2 how to get rid of message forwarding so that it is not needed at all. A
simple algorithm is given which could be implemented to achieve this auto-
matically for computational tasks.

6.2 How to get rid of message forwarding?

One has to realize that the following scenario can occur regardless the mecha-
nism of migration is considered. Process $P_B$ sent a message to the migrating
process $P_A$. For some reason this message was going to process $P_A$ very slowly so that in the meantime process $P_A$ was interrupted, its state was packed, the process was migrated to a new copy on a different machine and was killed as a useless process (because its successor was already active). In this case the message addressed to the old copy would not reach either the old or the new one. One can prevent from such situations by keeping the old copy alive for some time as a mailbox. It is shown in Figure 6.6. It would only send "slow" messages to the new copy. Unfortunately, it can be very costly if many successive migrations occur. Of course, it can be solved in many ways. The second migration could be blocked until the mailbox exists. After the mailbox is killed (after some time) migration could be enabled again. DAMPVM does not solve this problem in this way because it could harm the performance of the system and its scalability. On the other hand, as proved below, it does not confine the applications in any way if we consider the sort of applications DAMPVM was designed for. In computational applications (the main applications for this system) e.g. matrices operations, image processing etc. processes perform three kinds of operations:

1. Receiving input data

2. Processing data (the main activity of a process capturing the majority
of its working time)

3. Sending output data (results)

In this case the solution is very simple. There is no problem with message forwarding – it is not necessary – if migration is enabled only after receiving input data (as it was written in the exemplary application in appendix C). In very sophisticated processes (many successive sequences of send and receive operations) they can be divided into many processes each of which performing only the above 3 steps of execution and nothing more. A complex process can be divided into processes starting from the beginning of its code and cutting successive parts between send and recv operations. An example is shown in Figure 6.7.

New processes can send processed data to the new successive processes and fast and effective migration is possible then. This is not a problem in the case of computational applications which are considered in this work. If it is not activated, DAMPVM will not migrate a process. It should call function PCMigrationEnable() to have a chance to be migrated. This function can be called dynamically by a process. There is also function PCMigrationDisable() which disables the possibility of migration. The
main(...) {

    // process 1
    recv;
    recv;
    processing data;
    send;
    send;

    // process 2
    recv;
    processing data;
    send;
    processing data;
    send;
    send;

    // process 3
    recv;
    recv;
    processing data;
    recv;
    processing data;
    send;

}

**Figure 6.7: Dividing a process into smaller ones**

The author claims that the general solution to the problem described above would be much costlier (i.e. it would involve message forwarding).

There are also other solutions to this problem. It is possible to create a precompiler which could analyze a process code, divide it into smaller processes and generate their codes. This would eliminate the necessity of forwarding messages at all in a general case. This leads to dividing a code into sections as described above. Then one can form two processes from one as done in Figure 6.8. There is an initial process (left) and two derived processes (right). The first section goes to the code of the first process (right top) and the second section to the second process (right bottom). The
* main(...) {
  *   // added by the precompiler
  *   spawn(&section2id);
  *   // end of added by the precompiler
  *
  *   int a,b,c,d,e;
  *   a=b=3;
  *   recv(-1,tag1); unpack(&a);
  *   recv(-1,tag2); unpack(&b);
  *   c=a*b;
  *   d=b/c;
  *   recv(-1,tag1); unpack(&a);
  *   recv(-1,tag2); unpack(&b);
  *
  *   // added by the precompiler
  *   pack(&c); send(process1,20);
  *   if (a>=3) {
  *     pack(&d); send(process2,10);
  *     // added by the precompiler
  *     pack(&c); send(process1,20);
  *   }
  *   // possible int a=1;
  *   pack(&d);send(process2,10);
  *}
  *
  * main(...) {
  *   // added by the precompiler
  *   // Start of section 2
  *   recv(process3,tag3);
  *   unpack(&e);
  *   c=d*e;
  * }
  *   d=a+b;
  *   unpack(&d); unpack(&e);
  *   pack(&c);
  *   // end of added by the precompiler
  *   send(process1,11);
  *   send(process2,12);
  *
  *   // End of section 2
  *   recv(process3,tag3); unpack(&e);
  *   c=d*e;
  *
  *   // } removed by the precompiler
  *   d=a+b;
  *   pack(&c);
  *   send(process1,11);
  *   send(process2,12);
  *}

Figure 6.8: Initial process (left) and two processes derived from the initial one (right)
6.2. HOW TO GET RID OF MESSAGE FORWARDING?

changes the precompiler should introduce are the following:

1. send all the variables declared in section 1 to the second process at the end of the first process code

2. add endings for interrupted blocks (as in the example – right top)

3. declare all the variables declared in section 1 at the beginning of the second process code

4. receive all the variables from the first process at the beginning of the second process code

5. remove unnecessary endings of blocks which began in the previous section (as in the example – right down)

The result of this procedure is shown in Figure 6.8. Of course, the analogous procedure could be applied for the third, fourth and next sections.

6.2.1 Limitations

Unfortunately, it seems to be difficult to solve this problem generally. Although the above example shows that conditional instructions could be overcome, other situation may occur. These are the examples:

- there could be another instruction in the code of the initial process as indicated as a comment int a=1; in this case the previous value of variable a has been stored on the stack; the above method will result in the wrong value of a after the canceled block; in other words this method does not allow for the history of variables (hiding one variable by another)

- there could be problems with the recv; send; sequences in loops; on the other hand, such sequences seem to occur in loops rarely

The above limitations should not harm typical computational applications so an automated tool for them could be created. Of course, some more complex methods could be developed to solve the limitations described for any process automatically. This seems to be a good area for researching.
6.3 Visualization of data in DAMPVM

In the system with process migration a special mechanism for data visualization must be supported. A process should not print its output data to the standard output because the connection to the previous standard output would be lost after migration to a different machine. The same problem is with files – if there is no NFS\(^1\), the output file with data should be packed and restored as a part of the process state – it is also inconvenient and costly. The DAMPVM library provides a special data type – PCTextWindow. If a programmer creates a new object of this type in the code of a process (using operator \texttt{new}) a new graphical window is created on a specific (given) computer. From now, the process can send various data types to this window like to the standard output i.e. \texttt{int}, \texttt{char *}, etc. The process simply uses the pointer to the window. This is clearly visible in the exemplary application in appendix C. Obviously, a process can open as many windows as it wishes to. Special functions for packing and unpacking windows are supported. They are described in appendix B. In this way visualization is very easy even after many migrations of a process. The window for the root process of the application from appendix C is visible in Figure 6.9.

![PCTextWindow](image)

Figure 6.9: PCTextWindow

The PCTextWindow is itself a PVM application which waits for messages to print data. It would be possible to create other data types similar to PCTextWindow e.g. graphs (2D, 3D, linear, polar etc.) or other components which could be used by processes running in parallel in the network. One can notice that several processes could share the same window or the same

\(^1\)Network File System
components. This could be used for nice data visualization and be easily implemented in DAMPVM.
Chapter 7
Experiments & analysis

7.1 Migration need

If migration is not possible, the total execution time depends entirely on the assignment algorithm. This is because tasks once started can not be moved to different machines. They have to run where started until they finish their work. One can notice that the assignment algorithm can take into account only the parameters known when starting tasks. It is not possible to know the loads of the nodes by other users in advance e.g. after 1 minute from the last assignment. Of course, the best case is when this algorithm makes such an assignment that all the nodes have equal execution times (including the loads by other users). But if these loads change after some time the differences in execution times on the nodes will arise. Unfortunately, they can not be improved because there is no migration.

One can ask if migration is needed to perform a good assignment and achieve the least possible execution time of an application. As mentioned above the assignment algorithm can make the best possible assignment every time a new task is created. If this is the case we have:

\[ estt_1(t_a) = estt_2(t_a) = \ldots = estt_i(t_a) = \ldots = estt_n(t_a) \]

\[ \forall i \in \{1, 2, \ldots, |N|\} \quad estt_i(t_a) = T \]

where \( t_a \) is the moment of the last assignment. In this case the total execution time is \( T \) (from \( t_a \) moment) and it cannot be reduced any further. Of course, it can rarely be done because it depends on the sizes of processes (granularity) and the speeds of the nodes. Sometimes it can be done if execution times of processes are equal and processors of the same speeds are available. If not, usually it can be approximated only. From moment \( t_a \), if sets \( CP \) and
values $\theta h L$ remain the same, nothing can be improved and the assignment algorithm would do. Nothing more than PVM offers would be required.

Unfortunately, two events can occur in the meantime:

- the loads of the nodes by other users can change (e.g. a user finishes their jobs)
- application processes can have many possible paths of execution each of which of a different length (different numbers of instructions) and the number of instructions can change

In the second as well as in the first case the changes do not have to be known before and they are not in practice. In such cases certain processors can be relieved from some work and others can get more to do. As a result of this after some time some nodes have much work to do while others are idle. The scalability decreases greatly then which is not required. Such situations were met during the experiments which is described in section 7.2.

### 7.1.1 Migration adapts the assignment to other users

DAMPVM tries to solve the above problems (successfully) by enabling migration of processes. In Figure 7.1a one can see the optimal assignment described above (a system which contains 3 nodes). Every processor has the same amount of work to do i.e. 4 units of time. But at moment $t_n = 1$ a different user starts their tasks on processor 3. This is shown in Figure 7.1b. As a result of this they capture 50% of processor 3 so the total execution time for “our” application increases to 7 units of time. DAMPVM detects such situations because $\theta h L_3$ has changed and moves some tasks from node 3 to nodes 1 and 2. The gain on these operations is visible in Figure 7.1c. The total execution time for the application reduces to 4.6 units of time (optimally). That is a considerable profit. This example is very optimistic because it assumes zero migration costs and no delays. Moreover, it assumes the tasks are preemptive (in DAMPVM they are). But, in the real experiments the scalability shows that similar results may be achieved. Migration is performed when changes are noticed by DAMPVM i.e. at moment $t_n$. During the execution of an application many such operations may be required to balance work. It is obvious that balancing work on nodes leads to maximal reduction of the total execution time. Sometimes, of course, there is no sense in balancing work if migration costs are very high. In such a case, balancing would result in higher execution time. Actually, the area for processes of an application for each node should be the same for every node in the diagrams.
7.1. MIGRATION NEED

![Diagram showing optimal assignments and migration adaptation.](image)

Figure 7.1: Optimal assignments and migration adapts the system to other user activities

7.1.2 Optimal assignment without migration ≠ no idle time

A different situation may also occur. It is also mentioned in [47]. Even if the work of an application seems to be balanced, the execution times on separate nodes do not have to be balanced. This is because of precedence relations between processes i.e. a process cannot start its execution because it needs to wait for data from other processes. This is shown in Figure 7.2a. Syn denotes the synchronization needed. Idle time is introduced which decreases the scalability. Fortunately, in DAMPVM processes can dynamically tell the system in which state they are and how much instructions are left for them. In particular if process $p_{ij}(t)$ enters its idle state (waiting for something) it tells the system that it has temporarily $\text{instr}_{ij}(t) = 0$ (=a waiting state). In this way DAMPVM detects the state of a node. If process $p_{ij}(t)$ on node $N_i$ sets $\text{instr}_{ij}(t) = 0$, $\text{esst}_i(t)$ changes for node $N_i$ on which process $p_{ij}(t)$ resides and, if necessary, migration occurs. Finally we get the scenario shown in Figure 7.2b without idle time. Mi denotes the migration of processes from node 2 to node 1 and from 1 to 2 to avoid idle time. When a process (processes) at node 1 reduces its (their) $\text{instr}_{ij}(t_a)$, $\text{esst}_1(t_a)$ is also decreased. At this moment migration is performed from node 2 to 1 to balance $\text{esst}_1(t_a)$ and $\text{esst}_2(t_a)$. At moment $t_b$ the tasks which had reduced their $\text{instr}_{ij}(t_a)$ now increase them and $\text{esst}_1(t_a)$ increases. Now $\text{esst}_1(t_b)$ is bigger than $\text{esst}_2(t_b)$ and migration from node 1 to 2 is executed to balance the left work. This method of dynamic signalling how much instructions/work is left for a process seems to solve the
majority of idle time problems caused by synchronization. This could be the solution to the problem noticed in [47]. Migration Mi is performed immediately when it can reduce the execution time of the application.

![Diagram](image)

Figure 7.2: Idle time in spite of an optimal assignment

It is visible that migration is profitable. It has been confirmed by experiments described below where the above situations were simulated.

### 7.2 Description of experiments

Apart from the experiments described below some other variations of the "standard" algorithm were tested to achieve better performance of DAMPVM. They were described in subsection 5.2.6.

#### 7.2.1 Analyzed parameters

As mentioned in subsection 4.3.4 the goal of DAMPVM is to minimize the total execution time of an application. It depends on the assignment of this application to the system. Since finding the optimal one by looking over all possible assignments would take too much time, the experimental results are compared to the scalability calculated using equation 2.8. It means that this scalability is higher than the best possible to achieve in practice because migration times are higher than 0. Even if migration was not performed to spare on its cost, it would not be possible to divide the total execution time into perfectly equal parts to place on the nodes (in polynomial time). This is another NP-complete problem – the knapsack problem ([38]).
7.2. DESCRIPTION OF EXPERIMENTS

Figure 7.3: Application graph

Figure 7.4: Instrumentation of the application used in the experiments: Nodes: a. leaves b. non-leaves

7.2.2 Experiments environment

The network of up to 7 processors was used. The following computers were used in the experiments: coper01, coper02, coper03, coper04, coper05, nico10, nico13, nico14, jupiter (@eti.pg.gda.pl). Their relative speeds for the application described below are: 48, 48, 48, 48, 53, 61, 61, 61 and 121
<table>
<thead>
<tr>
<th>System</th>
<th>Execution times without external load [s]</th>
<th>Execution times with external load without migration [s]</th>
<th>Optimal speed-up</th>
<th>Achieved speed-up</th>
<th>Achieved speed-up/ Optimal speed-up 100% [%]</th>
<th>Time loss without migration [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>coperG1</td>
<td>2615</td>
<td>--</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>coperG1 + coperG2</td>
<td>1342</td>
<td>1355</td>
<td>1358</td>
<td>2</td>
<td>1.949</td>
<td>97.43</td>
</tr>
<tr>
<td>coperG1 + coperG2 + coperG3</td>
<td>910</td>
<td>1052</td>
<td>3</td>
<td>2.874</td>
<td>95.79</td>
<td>24.9</td>
</tr>
<tr>
<td>coperG1 + coperG2 + coperG3 + nice10</td>
<td>748</td>
<td>862</td>
<td>3.787</td>
<td>3.496</td>
<td>92.32</td>
<td>40.3</td>
</tr>
<tr>
<td>coperG1 + coperG2 + coperG3 + nice10 + nice13</td>
<td>630</td>
<td>729</td>
<td>4.574</td>
<td>4.151</td>
<td>90.75</td>
<td>43.3</td>
</tr>
<tr>
<td>coperG1 + coperG2 + coperG3 + nice10 + nice13 + nice14</td>
<td>548</td>
<td>631</td>
<td>5.361</td>
<td>4.772</td>
<td>89.00</td>
<td>53.9</td>
</tr>
<tr>
<td>coperG1 + coperG2 + coperG3 + nice10 + nice13 + nice14 + jupiter</td>
<td>525</td>
<td>592</td>
<td>5.757</td>
<td>4.981</td>
<td>86.52</td>
<td>63.2</td>
</tr>
</tbody>
</table>

Table 7.1: Experiments results - initial vector size=400000, sorted vector size=20000

respectively as far as the execution times of a selected program is concerned. The network is heterogenous since copers are PCs with Linux, nicos are Sun SPARC Stations with Solaris and jupiter is an HP with HP-UX.

7.2.3 Analyzed application

In order to check the performance of the proposed solution the author wrote an exemplary application using the DAMPVM library. The code of the application with the instrumentation and state detection done as explained in subsection 4.3.3 is shown in appendix C. The creation tree of this application is shown in Figure 7.3. The same application with various input data was
### 7.2. Description of Experiments

<table>
<thead>
<tr>
<th>System</th>
<th>Execution times without external load [s]</th>
<th>Execution times without external load with migration [s]</th>
<th>Execution times with external load with migration [s]</th>
<th>Optimal speed-up</th>
<th>Achieved speed-up</th>
<th>Achieved speed-up/Optimal speed-up 100% [%]</th>
<th>Time loss without migration [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>copero1</td>
<td>170</td>
<td>--</td>
<td>--</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>copero1 +</td>
<td></td>
<td>88</td>
<td>140</td>
<td>104</td>
<td>2</td>
<td>1.932</td>
<td>96.59</td>
</tr>
<tr>
<td>copero3</td>
<td></td>
<td>64</td>
<td>126</td>
<td>85</td>
<td>3</td>
<td>2.66</td>
<td>88.54</td>
</tr>
</tbody>
</table>

Table 7.2: Experiments results - initial vector size=50000, sorted vector size=10000

<table>
<thead>
<tr>
<th>System</th>
<th>Execution times without external load [s]</th>
<th>Execution times without external load with migration [s]</th>
<th>Execution times with external load with migration [s]</th>
<th>Optimal speed-up</th>
<th>Achieved speed-up</th>
<th>Achieved speed-up/Optimal speed-up 100% [%]</th>
<th>Time loss without migration [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>copero1</td>
<td>342</td>
<td>--</td>
<td>--</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>copero1 +</td>
<td></td>
<td>178</td>
<td>261</td>
<td>219</td>
<td>2</td>
<td>1.921</td>
<td>96.1</td>
</tr>
<tr>
<td>copero3</td>
<td></td>
<td>121</td>
<td>180</td>
<td>145</td>
<td>3</td>
<td>2.826</td>
<td>94.2</td>
</tr>
<tr>
<td>copero4</td>
<td></td>
<td>98</td>
<td>155</td>
<td>114</td>
<td>3.906</td>
<td>3.49</td>
<td>89.3</td>
</tr>
<tr>
<td>copero5</td>
<td></td>
<td>94</td>
<td>146</td>
<td>104</td>
<td>4.693</td>
<td>3.638</td>
<td>77.5</td>
</tr>
</tbody>
</table>

Table 7.3: Experiments results - initial vector size=100000, sorted vector size=10000

used which formed a few experiments. Depending on the experiment, the application is to sort the vector of 400000, 50000, 100000 or 200000 integer elements. It uses the bubble sort method to achieve this. The first process which gets the vector divides it into two equal parts, creates two children and sends one part to one child. They do the same until the size of the vector in a process is less than 20000, 10000, 10000 and 10000 elements respectively.
<table>
<thead>
<tr>
<th>System</th>
<th>Execution times without external load [s]</th>
<th>Execution times with external load without migration [s]</th>
<th>Execution times with external load with migration [s]</th>
<th>Optimal speed-up</th>
<th>Achieved speed-up</th>
<th>Achieved speed-up/ Optimal speed-up 100% [%]</th>
<th>Time loss without migration [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>cposer01</td>
<td>654</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>cposer01</td>
<td>343</td>
<td>443</td>
<td>424</td>
<td>2</td>
<td>1.907</td>
<td>95.3</td>
<td>3.8</td>
</tr>
<tr>
<td>cposer03     + cposer04</td>
<td>243</td>
<td>345</td>
<td>284</td>
<td>3</td>
<td>2.691</td>
<td>89.7</td>
<td>21.5</td>
</tr>
<tr>
<td>cposer01</td>
<td>193</td>
<td>276</td>
<td>218</td>
<td>3.906</td>
<td>3.389</td>
<td>86.8</td>
<td>26.6</td>
</tr>
<tr>
<td>cposer03     + cposer04 + cposer05</td>
<td>177</td>
<td>273</td>
<td>197</td>
<td>4.693</td>
<td>3.695</td>
<td>78.7</td>
<td>38.6</td>
</tr>
<tr>
<td>nice10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7.4: Experiments results - initial vector size=200000, sorted vector size=10000

The leaves of the tree sort their own vectors and then send the sorted ones to their parents. They merge the received vectors and send them to their parents. Finally the root merges and the whole application exits. It means that when the leaves are sorting the other processes must wait for them.

An example of another application with a double loop is presented in appendix A.2. It could be modified to do some tasks and be used in DAMPVM.

Examples of real processes assignments to the nodes are discussed in subsection 7.2.6 and shown in Table 7.5.

**Instrumentation**

The concrete instrumentation used, is presented in Figure 7.4. There is the instrumentation for a leaf process in Figure 7.4a and for a non-leaf process in Figure 7.4b. A leaf takes a vector to sort, sets its instr = 1000 (PC_Instructions()), sends the results back and terminates. A non-leaf process sends two parts of its vector to sort and must wait for them. It sets instr = 0 (PC_Wait()) before function PC_Recv() and then has to merge only (merging estimated for instr = 100 – PC_Instructions()).
7.2. DESCRIPTION OF EXPERIMENTS

Migration and communication costs

It must be told that migration costs were low comparing to the computation time of this application. A vector of e.g. 100000 integer elements means 400000 bytes, that is less than 400 KBs. In the Ethernet network it means less than a second for a process to migrate (it depends on other users and parameters too). This is low in comparison to the execution times (minutes). As far as communication is concerned there was no more communication apart from receiving and sending the results back (similar times as estimated above).

7.2.4 Experiments definitions

There were three experiments performed corresponding to the scenarios shown in Figure 7.1. At first the application was executed in the system without an external load (except the X Window overhead). The external load was simulated in the following way. After about 50 seconds from the application start (all the application processes have already started) another user started their 5 processes performing computations (the external load) on coper01. For each size of the initial table (400000, 500000, 1000000 and 2000000) there were 3 sub-experiments. The first one concerned the case without an external load with migration enabled, the second one the case with the above external load but without migration. The last one referred to the situation with both the external load and the migration possibility (here the results were averaging from many measurements). One can notice that the second sub-experiment is similar to the PVM environment. Of course, there is a different allocation algorithm in DAMPVM but both PVM and this case does not allow for migration which seems to be one of the main DAMPVM advantages.

The results let us observe how much can be gained on migration and formulate some conclusions which are presented in the next chapter.

7.2.5 Experiments results

The results are shown in Tables 7.1, 7.2, 7.3 and 7.4 and in Figures 7.6, 7.8 and 7.10 respectively. Speed-up values are in Figures 7.5, 7.7 and 7.9. Without migration although work was balanced by the dynamic assignment processes had to stay on coper01. In the migration case some of them were moved to other nodes. One can notice that the losses in the tables show the effective percentages of the external loads on coper01 (the more processors the fewer processes on coper01). This proves that in hard changeable
environments (considerable changes of the external load – a big amplitude) migration is very useful (Tables 7.1, 7.2, 7.3 and 7.4). In general in static or middle changeable environments (only slight changes of the external load – a low amplitude) a dynamic assignment mechanism would do. This is because if there are only slight changes of an external load with a small amplitude, moving processes gives nothing but it is itself some overhead.

In the successive experiments there were 63, 15, 31 and 63 processes respectively which means 32, 8, 16 and 32 sorting processes (the leaves). There are 7 processors of different speeds so it is not possible to balance the work perfectly (too few processes and different speeds of the nodes). Thus
the results should be regarded as good. One can notice that migration can be useful even without an external load. This is the case when the granularity of an application is rather high because migration can actually divide processes into smaller parts. This is shown in Figure 7.11. There are 3 processes and 2 processors only. Migration can divide one process so that almost the least possible execution time is achieved (Figure 7.11b).

Actually, the situation with idle time shown before in Figure 7.2 also occurred during the experiments described above. Since it was not possible to achieve the best theoretical balance because of different speeds of nodes and a quite low number of processes the leaves on some nodes finished their work.
Figure 7.9: Speed-up - initial vector size=200000, sorted vector size=10000

Figure 7.10: Execution times - initial vector size=200000, sorted vector size=10000

before the other nodes. In this case the state detection mechanism initiated migration and the idle time on these nodes was removed or decreased.

### 7.2.6 Exemplary assignments

Exemplary assignments of processes to the nodes in the network are shown in Table 7.5. They refer to the stable assignments at the moment when the leaves were sorting the 3000-th iteration out of 6250 (the external loop in the application code – C). The initial table size for these four experiments was 100000 elements and the maximum sort size in the leaves was 10000 (the
real sort size was 6250). One can see how processes from the application graph (Figure 7.3) were assigned to the network. One can see that the most important task was to assign the leaves from the application graph because they had much bigger number of instructions indicated by function PC_Instructions().

Table 7.5: Exemplary assignments

<table>
<thead>
<tr>
<th>System</th>
<th>coper03</th>
<th>coper02</th>
<th>coper01</th>
<th>coper05</th>
</tr>
</thead>
<tbody>
<tr>
<td>coper03 + coper02</td>
<td>1, 3, 6, 7, 8, 13, 14, 15, 16, 20, 22, 25, 26, 27, 29, 31</td>
<td>2, 4, 5, 9, 10, 11, 12, 17, 18, 19, 21, 23, 24, 28, 30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>coper03 + coper02 + coper01</td>
<td>1, 4, 5, 8, 9, 12, 18, 19, 25, 27, 28, 30</td>
<td>3, 6, 7, 13, 14, 15, 16, 17, 21, 29, 31</td>
<td>2, 10, 11, 20, 22, 23, 24, 26</td>
<td></td>
</tr>
<tr>
<td>coper03 + coper02 + coper01 + coper05</td>
<td>1, 5, 6, 13, 16, 17, 31, 22</td>
<td>4, 7, 12, 14, 15, 18, 20, 24, 25</td>
<td>3, 9, 10, 19, 23, 28, 29</td>
<td>2, 8, 11, 21, 26, 27, 30</td>
</tr>
</tbody>
</table>
Chapter 8

Final remarks

8.1 Experiments discussion

The experiments which results were shown in the previous chapter were to help finding relations between migration and the scalability of DAMPVM. In other words how does migration improve scalability? In this section a few lemmas will be presented as well as advantages and disadvantages of migration will be discussed. It must be emphasized that in the experiments there was a small number of processes comparing to the number of the nodes in the network. This fact is one of the reasons for the lemmas presented below. They need not be true for other proportions of the above factors.

Lemma 1 For a selected application with tree time dependencies between its processes and structure as in Figure 7.3 and another user who starts a fixed number of processes after the considered application, the lower number of processes per node the higher relative gain on migration versus the system without migration.

Proof.

Let us assume that another user starts $u$ processes (like in the experiments) and the considered application consists of $a$ processes. One can treat all the processes of the same size so the total execution time with migration will be (assuming that another user’s processes were started just after the considered application ones):

$$t_{migration} = \frac{a + u}{\sum_{i=1}^{|\mathcal{N}|} s \pi_i}$$

where $|\mathcal{N}|$ is the number of processors in the network and the execution time of a process equals 1 (one can assume this without losing generality). Let
us assume that another user started their processes on node $N_i$. Before this there were

$$a \frac{sp_i}{\sum_{i=1}^{[N]} sp_i}$$

processes of the considered application (optimal assignment assumed) on node $N_i$. Finally the total execution time on node $N_i$, which is also the total execution time of the whole application when no migration is performed, equals:

$$t_{no\ migration} = \frac{a \frac{sp_i}{\sum_{i=1}^{[N]} sp_i} + u}{sp_i}$$

Now time loss without migration can be counted as:

$$\frac{t_{no\ migration}}{t_{migration}} - 1 = \frac{aspi + u \sum_{i=1}^{[N]} sp_i}{(a + u)sp_i} - 1 = \frac{aspi}{(a + u)sp_i} + \frac{u \sum_{i=1}^{[N]} sp_i}{(a + u)sp_i} - 1 \quad (8.1)$$

It is visible that the first addend does not depend on the number of processors. Now if one considers a certain system with a certain number of processors, if another node is added the second addend is increased by $usp_{new\ node}$. This means the time loss is also increased by $usp_{new\ node} 100\%$.

The above lemma was verified by the experiments. In particular, for the described sorting problem and the size of the initial vector as well as the sizes of vectors in the leaves, the number of available nodes was increased. One can notice that the higher the number of available machines the higher the gain on migration is. The reason for this is that the number of active processes in the application – these are the leaves mainly (8, 16 or even 32) is comparable to the number of processors in the network which results in idle time, as explained below. Time dependencies between processes cause that some processes do their tasks but stop and have to wait for the results from other processes. One can notice that if all of the processes on a certain node stop, this node becomes idle i.e. it has nothing to do. If there is no migration, stopped processes have to stay on it and wait for the data from their logical predecessors in the application. The time they have to spend on waiting decreases the speed-up, of course. Alternatively, if there is migration enabled, DAMPVM notices that a node is idle and migrates processes from other nodes to the relieved one. They can use the free node. In this case, after stopped processes restart their computations, DAMPVM has to perform another load balancing by moving some tasks from this node to the others. Now, if there are more processors in the network the number of processes per
node is lower so the probability a node becomes idle is greater. The more processors the greater probability that all processes on a node enter their idle states (waiting for the results). As described above migration solves this problem and this also confirms the lemma. This has also some disadvantages. If the environment changes very quickly on end, process flickering may occur. One of the solutions to this problem is to increase the cost of migration (e.g. by increasing the $\zeta$ value) which decreases its occurrence.

Apart from that one can notice similar relations between the vectors (various experiments). In the experiments with the initial vector sizes of 50000, 100000 and 200000, the sizes of the vectors in the leaves are identical. Only the numbers of processes are different. For a given number of processors in the system the gain on migration is lower for higher sizes of the initial vectors = the greater number of processes per node. For example, for the 3–processor system the gain on migration is lower for 200000 than for 100000 and similarly it is lower for 100000 than for 50000. This is true for any number of processors in the system. Now, using equation 8.1, it will be proved. Equation 8.1 can be written as follows:

$$\frac{t_{\text{no migration}}}{t_{\text{migration}}} - 1 = \frac{as_p + u \sum_{i=1}^{N} sp_i}{(a + u)sp_i} - 1 = \frac{as_p + us_p}{(a + u)sp_i} + \frac{u \sum_{i\neq 1} sp_i}{(a + u)sp_i} - 1 = 1 + \frac{u \sum_{i\neq 1} sp_i}{(a + u)sp_i} - 1 = \frac{u \sum_{i\neq 1} sp_i}{(a + u)sp_i}$$  \hspace{1cm} (8.2)

Changing initial vector sizes if sorted sizes are constant means changing the number of processes i.e. $a$. It is clearly visible that the higher $a$ in equation 8.2 the lower migration loss or gain. This proves the lemma. Relative losses without migration are also presented in Figure 8.1.

Now equation 8.1 can be compared to the values from the experiments. Since the leaves do the majority of work in the application they can be regarded as the only processes of the application if the execution time is long enough. This can be simplified in this way for the case with the initial vector size of 400000 elements and the sorted size of 20000. Now one can assume there are active $a = 32$ processes (leaves) and $u = 5$ other user processes. Now for each number of nodes a theoretical migration loss can be calculated. The results are shown in Figure 8.2. The bigger difference for the bigger number of the nodes is caused by the decrease of the scalability.

On the other hand, there can be other applications with much different time interdependencies (a different application graph) and this lemma need not be true. If there is for instance only one node which depends on others it could be much different. The above conclusions concern this particular application graph and the tested application.
**Figure 8.1:** Relative loss without migration

**Figure 8.2:** Relative loss without migration - initial vector size=400000, sorted vector size=20000

**Lemma 2** For a given number of processors in the system and the number of processes of an application with time dependencies as in Figure 7.3, the more work processes have to execute (it takes more time) the greater scalability.

As in the previous example, the discussion will follow on the base of the experiments results. The experiments were constructed in such a way that there were two of them with the same number of application processes but of different lengths i.e. with the initial vector sizes 200000 and 400000. For each number of the nodes in the system the scalability is greater for the longer application (longer processes) i.e. 400000. It is obvious because the time needed to start processes and reach a stable assignment of processes
8.2 Final conclusions

Now the final conclusions of the experiments can be formulated.

The experiments showed that the mechanisms implemented in DAMPVM and described above were useful i.e. gave good scalability of an application in a distributed environment. It means that DAMPVM can be used for shortening the execution time of various applications, in particular computational ones. DAMPVM works well in changeable environments where other users start their own time-consuming tasks. This was confirmed by the experiments. It was visible that the migration mechanism could even divide processes into smaller parts (processes execution is preemptive) as in Figure 7.11 and improve scalability. It means that the migration mechanism is very useful when the granularity of the application is low (few processes versus the number of nodes).

8.3 Tendencies

8.3.1 Future work

There are many possible improvements to be tested. These can be the following:

- implementing an automated tool which could decompose processes into smaller parts as described in section 6.2; this could eliminate message forwarding

- implementing various algorithms – DAMPVM is a good base for testing the performance of algorithms and improving them

- testing DAMPVM for more nodes and implementing other mechanisms useful for a big number of processors in the network

- implementing other graphical data types or components similar to PCTextWindow as explained in section 6.3

Depending on the target environment, in the future DAMPVM can be equipped with a few working modes. They would refer to the DAMPVM reactions to the changes of the environment. These are the proposed working modes:
• "polite" – if the load of a certain node is high (e.g. higher than 60%) DAMPVM does not either start or migrate tasks on/to this node because this would slow down the programs executed by other users on this node – this is the mode implemented in the current version of DAMPVM

• "very polite" – the same as above and additionally if a certain user starts their processes on node $N_i$, DAMPVM migrates its processes from this node to some other ones to relieve node $N_i$ for the other user

• "normal" – DAMPVM competes with other users = tries to make the most of all available nodes for a given application (this policy can slow down other users processes significantly)

The above modes enable applications to make the most of cheap networks of workstations (e.g. PCs with Linux) so that time-consuming tasks can be executed with high scalability and they do not slow down the actions and programs of "normal" users (similarly to Condor).

The logical structure of a network (matrix $\mathbf{N_b}$) also seems to have some influence on the scalability of DAMPVM, if the number of nodes is high. For a few nodes only (as in the performed experiments) the "all-to-all" connections are even better (higher scalability) than other structures like a ring or a bus. This was confirmed by the experiments with DAMPVM.

### 8.3.2 Possible extensions

As the next steps, the extensions of DAMPVM can be obtained. Other mechanisms and models could be implemented and tested. The possibility of dynamic detection of process dependencies should be discussed and its performance ought to be evaluated. It seems that this feature could be profitable in big applications but give a big unprofitable overhead in small ones. Now, detection of a process state is possible if certain functions ($\text{PC\_MigrationEnable()}$, $\text{PC\_MigrationDisable()}$, $\text{PC\_Wait()}$) are called in the process code. Dynamic detection of process dependencies can be less useful than state detection developed in the current release of DAMPVM. This should be verified experimentally.
8.4 Implemented extensions

The author has also implemented a supplemental program which is the interface between DAMPVM and QESA\(^1\) – a project developed at the Technical University of Gdańsk. QESA is a tool for evaluating a project quality during its life cycle including all the developing phases. Data collected for many different applications let estimate the quality attributes, characteristics and metrics of various projects.

The interface between DAMPVM and QESA – “DAMPVM → QESA Interface” lets conduct experiments in the DAMPVM environment or in the UNIX operating system. It lets start and stop applications, measure their execution times, control machines on which tested applications are executed and set some environment parameters. All this information is passed then to the QESA system in a specified format. QESA reads the information and processes it.

The screenshots of “DAMPVM → QESA Interface” are presented in Figures 8.3 and 8.4.

![Screenshot of DAMPVM → QESA Interface](image)

Figure 8.3: “DAMPVM → QESA Interface” screenshot (1)

In the future, this “DAMPVM → QESA Interface” could execute a selected application automatically under different conditions. It could simulate an external load on one or more machines, change this during the application’s execution and calculate parameters itself e.g. scalability or communication costs.

\(^1\)Quality Evaluation of Software Applications
Figure 8.4: “DAMPVM → QESA Interface” screenshot (2)
Appendix A

DAMPVM Installation Manual and Programming Tutorial

A.1 DAMPVM Installation Manual

There are two software packages you need to install before installing and using DAMPVM. These are:

- PVM version 3.4+ (preferably the latest stable 3.4.0 or 3.4.1) (available from http://www.epm.ornl.gov/pvm)

- V GUI interface version 1.16+ (available from http://objectcentral.com)

After you successfully install them you should unpack the sources of DAMPVM and type `make` in the root directory of the DAMPVM sources. It will install DAMPVM with all of its options including a graphical interface developed using the V GUI interface. If you use various architectures in the virtual machine you need to compile DAMPVM for every architecture since DAMPVM works as a regular PVM application. You can also type `make nographics` instead of `make`. In this case the graphical interface of DAMPVM is not compiled.

Normally, the DAMPVM makefiles assume you install the V GUI library in the `$HOME/v` directory. In this case you do not have to change the makefiles unless the location of your X files differs from the default one. If you need to modify the makefiles, there are three of them: one in the root sources directory and then in the `interface` and `PCTextView` directories.
A.1.1 Running DAMPVM

Before running DAMPVM, add `HOME/pvm3/bin/<architecture>` to your PATH variable where `<architecture>` is/are your PVM architecture name(s) e.g. LINUX.

At first, you have to start PVM by invoking command `pvm`. After that machines to be used by DAMPVM should be added from the console using command `add <hostname>` where `<hostname>` is the name of the machine one wants to add to the virtual machine. All this is the same like in PVM. Now everything is prepared to start DAMPVM.

To run DAMPVM you need to type `dampvm`. The graphical interface will be started on the machine where `dampvm` has been invoked. There is a graphical window in DAMPVM representing each computer in the virtual machine (as shown in appendix D) but they are all started on one machine. You can easily observe the states of all the machines on these windows. If you start DAMPVM by invoking command `dampvm` it will assume that all the nodes in the virtual machine form a complete graph as far as neighbors are concerned – it means that each pair of nodes is neighbors. You can also start DAMPVM by typing `dampvm <filename>` where `<filename>` is the name of the file which defines the neighbor graph. Its structure is described below. Notice that if you use NFS there can be one file for all machines. But if there is no NFS you should prepare many configuration files (with the same location) for every machine. Then start by e.g. `dampvm $HOME/conf`. In this case every file should refer to the machine on which it is located.

A.1.2 Configuration file

The configuration file for DAMPVM defines the logical structure of the network i.e. which node is allowed to start or migrate tasks on/to which nodes. The structure of this file is as in Figure

```
# <host1_name> <host2_name> <host4_name>
# <host2_name> <host1_name> <host3_name>
# <host3_name> <host2_name> <host4_name>
# <host4_name> <host3_name> <host1_name>
```

Figure A.1: DAMVPM configuration file - a ring

Every line refers to the neighbors of the host after sign #. In particular, `<host2_name>` and `<host4_name>` are the neighbors of `<host1_name>`.
Similarly for the next lines. Notice that one can create e.g. two not connected graphs in one virtual machine.

A.2 DAMPVM Programming Tutorial

Programming in DAMPVM and using its features is easy. Although there are some rules one has to follow. They are given below:

- if not stated everything is to be like in the PVM environment

- in DAMPVM processes are identified by PCTids instead of tids like in PVM; communication functions in DAMPVM operate on PCTids instead of tids

- all DAMPVM functions start with PC_ – DAMPVM was developed using this convention and it was left it in the final version

- one should not use any PVM functions in the code while using the DAMPVM paradigm (they could interfere with each other)

- the main function should start with main(int argc, char **argv) and the process code should start with PC_Init(&argc, &argv) (there must not be any DAMPVM functions before this one)

- similarly the process code must end with PC_Finish() (this causes DAMPVM to release some resources: emptying lists etc.)

- functions PCMigrationEnable() and PCMigrationDisable() turn on and off migration possibility for the process which calls them; if migration has been enabled, DAMPVM can migrate the process to another machine and it will not if migration has been disabled

- if migration is enabled DAMPVM will call the functions PackState() and UnPackState() just before migrating the process and after migrating respectively; a programmer should pack and unpack the state of a process in these functions as well as support process recovery in the appropriate point of the code after migration (there are some functions which tell if the process has been migrated or not so they can be used in this case); appendix C contains the example of a DAMPVM application

- the code of a DAMPVM program must include #include "PC.H" and #include <iostream.h> instead of #include "pvm3.h" and #include <stdio.h>
the code of a DAMPVM program must be compiled just like a PVM program but must be additionally linked with library PC.a like in the makefile which also compiles the examples

- a special type PCTextWindow is supported for visualization what happens in a process; a process can create objects of this type which results in opening windows on certain machines (look at the example) and then send some information to these windows

- notice that processes which are to be migrated should be written in a special way so that they can start their execution from the interrupted point; look at the simple example below or the next one in appendix C; look that the process below is constructed in such a way that after migration it restarts with correct values of i and j (j can be smaller in order not to lose anything) so that the process does not have to start from the beginning; of course it may be much more complex like in the second example in appendix C

```
#include "PC.H"
#include <iostream.h>

// variables for the loops
int i=0;
int j=0;

PCTextWindow *window=new PCTextWindow; // PCTextWindow object
// for the migrated copies; it is empty (no computer name given)
// but will be prepared in function UnPackState

PackState() { // pack important variables
    // as well as the window (the 'pointer' to the window)

    PC_PkInt(&i);
    PC_PkInt(&j);
    PC_PkPCTextWindow(window);
}

UnPackState() { // unpack important variables
```
A.2. DAMPVM PROGRAMMING TUTORIAL

// as well as the window (the 'pointer' to the window)

PC_UPkInt(&i);
PC_UPkInt(&j);
PC_UPkPCTextWindow(window);

}

main(int argc,char **argv) {

PC_Init(&argc,&argv); // DAMPVM initialization (necessary!!!)

// start a new window on machine 'coper02'
// if I am not a migrated copy of this process
if (PC_How_Started()! = migrated) window = new PCTextWindow("coper02");

PCTid pctids[2]; // pctids for two children

// start two children if I am not a migrated copy and I am not a child
if ((PC_Parent()==PCNoParent) && (PC_How_Started()! = migrated)) {
    PC_Spawn("pctrivial",NULL,NULL,2,pctids,NULL);
}

PC_MigrationEnable(); // enable migration for this process

for(;i<10;i++) { // main loop - values of i and j are initialized above
    // or come from function UnPackState
    if (j == 10) j = 0; // set j to 0 if it was 10 before migration
    for(;j<10;j++) {

        // send a string to the window (it may reside on a different machine)
        (*window) << "i" << i << "j" << j << "%n";

        // delay or some work to put here
        for (int k=0;k<10000;k++)
            for(int l=0;l<1000;l++)

    }
}
}
A.3 Problems found

There are two problems detected during the experiments with DAMPVM:

- DAMPVM uses message handlers to interrupt process execution asynchronously; if a computer is heavily loaded by other processes, it takes much time for a process to handle a message which has been sent if it is executing non-PVM instructions; the solution to this problem is to put a PVM instruction which does nothing; there is \texttt{pvm\_nrecv()} used in the example in appendix C; it receives no message but this enables the process to handle messages

- there is a problem with more than about 15-20 processes on the PVM SUN4SOL2 architecture; PVM hangs up if more processes are created
Appendix B

List of DAMPVM functions

There is the list of DAMPVM functions available as a library one has to link with their programs written to use the DAMPVM mechanisms. Each of them is described in short.

- **PCTid PC_MyPCTid()** – returns the pctid of the calling process

- **void PC_Init(int *argc, char ***argv)** – the initializing function – must be called at the very beginning of a process code before using any other PC_* functions; the pointers to the variables passed to a process (main(argc, argv)) should be given as the parameters to this function

- **int PC_Spawn(char *task, char **argv, char *where, int ntask, PCTid *pctids, TaskParameters *params)** – the function which creates a new process

  - **task** – the name of the process to start (the executable file)
  - **argv** – command line arguments to the new process
  - **where** – specifies the machine where the process will be started; NULL indicates that DAMPVM should decide where to start the task (it is the recommended option)
  - **ntask** – the number of process copies to start
  - **pctids** – the table of the pctids identifiers of the started processes
  - **params** – the parameters describing new tasks (they give DAMPVM additional information about them) - they are described by the following structure:

    ```c
    typedef struct {
        long instructions;
    };
    ```
APPENDIX B. LIST OF DAMPVM FUNCTIONS

float migrationsize;
} TaskParameters;

where instructions is the number of simple instructions in the process/amount of work and migrationsize is the size of the process state - given e.g. in KBs

- int PC_Send(PCTid pctid, int msgtag) - sends the current buffer to the process identified by pctid with tag msgtag
- int PC_Recv(PCTid pctid, int msgtag) - receives (blocking) a message from the process identified by pctid with tag msgtag
- int PC_InitSend() - this function initializes sending or receiving messages (it clears the current buffer, like pvm_initsend() in PVM)
- PCTid PC_Parent() - returns the pctid of the parent of the calling process
- int operator==(PCTid pctid1, PCTid pctid2) - the operator which compares two variables PCTid; returns 1 if they are equal, 0 otherwise
- ostream& operator<<(ostream& stream, PCTid& pctid) - the operator which sends variable PCTid to the standard output
- int PC_PkInt(int *data) - the function packing variable data of type int to the active buffer, the return value less than 0 indicates an error
- int PC_UPkInt(int *data) - the function unpacking variable data of type int from the active buffer, the return value less than 0 indicates an error
- int PC_PkDouble(double *) - the function packing variable data of type double to the active buffer, the return value less than 0 indicates an error
- int PC_UPkDouble(double *) - the function unpacking variable data of type double from the active buffer, the return value less than 0 indicates an error
- int PC_PkFloat(float *) - the function packing variable data of type float to the active buffer, the return value less than 0 indicates an error
• int PC_UPkFloat(float *) – the function unpacking variable data of type float from the active buffer, the return value less than 0 indicates an error

• int PC_pkLong(long *) – the function packing variable data of type long to the active buffer, the return value less than 0 indicates an error

• int PC_UPkLong(long *) – the function unpacking variable data of type long from the active buffer, the return value less than 0 indicates an error

• int PC_pkShort(short *) – the function packing variable data of type short to the active buffer, the return value less than 0 indicates an error

• int PC_UPkShort(short *) – the function unpacking variable data of type short from the active buffer, the return value less than 0 indicates an error

• int PC_pkString(char *) – the function packing a string variable data to the active buffer, the return value less than 0 indicates an error

• int PC_UPkString(char *) – the function unpacking a string variable data from the active buffer, the return value less than 0 indicates an error

• StartWay PC_How_Started() – returns a value which indicates how the calling process has been started:
  - migrated – a migrated copy
  - spawned – started using function PC_Spawn
  - run_from_shell – started by a user from a console

• int PC:pkPCTextWindow(PCTextWindow *window) – the function packing the window of type PCTextWindow pointed by window to the active buffer, the return value less than 0 indicates an error

• int PC_UPkPCTextWindow(PCTextWindow *window) – the function unpacking the window of type PCTextWindow to window from the active buffer, the return value less than 0 indicates an error

• void PC_Finish() – the function which finishes process execution; it should be called before the end of a process code
• **void PC_MigrationEnable()** — the function which lets DAMPVM migrate the calling process (from the moment this function was called)

• **void PC_MigrationDisable()** — the function which forbids DAMPVM to migrate the calling process (from the moment this function was called)

• **void PC_Wait()** — this function informs DAMPVM that the calling process has just entered its idle state; it can be called e.g. before the **PC_Recv** operations before waiting for some results or data from other processes

• **void PC_Instructions(long instructions)** — this function informs DAMPVM that the calling process has **instructions** of instructions to finish its execution
Appendix C

Example source code

There is the code of the DAMPVM application used in the experiments with DAMPVM. See also appendix A.3 for the description of the problems encountered when working with DAMPVM.

/*
 * Author: Paweł Czarnul
 * Copyright (C) 1999 Paweł Czarnul
 * email: Paweł.Czarnul@ask.eti.pg.gda.pl
 * WWW: http://www.ask.eti.pg.gda.pl/~pczarnul
 * Last change 30.06.1999
 *
This application is to sort the vector of tabsize integer elements. The initial process gets the whole vector, divides it into two parts, creates two children and sends one copy to one child. They do the same until the size of the vector becomes smaller than tabtotosortsize. When this occurs a child sorts the vector (does not create its children) and sends the results up. Its parent merges the copies received from its children and sends the merged table up. Finally, the root obtains the sorted vector.

In the meantime DAMPVM can migrate processes. Functions PackState() and UnPackState() pack and unpack the current process state respectively. The code is written in such a way that it can recover when it was interrupted after migration occurs.

*/

int tabsize=10000;  // initial vector size
int tabtotosortsize=10000;  // maximum vector size to divide
                           // (smaller are sorted)

#include "PC.H" /* DAMPVM header file */

#include <iostream.h>
int merging=0; /* tells if merging was interrupted */

int i=0; /* main loop variables */
int j=0;

int imigrating; /* loop index at the migration moment */
int k; /* additional variable */

int element1,element2; /* variables for exchanging values during sorting */
int section=0;

int *tab,*tabp; /* two tables with sorted integers,
there are two of them to support easy recovery after migration */

int packing=0; /* defines which table (tab, tabp)
contains true values */

/* a graphical window for the root process */
PCTextWindow *window=new PCTextWindow;

PackState() { /* this function packs the current state */
PC_PkPCTextWindsw(window);

PC_PkInt(&merging);
PC_PkInt(&tabsize); /* the size of the current table */

if (!merging) { /* tells that not merging was interrupted */
  if (packing) { /* defines which table (tab, tabp)
    contains true values */
    for(k=0;k<tabsize;k++) PC_PkInt(&tab[k]);
  } else {
    for(k=0;k<tabsize;k++) PC_PkInt(&tabp[k]);
  }
} else {
  for(k=0;k<tabsize;k++) PC_PkInt(&tabp[k]);
}

PC_PkInt(&imigrating);
}

UnPackState() { /* this function unpacks the current state */
PC_UPkPCTextWindsw(window);

PC_UPkInt(&merging);
PC_UPkInt(&tabsize);

/* now new tables must be created */
int *tab=new int [tabsize];
if (!tab) if (PC_Parent()==PC_NoParent) (*window) << "\nNot enough memory";
tabp=new int [tabsize];
if (!tabp) if (PC_Parent()==PC_NoParent) (*window) << "\nNot enough memory";
/* unpack data integers to the table */
if (!merging) {
    for (k = 0; k < tabsize; k++) PC_UPkInt(&tab[k]);
} else {
    for (k = 0; k < tabsize; k++) PC_UPkInt(&tabp[k]);
}

main(int argc, char **argv) {
    PC_Init(argc, argv);
    if ((PC_How_Started() != migrated) && (PC_Parent() == PCNoParent) && (argc != 2)) {
        cout << "psort: too few parameters..." << endl;
        cout << "Syntax: " << endl << "psort <interface host name>" << endl;
        PC_Finish();
        exit(0);
    }

    if ((PC_How_Started() != migrated) && (PC_Parent() == PCNoParent)) window = new PCTextWindow(argv[1]);

    // start a graphical interface for the root process */
    PCTid pctsids[10];
    PC_MigrationDisable();

    if (PC_How_Started() != migrated) { /* this is executed not for a migrated process */
        if (PC_Parent() == PCNoParent) { /* the root process */
            if (PC_Parent() == PCNoParent) (*window) << "Root";
        }

        /* create tables */
        tab = new int [tabsize];
        if (!tab) if (PC_Parent() == PCNoParent) (*window) << "Not enough memory";
        tab = new int [tabsize];
        if (!tab) if (PC_Parent() == PCNoParent) (*window) << "Not enough memory";

        /* initialize tables */
        for (i = 0; i < tabsize; i++) tab[i] = tabsize - i;
    } else { /* children processes */
        if (PC_Parent() == PCNoParent) (*window) << "Child";
        PCTid mypctid = PC_MyPCTid();

        /* wait for elements to sort */
        PCTid parentpctid = PC_Parent();
        PC_Recv(parentpctid, -1);
        PC_UPkInt(&tabsize); /* unpack the number of elements */
        tab = new int [tabsize];
        if (!tab) if (PC_Parent() == PCNoParent) (*window) << "Not enough memory";
        tab = new int [tabsize];
if (!tabp) if (PC_Parent() == PCNoParent) (*window) << "\nNot enough memory\n";
if (PC_Parent() == PCNoParent) (*window) << "\nInTabsize: " << tabsize;
for (i=0; i<tabsize; i++) PC_UFPInt(&tab[i]); /* unpack the elements */
}
*/
/* now divide or sort */
if (tabsize > tabsortsize) { /* divide */
int dividepoint = tabsize / 2;
if (!merging) {

TaskParameters params[2];
/* initialize children processes parameters */
params[0].migrationsize = tabsize / 8 / 1000; /* initial migration sizes for the children */
params[1].migrationsize = tabsize / 8 / 1000;

/* initial instruction values for processes */
if ((tabsize / 2) > tabsortsize) {
params[0].instructions = 10;
params[1].instructions = 10;
} else {
params[0].instructions = 1000;
params[1].instructions = 1000;
}

PC_Spawn("p:sort", NULL, NULL, 2, pctids, params); /* start the children */

/* now send data to the children */
int size;

size = dividepoint;
PC_InitSend();
PC_FkInt(&size);
for (i=0; i<dividepoint; i++) PC_FkInt(&tab[i]);
PC_Send(pctids[0], 1);

size = tabsize - dividepoint;
PC_InitSend();
PC_FkInt(&size);
for (i=dividepoint; i<tabsize; i++) PC_FkInt(&tab[i]);
PC_Send(pctids[1], 1);

PC_Wait(); /* now this process has to wait for the results
- it will take much time --
it is worth informing DAMPVM about it */

/* wait for sorted tabs */
PC_Rcv(pctids[0], -1);
for (i=0; i<dividepoint; i++) PC_UFPInt(&tabp[i]);
### Code Snippet

```c
PC_Recv(pktids[1], 1);
for (i = dividepoint; i < tabsize; i++) PC_UPkInt(&tabp[i]);

if (PC_Parent() == PCnParent) (*window) << "Get sorted tabs";
}

merging=1; /* merging section */
PC_Instructions(1000/*tabsize*/);

PC_MigrationEnable();

/* merge the tabs */
for (k = 0, i = 0, j = dividepoint; k < tabsize; k++) {
    if ((i == dividepoint) && (j < tabsize)) {
        if (tabp[i] > tabp[j]) {
            tab[k] = tabp[i];
            if (i == dividepoint) i++;
        } else {
            tab[k] = tabp[j];
            if (j < tabsize) j++;
        }
    } else {
        if (i == dividepoint) tab[k] = tabp[j++];
        if (j == tabsize) tab[k] = tabp[i++];
    }
}

if (PC_Parent() == PCnParent) (*window) << "\Tabs merged";

PC_MigrationDisable();
/* send back the result to the parent */
if (!PC_Parent() == PCnParent) {
    PC_InitSend();
    for (i = 0; i < tabsize; i++) PC_PkInt(&tab[i]);
    PC_Send(PC_Parent(), 1);
} else {
    if (PC_Parent() == PCnParent) (*window) << "Sorted";
}
}

else { /* sort the table */
    for (k = 0; k < tabsize; k++) tabp[k] = tab[k];
    imigrating = 0;

    if (PC_How_Started() == migrating) PC_MigrationEnable();
    if (PC_How_Started() != migrating) i = 0;
    for (; i < tabsize; i++) {
        if (!(i % 100)) {
            if (PC_Parent() == PCnParent) (*window) << "\n" << i;
        }
    }

    pcm_nrecv(100, 100); /* supports signal handling immediately by PVMM - there is a big delay without this*/
}
```

### Notes

This code snippet is likely part of a larger program dealing with network communication and data processing. It appears to be a part of a network packet processing routine, handling packet IDs and sending packets using a PCUP protocol. The code includes logic for merging tabs, sending data back to the parent, and handling migrating data. It also includes comments and functions related to packet communication and handling.
if (i==200) PC_MigrationEnable(); // enable migration after
// some time - not during starts of other processes
if ((i>=(tabsize-500)) && (i<(tabsize-200))) PC_MigrationDisable(); /* disable migration
before all the processes end */
}

for(j=1;j<tabsize;j++)
if (tab[j-1]>tab[j]) { /* invert the elements */
  element1=tab[j-1];
  tab[j-1]=tab[j];
  tab[j]=element1;
}
/* send back the result */
PC_InitSend();
for(i=0;i<tabsize;i++) PC_PkInt(&tab[i]);
PC_Send(PC_Parent(),i);
}
PC_Finish();
}
Appendix D

DAMPVM console screenshot

There is a screenshot of the DAMPVM interface presented in Figure D.1. Each node has its own graphical console one can observe its state on.

There are two graph categories one can choose in the current version of DAMPVM (1.0). These are:

- CPU loads – the total load and the load by other users, so the difference is the load by DAMPVM processes
- the current number of DAMPVM processes

These graphs are self-scalable and they scroll automatically every second (this is the default step). A few graph options are available from the menu. Apart from the above graphs there is the lower window where information about activities on the node is displayed. This includes a new process started on the node, process termination or the migration of a process from the node to another one.
Figure D.1: DAMVPM screenshot
Bibliography


[32] Ohio Supercomputer Center, The Ohio State University. MPI Primer/Developing with LAM, November 1996.

[33] Chao-Wei Ou and Sanjay Ranka. Parallel Incremental Graph Partitioning. IEEE Transactions on Parallel and Distributed Systems, 8(8), August 1997.


BIBLIOGRAPHY


List of Figures

2.1 An exemplary application graph .......................... 11
2.2 An exemplary system graph ............................. 12
2.3 The Amdahl law ........................................... 16
3.1 CORBA architecture ...................................... 23
3.2 Intermachine object referencing ........................ 24
3.3 Communication vs migration in MPVM ............... 26
4.1 Network computing environment ...................... 30
4.2 Possible node load ....................................... 32
4.3 Creation – dynamic tree structure of application A ... 34
4.4 Definition of the instr and eslt values ............... 35
4.5 Example of instrumentation of application A ........ 37
5.1 Mutual relations among system elements .............. 40
5.2 Incoherent neighbor relations .......................... 41
5.3 DAMPVM structure ....................................... 42
5.4 Mutual relations among system elements (2) .......... 43
5.5 Algorithm concept ........................................ 44
5.6 Spawning – function Spawn() ......................... 46
5.7 Migration – function Schedule() ...................... 47
5.8 DAMPVM migration criterion ............................ 48
5.9 Migration – function Schedule() – a locking version ... 49
5.10 Migration and allocation algorithm for node $N_i$ .... 50
5.11 Overloading a node ...................................... 51
6.1 Process creation .......................................... 57
6.2 Process migration ......................................... 58
6.3 function PC_Send() ....................................... 59
6.4 function PC_Recv() ....................................... 60
6.5 Bottleneck links .......................................... 61
6.6 Process migration with a mailbox ....................... 62
115
6.7 Dividing a process into smaller ones .......................... 63
6.8 Initial process (left) and two processes derived from the initial
one (right) .......................................................... 64
6.9 PCTextWindow ..................................................... 66
7.1 Optimal assignments and migration adapts the system to other
user activities ......................................................... 71
7.2 Idle time in spite of an optimal assignment ...................... 72
7.3 Application graph .................................................. 73
7.4 Instrumentation of the application used in the experiments:
Nodes: a. leaves b. non-leaves ................................. 73
7.5 Speed-up - initial vector size=400000, sorted vector size=20000 78
7.6 Execution times - initial vector size=400000, sorted vector
size=20000 .......................................................... 78
7.7 Speed-up - initial vector size=100000, sorted vector size=10000 79
7.8 Execution times - initial vector size=100000, sorted vector
size=10000 .......................................................... 79
7.9 Speed-up - initial vector size=200000, sorted vector size=10000 80
7.10 Execution times - initial vector size=200000, sorted vector
size=10000 .......................................................... 80
7.11 a. Static assignment only b. Static assignment + migration .. 81
8.1 Relative loss without migration .................................. 86
8.2 Relative loss without migration - initial vector size=400000,
sorted vector size=20000 ............................................ 86
8.3 "DAMPVM → QESA Interface" screenshot (1) ................. 89
8.4 "DAMPVM → QESA Interface" screenshot (2) ................. 90
A.1 DAMVPM configuration file - a ring .......................... 92
D.1 DAMVPM screenshot ............................................. 108
## List of Tables

3.1 PVM-related systems comparison .................................................... 28

7.1 Experiments results - initial vector size=400000, sorted vector size=20000 .............................................................. 74
7.2 Experiments results - initial vector size=500000, sorted vector size=10000 ................................................................. 75
7.3 Experiments results - initial vector size=1000000, sorted vector size=10000 ................................................................. 75
7.4 Experiments results - initial vector size=2000000, sorted vector size=10000 ................................................................. 76
7.5 Exemplary assignments ................................................................. 81
Index

algorithm
  complexity, 6
DAMPVM
  allocation, 43
  concept, 44
  migration, 43
  pseudocode, 46
  reassignment initiation, 44
  reassignment rules, 45
  genetic, 6
  gradient, 6, 45
  heuristic, 5, 6
  kind
    diffusion, 7
    dimension exchange, 7
  nearest neighbor, 6, 7
allocation, 2, 3, 15
  Condor, 22
  DAMPVM, 43
    algorithm, 43
  MPI, 21
  MPVM, 25
  of a task, 9, 13
    definition, 10
  of data, 5
  other systems, 19
  PVM, 20, 29
architecture, 1, 60
  Common Object Request Broker Architecture, 23
  of CORBA, 23
  of DAMPVM, 39
  PVM, 21
architecture-dependent mechanisms, 18
assignment, 6, 7, 12, 29, 37, 44, 45, 69, 70, 86
DAMPVM
  initiation, 44
  rules, 45
  definition, 10
dynamic, 3, 5, 6, 38, 77
  definition, 10
  subclasses, 10
  of an application to a system, 9, 11
  optimal, 70, 71
static, 4, 5, 79
  definition, 10
  vs decomposition, 10
bottleneck link, 61
Chorus, 17
  Condor, 3, 8, 18, 22, 34, 57, 88
  allocation, 22
  vs MPVM, 25
CORBA, 23
  architecture, 23
  migration, 24
    cost, 25
  object referencing, 24
DAMPVM, 7, 18, 29, 36, 37, 71, 87, 107
  algorithm, 38
  allocation, 43
INDEX

- concept, 44
- migration, 43
- pseudocode, 46
- reassignment initiation, 44
- reassignment rules, 45
- architecture & implementation, 39
- disadvantages
  - process flickering, 85
- extensions, 88
  - QESA interface, 89
- goals, 36
- library, 66, 74
- limitations, 37, 61
- load balancing, 41
- migration, 70, 84
- neighbors
  - configuration file, 31
  - requirements, 37
  - implementation, 38
- scalability, 88
- structure, 39, 41, 42
  - vs operating system, 39
  - vs PVM, 39, 40
- working modes, 38, 87
- decomposition, 7, 16
  - of a problem (definition), 9
  - vs assignment, 10
- definition
  - assignment of an application to a system, 9
  - communication link, 9
  - computer system, 9
  - parallelization, 9
  - problem decomposition, 9
  - process, 9
  - process migration, 9
  - task, 9
- definitions, 9
- DPVM, 18, 59
- example
  - of a DAMPVM application, 101
- experiments, 69
  - conclusions, 87
  - definitions, 77
  - description, 72
  - discussion, 83
  - environment, 73
  - results, 77
- function
  - PC_Recv(), 60
  - PC_Send(), 59
- identification
  - of a process, 3, 39
    - DAMPVM, 37, 53, 54
    - vs migration, 53
- IDL, 24
- idle time, 80
- implementation, 17
  - Condor, 22
  - CORBA
    - of an object, 23
    - repository, 24, 25
    - DAMPVM, 39
    - communication functions, 57
    - requirements, 38
  - of MPI, 22
  - LAM, 22
  - MPICH, 22
- Incoherent neighbor relations, 41
- installation
  - DAMPVM, 91
  - manual, 91
- instrumentation, 36
  - example, 37, 73
- limitations, 3, 33
  - DAMPVM, 37
  - migration in CORBA, 24
Mach, 18
message passing paradigm, 13
migration, 3, 6, 14, 29, 34, 39, 44, 69–72, 80, 84, 85
CORBA, 24
costs, 25
of objects, 24
cost, 37
criteria, 46
DAMPVM, 33
algorithm, 43, 45
functions, 62, 88
heterogeneity, 59
in a process, 57
interaction diagram, 58
interaction diagram with a mailbox, 62
limitations, 60
process size, 33
topology, 31, 40
transparency, 57
vs other systems, 3
vs visualization, 66
definition, 9
experiments, 77
conclusions, 79
cost, 77
heterogeneous, 2, 18, 26
homogeneous only, 18
increasing & decreasing, 47
MPVM, 25
vs communication, 26
process identification, 38, 53
strategies, 17
vs data redistribution, 6
vs granularity, 79
vs scalability, 83
MIST, 18
model, 5, 13, 88
classic, 5
Extended Gradient Model, 7
of a computing environment, 30
of a program, 29, 53
of an application, 33
of processing, 30
probabilistic, 6, 45
PVM, 30
MPI, 21
implementations, 22
LAM, 22
MPICH, 22
process identifier, 53, 54
vs PVM, 21
MPI-2, 21
MnPVM, 18, 26
MPVM, 4, 8, 18, 25, 34, 54, 57, 60
communication vs migration, 26
vs Condor, 25
NFS, 66
node
overloading, 51
OMG, 24
ORB, 24
parameter, 12, 69, 72
MPI_Spawn, 22
context, 55
DAMPVM, 34–36, 46
changes, 44
migration, 33
of a node, 32, 34
of a process, 34
of a shell command, 19
of a system, 5
processor, 1
PCTextWindow, 66
preemptive tasks, 70
programming
in DAMPVM, 91
tutorial, 93
pseudocode
INDEX

DAMPVM algorithm, 46
  function Schedule(), 47
  function Spawn(), 46
locking function Schedule(),
  49
main, 50
PVM, 3, 6, 20, 29, 70
  DAMPVM requirement, 38
extension, 4
  DAMPVM, 29, 38
  DAMPVM library, 37
MPVM, 25
mechanisms used in DAMPVM,
  54
message routing, 12
processes, 53
tid, 53, 55
vs DAMPVM, 39, 43

QESA, 89
  DAMPVM interface, 89
reallocation, 14, 29
reassignment, 14, 29
  DAMPVM
    initiation, 44
    rules, 45
requirement
  Condor, 22
  DAMPVM, 4, 37
    implementation, 38
results, 77
  discussion, 83
scalability, 3, 7, 47, 61, 70, 71, 83,
  86–88
  in CORBA, 24, 25
speed-up, 15, 17, 30, 72, 84
  figure, 78–80
  maximum, 17
  optimal, 72
Sprite, 17
tendencies, 87
UNIX, 19, 89
  ps command, 32
  process, 23
V (library), 38
visualization
  DAMPVM
    data in a process, 66