Dynamic Assignment with Process Migration in Distributed Environments *

Pawel Czarnul and Henryk Krawczyk
Dept. of Computer Systems Architecture
Technical University of Gdańsk
Pawel.Czarnul@ask.eti.pg.gda.pl, hkrawk@pg.gda.pl

Abstract This paper considers dynamic assignment of application processes to processing nodes. It is assumed that both the application and the system parameters can change dynamically. Monitoring the environment and instrumentation of an application code are used to support the prediction of these parameters. The extension of the PVM environment called DAMPVM\(^1\) is proposed which supports both dynamic process allocation and heterogeneous migration. Its suitability for efficient processing of distributed applications is evaluated.

1 Introduction

Computations executed in distributed environments require a good assignment of application components (processes) to processing nodes. Concrete assignment techniques depend on the type (architecture, granularity) of user applications, the configuration (structure) of a distributed or parallel system and the parameters of processing. These parameters include the costs of computations performed in each node, the cost of communication between each pair of nodes and the cost of routing information in the system ([3]). Taking all this into account we try to organize the computations in such a way that minimum execution time of the application is achieved.

In general, there are two basic assignment approaches: the static one ([1]) which assumes that both the architecture of an application and the computing system structure do not change in time and the dynamic one where application processes or system processes can be created or deleted in the run-time. To prepare and evaluate the latter approaches simulation and modeling are largely used ([6]). Various dynamic strategies are considered in literature. In paper [8] it is assumed that from time to time some new processes of the program are added (created) or some of them must be deleted (are completed). The correction of the previous assignment must be performed then. The loads of all the processing nodes should remain approximately the same while the communication costs between the nodes are minimized. This strategy is achieved using the

---

* Supported in part by EU under INCO-Esprit 977100 and by the State Committee for Scientific Research (KBN) under grant 8 T11C 043 12.

\(^1\) Dynamic Allocation and Migration PVM (www.ask.eti.pg.gda.pl/~pczarnul).
linear programming method. An alternative approach is presented in [2]. The authors focus on data redistribution instead of process migration. On the other hand, there are so-called neighbor algorithms which try to reach a global, good assignment by successive exchange of load (process migration rather than data redistribution) among neighbors. A few classes of such algorithms are distinguished and compared in [10]. Gradient methods have been described in [4] (a basic version) and [5] which is the study of various versions of such strategies.

We propose the DAMPVM environment which consists of kernels, each with its own graphical interface window, running on each machine and monitoring node parameters and application processes running on it, exchanging information between each other and performing dynamic allocation of processes and process migration, if necessary. Moreover, the library of functions analogous to PVM ones supports communication, creation and process instrumentation. Migration is heterogeneous since it is based on two functions for each process: packing and unpacking the current state (the values of important variables, tables etc. instead of machine registers or stack segments). A programmer should write the functions and the code in such a way so that the process can start from the interrupted point after migration. The disadvantage of this method is a bigger programming effort. In return we get homogeneous and fast migration which is not available in many other systems. A multi-user environment is also assumed where other users can run their own tasks at the same time without any limitations. DAMPVM detects the presence of other users and the loads and speeds of the nodes as well as predicts the forward work for processes to achieve minimum execution time of an application (for the user who uses it). DAMPVM design is based on the new mechanisms of the latest PVM 3.4. Experimental results confirm the usefulness of the proposed solution for the PVM environment.

2 Model of Network Processing

2.1 Computing Environment Model

PVM model, while being very general, does not take into account other users executing their processes on the same machines at the same time. Let a distributed computing environment be described by graph \( G_s(N, N_b) \), where \( N \) is the set of processing nodes \( N = \{N_1, N_2, \ldots, N_n\} \) and \( N_b \) describes the neighbor relation between node \( N_i \) and \( N_j \) (defined below).

Node Characteristics. Let us consider the execution of application \( A \). Then we distinguish two classes of processes running on each system node - processes of application \( A \) and others as illustrated in Fig. 1. The percentages of a processor captured by other users processes and system processes are monitored to determine the fraction of a processor available for application \( A \). Each node is characterized by the same parameters (variable in time) which values can be determined by a suitable system function or estimated. They are as follows:

1. \( s_{pj} \), the speed of node \( N_j \) defined as the inverse of time needed to perform a selected set of instructions; for our consideration this set should contain the instructions used in application \( A \).
2. $\text{othL}_i(t)$ 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 UNIX `ps` command

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

4. $NP(i, t)$ the set of new processes of application $A$ starting on node $N_i$ at moment $t$

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

**Communication Costs.** They are defined by matrix $\Gamma(t) = [\gamma_{pq}(t)]_{n\times n}$. $\gamma_{pq}(t)$ is the cost of sending a unit of information via the physical communication link from node $N_p$ to node $N_q$ at moment $t$. The way of estimation these values is given in [7]. DAMPVM estimates $\Gamma(t)$ by sending messages between nodes.

**Neighbor Relations.** This was introduced to limit interprocessor communication. If node $N_i$ is a neighbor of node $N_j$ it means that node $N_i$ can locate a new process on node $N_j$ (if it is needed) and node $N_j$ can migrate the processes running on it to node $N_i$. These possibilities are determined by matrix $\text{Nh} = [n_{pq}]_{n\times n}$

where:

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

This matrix does not have to be symmetrical. It can be defined by the user of DAMPVM by a configuration file. It means that the topology of spawning and migration is flexible, determined in advance for an application and fixed during its execution.

### 2.2 Application Model

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

**Dynamic Creation Tree.** Processes of application $A$ can be created any time without any limitations, i.e. new processes can be created by the processes which have already been started before. As a consequence we obtain a dynamic tree in which the root process is the initial one and it creates its children, they do the same and so on. An exemplary tree is shown in Fig. 2.

**Migration.** A process of application $A$ can be migrated from node to node. In this case its execution is interrupted, its state is packed and sent to another node, a new copy of the process is started on this node, this copy enters the last
state of its predecessor and carries on the execution. We introduce one parameter connected with migration: $PS_{ij}(t)$ the size of the state of the $j$-th process on node $N_i$ at moment $t$. The state includes all the variables and data necessary to execute the process from the interrupted point, expressed in kilobytes.

**Forward Execution Time Prediction and Estimation.** To allocate application processes to system nodes additional information is used. It refers to the prediction of forward work for a process and is represented by parameter $instr_{ij}(t)$. This means the estimated number of elementary instructions of the $j$-th process on node $N_i$ (which belongs to application $A$) from moment $t$ until the process finishes. This parameter can represent the number of instructions, the number of loops in the process, function calls or the value representing the remained work for the process and therefore this can be often rough estimation. The same as with node parameters $instr$ and $PS$ can change in time.

Let us consider the relationship between $instr_{ij}(t)$ and $est_{ij}(t)$. They refer to estimated values of work required for the $j$-th process on node $N_i$ and the execution time for node $N_i$ (from moment $t$) respectively. Then:

$$\forall i \in \{1, 2, \ldots, n\} \quad est_{ij}(t) = \frac{1}{sp_i(1 - \frac{a_b L_i(t)}{100})} \sum_{j=1}^{\lvert CP_i(t) \rvert} instr_{ij}(t) \quad (1)$$

We distinguish two different states of each process as follows: **active** performing computations and **waiting** waiting for the results of computations from other processes or for access to some resources. When the $j$-th process on node $N_i$ enters a waiting state it can inform DAMPVM about it by calling a special library function. When DAMPVM receives this message it sets $instr_{ij}(t) = 0$ for the $j$-th process on node $N_i$ temporarily. It changes $est_{ij}(t)$ and enables other processes from the other nodes migration to node $N_i$.

**Instrumentation of Application $A$**. The estimation of $instr_{ij}(t)$ can be easily solved by putting a special function in appropriate points of a process code.

1. Set $instr = 0$ before calling receive functions (they can take much time)
2. Set $instr$ to appropriate values after conditional instructions

This informs DAMPVM about the approximate amount of work to be carried out by a process as well as its state. Allocation of this extra instructions should be done by the designer of an application who is the best responsible person for such estimation. The examples of this are shown in Fig. 3.

### 3 DAMPVM Tool

DAMPVM implements the models presented above with the described mechanisms. The goal of DAMPVM is to minimize the total execution time of appli-
cation $A$. This is done by minimizing the following value:

$$\max_{i \in \{1, 2, \ldots, n\}} \{est(t)\}$$

which is performed by balancing the estimated processing times of the nodes known at the moment ($est(t)$). This results in minimizing the total execution time of application $A$. If external conditions change or any of $instr_{ij}$ changes or new tasks appear (new set $NP(i, t)$) DAMPVM tries to balance the work to achieve good assignment (the correction of the current assignment is performed). Similarly to [4] processes are transferred from more loaded machines (higher $est$) to their lightly loaded (lower $est$ value) neighbors. If the current state of the system ($instr$ and $othL_i(t)$ which result in $est$) is constant then no improvements are done. Fig. 4 illustrates the assignment strategy supported by DAMPVM. DAMPVM is event-driven which means that for node $N_i$ two classes of events can cause changes of the current assignment (see Fig. 4):

1. new $spawn$ calls on node $N_i$ or its neighbors only assignment decisions are taken and new tasks are located on nodes
2. significant changes of $instr_{ij}(t)$ or $othL_i(t)$ parameters for node $N_i$ or its neighbors then suitable migration decisions are taken and performed

The above tasks are performed asynchronously for each node. $spawn$ messages can come anytime and anywhere (to different nodes).

Figure 6 shows the DAMPVM console of a node the current state of a node in the system i.e. the current number of application processes and the load. In DAMPVM there is such a console for each node.

4 Experiments

4.1 Experiments Environment and Application

Environment. The LAN network of up to 7 processors was used. The following computers were successively attached to the system: copere01, copere02, copere03, nicel0, nicel13, nicel14, jupiter (@st.pi.gda.pl). Their relative speeds ($sp_i$) for the application described below are: 20.8, 20.8, 20.8, 16.4, 16.4, 16.4 and 8.3 respectively. The network is heterogeneous since copere is PCs with Linux, nicel is Sun SparcStations with Solaris and jupiter is the HP with HP-UX. It was $nb_{pq} = 1 \forall p, q \in \{1, 2, \ldots, n\}$ and $p \neq q$.

Application. In order to check the performance of the proposed solution we
Figure 4. The concept of DAMPVM

wrote an exemplary application using the DAMPVM library. The creation tree is a binary tree of depth 6 like in Fig. 2 but it is balanced. The application is to sort the vector of 40000 integer elements. It uses the bubble sort method to achieve this. The initial process gets the whole 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 elements. The leaves of the tree sort their own vectors and then send the sorted ones to their parents. They merge the vectors and send them up. Finally the root merges and the whole application exits. It means that when the leaves sort, the other processes must wait for them. Instrumentation and state detection was used as shown in Fig. 3 the leaves (a) and the other nodes (b). There were 63 processes (32 sorting). It must be stated that migration costs were low comparing to the computation times of the processes (one migration took a few seconds).

4.2 Experiments Definitions and Results

Definitions. There were three experiments performed corresponding to the situations shown in Fig. 5. At first the application was executed in the system without an external load (except small X Window overhead) with migration enabled. In the second and third experiments another user started their 5 processes (the external load) on coper01 50 seconds after the start of the application (all its processes had already started). There was no migration in the second experiment and migration was enabled in the third one.

Results. The reproducible results are shown in Fig. 7. Without migration although the work was balanced by the dynamic assignment, processes had to stay on coper01 with the external load. In the migration case, some of them were moved to other nodes. One can notice that the more processes the fewer
processes of the tested application were on copper01. These experiments prove that in very changeable environments (considerable changes of external load—a big amplitude) migration is very useful. Generally, in static or nearly static environments (only slight changes of external load—a low amplitude) dynamic assignment would do. In case of slight changes of an external load with a small amplitude moving processes gives nothing but it is itself some overhead.

DAMPVM solves also two other problems. Migration can divide processes into smaller parts and improve scalability. It can be useful in situations when the number of processes is comparable to the number of nodes. Also, if the work of an application seems to be balanced the execution times on separate nodes need not to be balanced. This is because of precedence relations between processes. DAMPVM processes can dynamically show in which state they are and migration balances work. The instrumentation of processes removes idle time in spite of the "optimal" assignment.

5 Final Remarks

The experiments showed that the implemented mechanisms are useful i.e. give very good scalability of an application. DAMPVM has been tested intensively for various dynamic applications. The proposed solution can be mostly recommended for time-consuming computational tasks. DAMPVM supports high scalability and significant reduction of execution time thanks to heterogeneous migration. We can also regulate the cost of migration which allows adapting the migration strategy to concrete changes of a computing environment i.e. to decrease the migration occurrence if there are short and big changes of the loads.
of nodes. Instrumentation and matrix $\mathbf{N}b$ can be used for testing i.e. for using various code instrumentation and dividing the network into parts. The future work includes building an extra tool which would support automatic process instrumentation by examining its code.

References