Efficient Process Migration for Parallel Processing on Non-Dedicated Networks of Workstations

Kasidit Chanchio *Xian-He Sun**

Department of Computer Science
Louisiana State University
Baton Rouge, LA 70803-4020

Abstract

This paper presents the design and preliminary implementation of MpPVM, a software system that supports process migration for PVM application programs in a non-dedicated heterogeneous computing environment. New concepts of *migration point* as well as *migration point analysis* and *necessary data analysis* are introduced. In MpPVM, process migrations occur only at previously inserted migration points. Migration point analysis determines appropriate locations to insert migration points; whereas, necessary data analysis provides a minimum set of variables to be transferred at each migration point. A new methodology to perform reliable point-to-point data communications in a migration environment is also discussed. Finally, a preliminary implementation of MpPVM and its experimental results are presented, showing the correctness and promising performance of our process migration mechanism in a scalable non-dedicated heterogeneous computing environment. While MpPVM is developed on top of PVM, the process migration methodology introduced in this study is general and can be applied to any distributed software environment.

*This research was supported in part by the National Aeronautics and Space Administration under NASA contract No. NAS1-19480 while the second author was in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-0001, and by NASA contract No. NAS1-1672 and Louisiana Education Quality Support Fund.

1 Introduction

The construction of national high-speed communication networks, or the so-called "information highway", marks the beginning of a new era of computing in which communication plays a major role. Until recently, computing resources on networks remained separate units. Now, while the World Wide Web and electronic mail are changing the way people do business, heterogeneous networks of computers are becoming commonplace in high-performance computing. Several commercial and non-commercial software systems, such as *Express*, MPI, the *P4* system, and PVM [1, 2, 3], have been developed to support distributed computing. However, given the current success of these software systems, experiences have shown that process migration is essential for achieving guaranteed performance in a non-dedicated distributed computing environment and for establishing the standard for mainstream parallel and distributed computing.

In a non-dedicated environment, computers are privately owned. Individual owners do not want to see their systems being saturated by others when they need them. To have a motivation to participate in a network, the individual owners need their resources readily available when they are trying to work. This means the privately owned machines may only be used for parallel processing on an "availability" basis. The uncertainty of the "availability" of an individual machine makes the performance of each individual machine non-deterministic. In addition, a non-dedicated network of computers is more likely to be heterogeneous than is a dedicated system. "Availability" and "heterogeneity" are new issues which do not arise in tightly coupled parallel systems. Competition for computing resources does not lead to guaranteed high performance. "Stealing" of computing cycles, instead of competing for computing cycles, is a more reasonable way to achieve parallel processing in a non-dedicated parallel and distributed environment. The process migration mechanism is proposed as a solution to simultaneously utilizing idle machines and maintaining high capabilities for local computations. The simple idea underlying this mechanism is that when the workload of a distributed system becomes imbalanced, parallel processes residing on overloaded machines are migrated to other available machines.

PVM [1] is a popular software system that lets programmers utilize computing resources in a distributed environment. It allows application developers to create virtual machines, and to run their parallel processes (or tasks) over a network of computers. Since the virtual machines can be created on various parallel platforms, application developers can also exploit heterogeneous resources available in a virtual computing environment. The MpPVM (Migration-point based PVM) software system proposed in this study is designed to support process migration in a non-dedicated, heterogeneous computing environment. Along with the migration mechanism, we present novel ideas of *migration point* and *necessary data analysis*. *Migration point analysis* is how MpPVM inserts its migration points to the PVM source file; whereas, *necessary data analysis* is a methodology to reduce the size of data that must be transferred during the migration. We also propose a methodology to perform reliable indirect communications among processes in a migration environment. Experimental results are presented to verify the applicability of the design and to demonstrate the performance gain of process migration. Measured results show that under the MpPVM system the

process migration cost is small compared with computing and communication cost; the performance gain of process migration is significant; and the MpPVM system is scalable in the sense that the migration cost does not increase with the ensemble size. While the experimental results are only for the examined application and environment, they demonstrate the feasibility and high potential of process migration in a non-dedicated heterogeneous environment.

This paper is organized as follows. Background and related work is presented in Section 2. An overview of the structure of MpPVM is introduced in Section 3. In Section 4, the design and functionalities of important components of MpPVM, including the precompiler, migration point analysis, necessary data analysis, migration macros and variables, and the interactions of these macros during process migration are described. In Section 5, the interaction between process migration and resource management, along with a methodology to handle data communication among processes in a migration environment, are discussed. Preliminary implementations and experimental results are given in Section 6. Finally, Section 7 gives a conclusion of current results and a discussion of future work.

2 Related Work

Despite the fact that the importance of load balance in a non-dedicated environment has been recognized, none of the existing distributed software systems support efficient process migration in a heterogeneous environment. Many attempts such as Condor, MPVM, ULP and MIST [4, 5, 6] have been made recently to support process migration. These current attempts are for homogeneous computing only, which severely limits their applicability in an actual environment.

2.1 Background

There are several software systems available to support parallel processing in a distributed environment [1]. Among these, MPI [3] and PVM [1] are the most popular software environments. The migration approach proposed in MpPVM can be applied to any existing message passing software. We chose to implement the migration mechanism on PVM instead of MPI because of the following reasons:

- PVM has a single implementation and its structure is well-defined. On the other hand, MPI has many different implementations such as LAM [7], MPICH, and UNIFY [8]. Since our work is specifically dependent on implementation, supporting process migration in MPI application codes either would limit our effort to a smaller group of users, if migration enhancement is limited to a particular implementation of MPI, or would be exhausting, if all the different implementations have to attacked for heterogeneous computing.

- Some MPI implementations are built on top of PVM, such as MPICH/PVM, or integrated into PVM such as UNIFY. Thus, if we build our prototype on PVM, the idea and implementation can be extended to MPI systems as well.

- The nature of MPI processes is relatively static compared to those of PVM. Current version of MPI does not allow process creation at run-time and has static group membership. Processes in MpPVM can be dynamically created, terminated, or migrated from one machine to another. Therefore, it is more natural to incorporate our ideas into PVM than MPI.

2.2 Transparency and Heterogeneity

Unlike MPVM and ULP [5] developed at Oregon Institute of Science & Technology, migration in MpPVM will be performed at *Migration Points (Mp)* only. These *Mp*s will be inserted automatically by the precompiler. Users can also specify their migration points. In the latter case, the precompiler will adjust the arrangement of its *Mp*s to fit the user's needs and the needs of the underlying computing environment.

Process migration is transparent to users of MpPVM. The precompiler will modify the PVM source code to support the migration mechanism. All necessary definitions and operations are inserted into the input file by the precompiler and require no work from the user. With the use of this precompiler, PVM's applications are able to run on MpPVM as well. Note that MpPVM also allows users to customize the automatically generated migration operations in their application programs to improve the performance of their parallel computations. With the awareness of process migration during parallel execution, the users can help the precompiler select the migration points which are most suitable to the structure of their application.

MpPVM supports process migration in a heterogeneous computing environment. The precompiler will add migration definitions to its input source program. When a migration event occurs, a process will perform migration operations at a high level by executing a set of programming language statements. Therefore, a process can migrate from one machine to another without any restriction with respect to the machines' architectures.

In MPVM and ULP, although the migrations can be operated at any point within an application program, they do not support migration in a heterogeneous environment. In their designs, the migrations are performed at low level by transferring data, stack, and heap segments of the execution code to another machine. Thus, source and target machines for migration events must be binary compatible.

2.3 Migration Point and Checkpoint

In DOME [9] and Condor [10] developed at CMU and University of Wisconsin, Madison, respectively, checkpointing can be used for both fault-tolerance and process migration purposes. DOME uses its high-level checkpointing scheme mostly for fault-tolerance purposes. However, because the checkpointed processes can be restarted at other machines in a heterogeneous environment, DOME's checkpointing scheme can be used for process migration as well. In Condor, checkpointing is used for process migration. During a migration, Condor creates a checkpoint file before terminating the original process. This checkpoint file will be used when the process is restarted at a new machine.

Checkpointing involves the access of file system and disk, that slows down the migration process. More importantly, checkpointing lacks of the ability to handle network communication. In order to maintain a consistent global state, some processes that communicated with the migrating process have to be rolled back to their last saved checkpoints, which might cause other processes to roll back, and so on, until the program might be rolled back to its beginning. This effect is known as *Domino Effect* [11]. Checkpointing achieves fault tolerance but not efficience. It is not the best way to support process migration in network computing. Migration and checkpointing schemes should be designed separately for best performance. In MpPVM, necessary data is transferred directly from migrating machine to migrated machine through network communication. The compiler-assisted and user-directed concepts will be applied to migration points instead of checkpoints.

In the user–directed version of Libckpt, developed at University of Tennessee [12], memory exclusion concepts are applied to reduce memory overhead when saving to its checkpoint file during the execution of a process. Similarly, to minimize memory overhead, MpPVM has its own memory analysis to find out the *necessary* set of variables for each migration point. This set of variables is the smallest set of data to be transferred across the machines during the migration. Since the usage of checkpoints and migration points is very different, MpPVM's memory analysis is totally different from that of Libckpt.

3 Overview of MpPVM

MpPVM has three main components:

- *MCL* (MpPVM's precompiler), the precompiler that translates PVM source code into MpPVM source code by dividing its input program into several subsequences of instructions and inserting migration points to separate those subsequences from each other. MCL will perform *migration point analysis* and *necessary data analysis* to determine the locations of migration points in the source program and to evaluate the minimum set of variables to be transferred at each migration point, respectively. Finally, MCL will insert migration operations, including global definitions and macros, to the source program to produce its output.

- *Mpd* (MpPVM's daemon), the modified version of *pvmd,* which handles reliable point-to-point, indirect message passing mechanism in the migration environment. It also provides a protocol to support data communication among the scheduler, the migrating process, and the new process on an idle machine at a migration event.

- *Mlibpvm* (MpPVM's library), the modified version of *libpvm,* which provides programming language subroutines for the application programs and for the scheduler (the resource manager).

To develop software on MpPVM, users must feed an application program written in C or FORTRAN to the precompiler (see Figure 1). The precompiler will produce two output files, a map file (MAPF) and a modified file (MODF). The map file (MAPF) will show locations of every

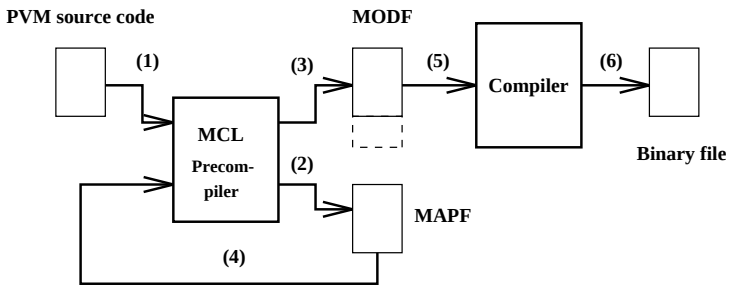

Figure 1. Basic steps in developing application programs for MpPVM.

migration point. If the users are not satisfied, they may change this map file and then input it to the precompiler again. The modified file (MODF) is the original PVM source file that was modified by the precompiler (MCL) to support process migration. MCL will analyze the structure of the program and then insert the necessary definitions and macros at every migration point (Mp). After getting acceptable MAPF and MODF files, the MODF will be compiled (using a regular Fortran or C compiler) and linked with Mlibpvm, the MpPVM library, to create the MpPVM executable file.

Before running the applications, Mpd and the scheduler must be running in a parallel computing environment. Like *pvmd*, Mpd is a daemon process that runs on every computer. The cooperation of every Mpd in the system can be viewed as a logical computing unit called *a virtual machine.* On the other hand, the scheduler is a process (or processes) that monitors and controls the workload of the environment. At run–time, applications will request services from the virtual machine and the scheduler. The virtual machine provides indirect message passing services for the applications; while, the scheduler handles requests such as process creation, termination, and other process operations that affect the workload and configuration of the system.

When load imbalance occurs, processes may be migrated to solve the problem. In migrating a process, the scheduler will determine the migrating process and choose one of the idle or lightly loaded machines to be the destination of the migration. Then, the scheduler will signal the destination machine to load the *equivalent* MpPVM executable file, the binary files that were generated from the same MODF file as that of the migrating process, into its system. In a heterogeneous environment, these equivalent execution files have the same functionalities and execution behaviors since they are generated from the same source code. This loading operation is called *process initialization.* According to the definitions and operations generated in the MODF file, the loaded process will wait for the connection from the migrating process. Usually, the initialization is performed when the scheduler wants to migrate a process. In our model, the process can also be initialized at any time before the migration is needed. This situation is called *pre–initialization.* Since the destinations of process migration are one of the idle or lightly–loaded machines, pre-initialization will not effect other users in the system. The strategies to manage pre–initialization depend mostly on the design of the scheduler. Although the implementation detail of the scheduler is not a focus of this study, pre-initialization is recommended as an option to reduce migration overhead.

After initialization, the scheduler will send information of the initialized process to the migrating process. The migrating process will make a direct TCP connection to the initialized process and start transferring its data. When the migration is finished, the migrating process will terminate. The execution will be resumed by the new process at the migration point on the destination machine.

4 MCL

In this section, we describe the functionalities of MCL including the migration point analysis and data analysis. We show the definitions and macros generated in the MODF file and illustrate how these macros work during process migration.

4.1 Migration Point Analysis

MCL adopts a new approach for migration point analysis which is different from any existing software environment. MCL views each programming language statement as an *instruction*. The instructions are classified into three types: control, plain, and library instruction. Control instruction consists of branching, loop, and subroutine instructions. Branching instructions are the conditional statements such as the IF–THEN–ELSE or CASE statements in FORTRAN or C. Loop instructions are those that create repetitive executions such as *while()* and *for() loops* in C, and *do*$\cdots$ *continue loops* in FORTRAN. A subroutine instruction is a subroutine calling statement in the language. Plain instructions are the instructions or operations that come with the programming language such as *+*, *-*, ***, */*, *etc.* Finally, library instructions are subroutines or operations defined in a standard library, such as functions and subroutines in *stdlib* or *Mlibpvm*, etc.

For simplicity, in our analysis model for automatic migration-point insertion, we assume that every control, plain, and library instruction requires a fixed number of CPU cycles for its execution. Arithmetic (+,-,*,/, etc.), logical (.AND., .OR., etc.), relational (.EQ., .LE., .GE., etc.) operations and other instructions that do not depend on run-time information are examples of these types of instructions. According to the assumptions, every plain or library instruction in this model is associated with a constant value, C_i, which represents its approximate execution cost. To perform automatic insertion of migration points, users have to specify a maximum cumulative Execution Cost (EC) value between any two migration points. Then, the following strategies are applied:

1. In a sequence of plain and library instructions, MCL will insert a migration point at the location where the cumulative execution cost value, counting from the beginning of the sequence or the previous Mp, is equal to or exceeds the EC value. After the insertion, the execution cost counter will be reset. Then, MCL will continue its analysis at the next instruction right after the new Mp.

2. When MCL encounters a branching instruction such as an "if–then–else" statement, it will assign different counters to accumulate execution costs for each branch. Then, MCL will analyze the body of each branch separately. At the exit of the branching instruction, the

maximum accumulated cost counter among these branches will be used at the continuing analysis in the next instruction.

3. In case of loop, if the number of iterations can be determined at compile–time, the insertion of migration points will depend on relationship of the Total execution Cost (TC) of the loop and the value of EC. The TC value is simply defined by the multiplication of the number of iterations (N) and the Total execution cost of the loop Body (TB).

   In case $TB \geq EC$, MCL will normally enter the loop and accumulate the execution cost. The same set of strategies will be applied again when MCL faces new instructions in the body of the loop. At the end of the loop, MCL will just continue its analysis at instructions outside of the loop body. On the other hand, if $TB < EC$, there are two cases to consider. First, if $TC \geq EC$, MCL will put a migration point and a special index at the end of the loop body. At a migration event, this special index will be used by the migration macros to determine the number of iterations of the loop that can be migrated. This scheme is applied in case the loop has a small number of instructions but a large number of iterations. Second, if $TC < EC$, MCL will just accumulate the TC value and skip the loop.

   In case the number of iterations cannot be determined, if $TB \geq EC$, analysis as described above will be applied. Otherwise, MCL will assume that $TC \geq EC$ and apply the same strategy used for the known number of iterations.

4. From the last Mp to the end of the program, the cumulative execution cost must be at least equal to the EC value.

5. In case of subroutine calls, if the Total execution cost of the Subroutine body (TS) is less than EC, MCL will skip the subroutine call and accumulate the TS value. In this study, MCL will ignore recursive calls in the subroutine body by continuing its analysis at the instruction after the calls.

   If $TS \geq EC$, MCL knows that at least a migration point will be inserted in the subroutine body. Thus, it will insert a migration point right before the function call to keep track of subroutine calling information at the migration event. The functionality of this migration point will be clarified in Section 4.3. In the subroutine, MCL will reset its cost counter and start the new analysis at the beginning of the subroutine body. After every subroutine call, MCL will also reset its cost counter and start the analysis again.

In applying the above rules, higher priority is given to the rule with higher number. For example, the fourth rule has greater priority than the third, and so on.

In Figure 2, we give a simple example of how to put migration points into a program. Let E be an expression and I_0 to I_9 and I_{s0} to I_{s3} be plain or library instructions. We assume that each instruction I_i has its relative execution cost C_i, where $i \in \{0 \cdots 9\} \cup \{s0 \cdots s2\}$. M_i, where $i \in \{0 \cdots 5\}$, represents the location of a migration point in the source file.

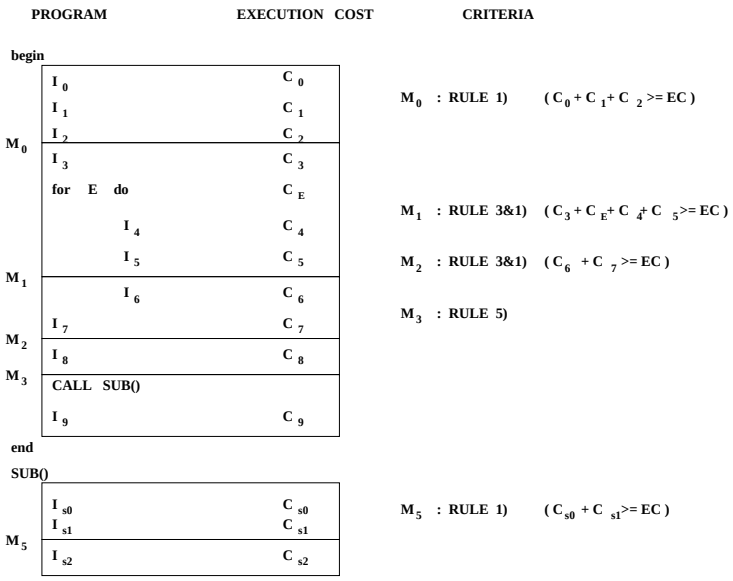

Figure 2. A simple example of how to insert migration points.

Let us start the analysis from the beginning of the program. After scanning through the first three instructions, MCL found that $C_0+C_1+C_2 \geq EC$. Thus, MCL inserts the first Mp at M_0 using the first rule. After that, MCL reset its cost counter and starts to accumulate the execution cost from I_3. When MCL encounters a "for" loop, the third rule is applied. Since $C_E+C_4+C_5+C_6 > EC$ (or $TB \geq EC$), MCL simply enters the loop and continues its analysis. Within the loop, a new Mp is put at M_1 because $C_3 + C_E + C_4 + C_5 \geq EC$. From I_6 to the end of the loop, there is only one instruction (I_6) and its execution cost is less than the EC value. Thus, according to the third rule, MCL jumps out of the loop and continues its consideration at I_7. Since $C_6 + C_7 \geq EC$, MCL puts an Mp at M_2. While counting the execution cost from I_8, MCL encounters a subroutine call and the fifth rule is employed. Since the total execution cost of subroutine body (TS), $I_{s0} + I_{s1} + I_{s2}$, is greater than EC, MCL knows that at least one migration point will be inserted into the subroutine body. Therefore, a new Mp is inserted at M_3 to keep track of function calls during a migration event. In the subroutine body, MCL starts collecting execution costs of instructions in the subroutine and then perform the same analysis as in the main function. Thus, according to the first rule, a new Mp is inserted at M_5 because $C_{s0} + C_{s1} \geq EC$.

After that, MCL will reset its cost counter and start the analysis from I_9. Suppose that $C_9 \geq EC$, according to the first rule, an Mp should be inserted right after I_9, which is the end of the program. Obviously, this situation conflicts with the fourth rule. Because the fourth rule has more priority than the first one, MCL will not insert a migration point after I_9. This scheme automatically prevents a process from being migrated to a new machine when its execution is almost finished.

4.2 Data Analysis

The goal of data analysis is to minimize the data transfer time during process migration. We do this by conducting *necessary data analysis*. MCL finds the set of variables that have been initialized before the migration point and the set of variables that will be referred to by other instructions after the migration point. In Figure 3, a migration point (Mp) and its data analysis are given. In this example the two sets of variables would be {a, b, x} and {x, b, c, y}, respectively. The intersection of these two sets, {x, b}, gives us the minimal set of data needed to be migrated at Mp.

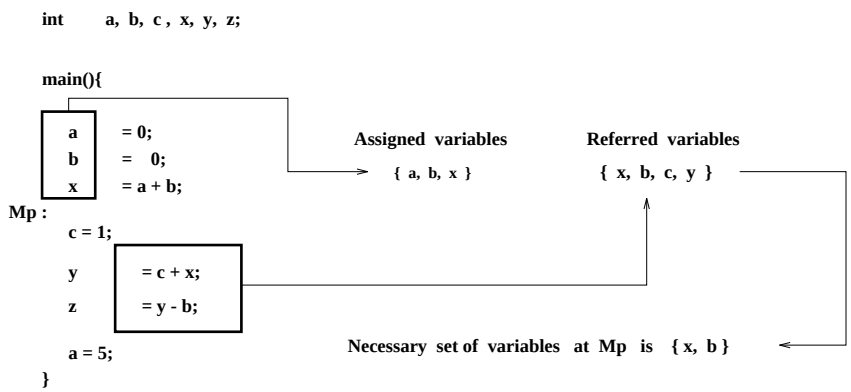

Figure 3. A simple example of necessary data analysis.

The idea behind the data analysis methodology is that the valid set of variables to be migrated should be those that have already been initialized before the migration point and will be used by other instructions after the migration is finished.

4.2.1 Assigned and Referred Variables

In the data analysis, we define two sets of variables: the *Assigned Variable* set and the *Referred Variable* set. The *Assigned Variable (AV)* set is a set of variables that are assigned values after the execution of an instruction, expressions, or a sequence of instructions; while, the *Referred Variable (RV)* set is a set of variables whoes values are used during the execution.

We classify variables in C programs into two types: the *fixed* and *pointer* variables. The fixed variables are whose are bound to single memory addresses at run-time. A particular memory space (or a storage) will be allocated for a variable to store its values. The variable will refer to the start location of that memory space for all of its lifetime. Examples of these variables are those declared at examples 1, 2, 3, 7, 10, and 11 in Table 1. On the other hand, we call the variables that can dynamically change their binding to memory addresses at run-time as pointer variables. Since a pointer is a variable that stores the address of a memory storage, its address content can be changed at any time during the execution of the program. Examples of pointer variables are shown at examples 4, 5, and 9 in Table 1.

The difference between the fixed-size and pointer variables effects the scheme to extract the AV

Example	Variables	Instructions	AV	RV
1	int a, b, c;	$a = b + c;$	$\{a\}$	$\{b, c\}$
2	int d[10], f[10];	$d[i] = a + f[j];$	$\{d\}$	$\{i, j, a, f\}$
3	struct... g, h;	$g.x = h.y + 2;$	$\{g\}$	$\{h\}$
4	int *p;	$p = \&d[0] + 5;$	$\{p\}$	$\{d\}$
5	struct... *q;	$*(p + 1) = *(q- > x);$	$\{p\}$	$\{p, q\}$
6		$*(q- > x) = *(p + 1);$	$\{q\}$	$\{p, q\}$
7	int *r[10];	$r[0] = d;$	$\{r\}$	$\{d\}$
8		$*r[0] = d[1];$	$\{r\}$	$\{d\}$
9	int (*s)[10];	$s = t;$	$\{s\}$	$\{t\}$
10	int t[2][10];	$s[0][0] = a;$	$\{s\}$	$\{s, a\}$
11	int u[2][10];	$s = u;$	$\{s\}$	$\{u\}$
12		$s[0][0] = b;$	$\{s\}$	$\{s, b\}$

Table 1. Examples of high-level instructions with their AV and RV sets.

and RV sets from a high-level instruction. If we assign a value to the storage of a fixed variable, the variable name will be put to the AV set. On the other hand, if the value is assigned to the storage of a pointer variable, the pointer name will be assigned to both the AV and RV sets. Since the address of a storage stored in the pointer variable can be changed during execution, the program will have to *refer* to the address of the pointer first before *assigning* a value to its storage.

In Table 1, Example 1 shows a simple assignment statement where the integer variable a, b, and c have their names bounded to fixed addresses. The binding will not change during a program execution. Thus, we can directly access the storages by referring their names. The arrays and structures in Examples 2 and 3 are also fixed variables since their names cannot be altered at run-time. On the other hand, variables p and q declared in Examples 4 and 5, respectively, are pointer variables. Their contents could be changed during program execution as shown in Example 4. Therefore, we need to refer to their address content first before accessing the storage that the address points to. As the results, p is put in both the AV and RV sets in Example 5, and q is also put in both sets in Example 6. In Example 7, r is declared as an array of pointers. Since r is bounded to a fixed-size storage, it is a fixed variable. We can refer to its storage directly. Thus, we just put r in the AV set in Example 8. From Example 9 to 12, s is declared as a pointer to an integer array of size 10. Since s is a pointer variable its address content could be assigned to various storages with the same type as shown in Example 9 and 11. Thus, s is put in both the AV and RV sets in Examples 10 and 12.

4.2.2 Necessary Variables

According to the basic ideas of data analysis, the *Necessary Variable (NV)* set at a migration point is defined by the intersection of the AV set for instructions that execute before the migration point and the RV set for instructions that execute after the migration point.

We now define the AV, RV, and NV sets for the different language constructs including sequential, branching, loop, and subroutine. The notation $B_i \Rightarrow B_j$ is used extensively in our analysis to represent an execution flow from B_i to B_j where B_i and B_j are instructions or sequences of instructions. We also let AV_i and RV_i represent AV and RV sets of B_i, respectively. Note that i and j are any integer or characters.

In case of sequential code, let B_0 and B_1 be two continuing sequences of instructions that are separated by a migration point Mp. According to the definition of the NV set, we can describe the NV set at the migration point Mp as $AV_0 \cap RV_1$. The total AV and RV sets of this sequential code are $AV_0 \cup AV_1$ and $RV_0 \cup RV_1$, respectively.

(a) Branching	(c) Subroutines
$B \rightarrow B_0\ B_1\ B_2$	$M(fp_m) \rightarrow B_1\ f_0(ap_0)\ B_2\ Mp_0\ B_3\ f_0(ap_1)\ B_4$
$B_1 \rightarrow if\ (\ E\ )\{\ B_{11}\ \}else\{\ B_{12}\ \}$	$f_0(fp_{f0}) \rightarrow B_{01}\ f_1(ap_2)\ B_{02}\ f_2(ap_3)\ B_{03}\ f_4(ap_4)\ B_{04}\ f_0(ap_5)\ B_{05}$
$B_{11} \rightarrow B_{111}\ Mp_1\ B_{112}$	$f_1(fp_{f1}) \rightarrow B_{11}\ f_2(ap_6)\ f_3(ap_7)\ B_{12}$
$B_{12} \rightarrow B_{121}\ Mp_2\ B_{122}$	$f_2(fp_{f2}) \rightarrow B_{21}\ f_3(ap_8)\ B_{22}$
(b) Loop	$f_3(fp_{f3}) \rightarrow B_{31}\ f_0(ap_9)\ f_1(ap_{10})\ B_{32}$
$B \rightarrow B_0\ B_1\ B_2$	$f_4(fp_{f4}) \rightarrow B_{41}\ f_1(ap_{11})\ B_{42}$
$B_1 \rightarrow for\ (\ E\ )\{\ B_{11}\ Mp_1\ B_{12}\ \}$	

Table 2. Examples of language constructs. M, B, and B_i, where i is any integer or character or combination thereof, represent sequences of instructions. while f_i and Mp_i denote a function call and a migration point, respectively.

Table 2 shows examples of branching, loop, and subroutine structures that are discussed in our analysis. The branching structure is described in Table 2(a). In this example, we assume that the migration points Mp_1 and Mp_2 are inserted into the sequences of instructions B_{11} and B_{12}, respectively. Mp_1 divides B_{11} into two continuing sequences of instructions B_{111} and B_{112}. Likewise, Mp_2 divides B_{12} into B_{121} and B_{122}.

Since the execution flow of B_1 could be either $E \Rightarrow B_{11}$ or $E \Rightarrow B_{12}$, the variables in E, B_{11}, and B_{12} will be used in defining the AV_1 and RV_1 sets. Therefore, the AV_1 and RV_1 sets are $AV_1 = AV_E \cup AV_{11} \cup AV_{12}$ and $RV_1 = RV_E \cup RV_{11} \cup RV_{12}$, respectively. Since the execution flow that pass Mp_1 is $B_0 \Rightarrow E \Rightarrow B_{111} \Rightarrow B_{112} \Rightarrow B_2$ and the flow that pass Mp_2 is $B_0 \Rightarrow E \Rightarrow B_{121} \Rightarrow B_{122} \Rightarrow B_2$, the NV_1 at Mp_1 as well as the NV_2 set at Mp_2 can be described as

$$\begin{aligned} NV_1 &= (AV_0 \cup AV_E \cup AV_{111}) \cap (RV_{112} \cup RV_2), \\ NV_2 &= (AV_0 \cup AV_E \cup AV_{121}) \cap (RV_{122} \cup RV_2). \end{aligned}$$

We present a loop structure in Table 2(b). We also assume that the migration point Mp_1 is inserted into the loop body, which divides the body into two sequences of instructions: B_{11} and B_{12}. According to the given structure, no matter how many times the execution of loop B_1 repeats, only E, B_{11}, and B_{12} are executed in the execution flow. Thus, AV_1 and RV_1 are defined

by $AV_1 = AV_E \cup AV_{11} \cup AV_{12}$ and $RV_1 = RV_E \cup RV_{11} \cup RV_{12}$, respectively.

Since the execution flow in the loop B_1 is $E \Rightarrow B_{11} \Rightarrow B_{12} \Rightarrow E \Rightarrow B_{11} \Rightarrow B_{12} \Rightarrow \cdots \Rightarrow E$, the expression and instructions including E, B_{11}, and B_{12} could be executed either before or after the migration point Mp_1. Therefore, the NV_1 set at migration point Mp_1 is

$$NV_1 \quad = \quad (AV_0 \cup AV_E \cup AV_{11} \cup AV_{12}) \cap (RV_{11} \cup RV_{12} \cup RV_E \cup RV_2).$$

The precompiler will analyze contents of each function in the application program separately. First, it will determine the AV and RV sets for every instruction, and then consider the NV sets of every migration point in the function body. In the case that the instruction is a function call, we first analyze the actual parameters and then the contents of the function.

Every variable name in the actual parameters that passes pointers or memory addresses to the function are assigned to the AV and RV sets because their values could be altered during the execution of the function. On the other hand, the actual parameters that are passed by value are put in the RV set. Since the values are copied to the function, there would be no changes to the parameters after the function call is finished.

We divide contents of a function into two parts: a header and a body. The function header consists of a function name and its formal parameters. Since the formal parameters are assigned values at every function entry, we put each formal parameter into the AV set. In the function body, the analysis described in the previous section for the sequential, branching, and loop structures is applied. However, different analysis is needed when recursive subroutine calls are encountered. We demonstrate the scheme to analyze the AV and RV sets of a function call as follows.

Table 2(c) shows an example of a migration point Mp_0 being inserted into a function M that has function calls (f_i) where $0 \leq i \leq 4$ in its body. We let fp_i where $i = m$ or $0 \leq i \leq 4$ be sets of formal parameters; whereas, the ap_j where $0 \leq j \leq 11$ be sets of actual parameters. We also suppose that AV_{fpi} and RV_{fpi} represent the AV and RV sets of fp_i, respectively. Likewise, the AV_{apj} and RV_{apj} are the notations for the AV and RV sets of ap_j. According to the definition of the function M, we can define the NV_0 set at Mp_0 as

$$NV_0 = (\ AV_{fpm} \ \cup \ AV_1 \ \cup \ AV_{f0} \ \cup \ AV_{ap0} \ \cup \ AV_2\ ) \ \cap \ (\ RV_3 \ \cup \ RV_{f0} \ \cup \ RV_{ap1} \ \cup \ RV_4\ ).$$

Every formal parameter in the function M is put in AV_{fpm} at the entry of the function. At the function call $f_0(ap_0)$, we can find out the AV set of the actual parameters ap_0 (AV_{ap0}) by considering how they are passed. We can use the same analysis to obtain RV_{ap1} at the function call $f_0(ap_1)$. We let AV_i and RV_i represent the AV and RV sets of other instructions that are not subroutine calls (B_i).

According to scope rules, only the global variables and the actual parameters can have their values (or any values in the data structure that they can refer to) accessed in a function call instruction. Therefore, the set of global variables and actual parameters is the superset of the AV and RV sets of the function call.

Since we already have the AV and RV sets of actual parameters of $f_0(ap_0)$ and $f_0(ap_1)$, we only need to derive the set of global variables that are assigned values (AV_{f0}) or referred to (RV_{f0}) during execution of the function. To find the AV_{f0} and RV_{f0} sets, we can recursively draw infinite execution flows of f_0. However, no matter how many times the direct or indirect recursions occur in the execution, the maximum set of B_i that could be executed in this structure is

$$\{B_{01},\ B_{02},\ B_{03},\ B_{04},\ B_{05},\ B_{11},\ B_{12},\ B_{21},\ B_{22},\ B_{41},\ B_{42},\ B_{31},\ B_{32}\}.$$

Likewise, the maximum set of function calls that could occur is

$$\{f_1(ap_2),\ f_2(ap_3),\ f_4(ap_5),\ f_0(ap_5),\ f_2(ap_6),\ f_3(ap_7),\ f_3(ap_8),\ f_0(ap_9),\ f_1(ap_{10}),\ f_1(ap_{11})\}.$$

Now we know all the instructions and function calls that the recursion can possibly go through. What we have to do now is just to analyze the global variables that appear in these instructions. Let GAV_i and GRV_i be the AV and RV sets of global variables in B_i, respectively. Also, we let GAV_{api} and GRV_{api} represent the AV and RV sets of the global variables in the actual parameters ap_i where $0 \leq i \leq 11$. As a consequence, we can then define AV_{f0} and RV_{f0} as

$$\begin{aligned}
AV_{f0} &= (GAV_{01} \cup GAV_{02} \cup GAV_{03} \cup GAV_{04} \cup GAV_{05} \cup GAV_{11} \cup GAV_{12} \\
&\quad \cup GAV_{21} \cup GAV_{22} \cup GAV_{41} \cup GAV_{42} \cup GAV_{31} \cup GAV_{32}) \\
&\quad \cup (GAV_{ap2} \cup GAV_{ap3} \cup GAV_{ap4} \cup GAV_{ap5} \cup GAV_{ap6} \cup GAV_{ap7} \\
&\quad \cup GAV_{ap8} \cup GAV_{ap9} \cup GAV_{ap10} \cup GAV_{ap11}), \\
RV_{f0} &= (GRV_{01} \cup GRV_{02} \cup GRV_{03} \cup GRV_{04} \cup GRV_{05} \cup GRV_{11} \cup GRV_{12} \\
&\quad \cup GRV_{21} \cup GRV_{22} \cup GRV_{41} \cup GRV_{42} \cup GRV_{31} \cup GRV_{32}) \\
&\quad \cup (GRV_{ap2} \cup GRV_{ap3} \cup GRV_{ap4} \cup GRV_{ap5} \cup GRV_{ap6} \cup GRV_{ap7} \\
&\quad \cup GRV_{ap8} \cup GRV_{ap9} \cup GRV_{ap10} \cup GRV_{ap11}).
\end{aligned}$$

After getting the NV set of each migration point, MCL will classify the variables in the NV set into two types: the necessary local variables and global variables. The *necessary local variables* (NV_l) are those in the NV set defined locally in the function that contain the migration point. The *necessary global variables* (NV_g) are the variables in the NV set declared as global variables in the program.

4.3 Migration Macros and Variables.

After defining migration points and their necessary variables, MCL will insert special global variables and macros to create its output (the MODF file). These global variables and macros are reserved by MCL and must be different from those defined by a user. These variables and macros are built for two major purposes: to migrate a process from the original machine, and to resume

the execution of the process at the destination machine.

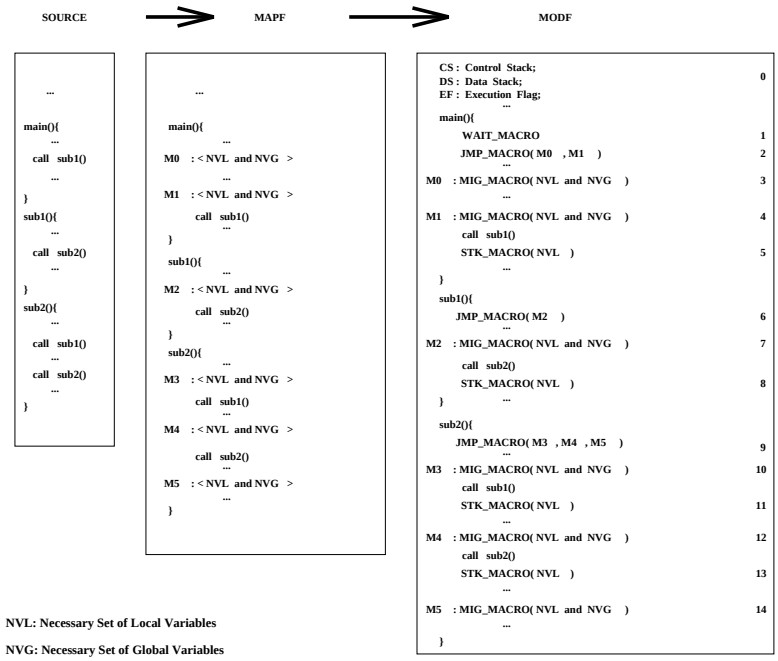

Figure 4. An example of how MCL puts its special purpose macros and variables to a source program.

Figure 4 shows examples of a source file, a MAPF file, and a MODF file. After creating the MAPF file, MCL will generate the MODF file by inserting global variables and macros. The global variables include a Control Stack (CS), a Data Stack (DS), an Execution Flag (EF), and other variables such as those for inter-process communications at the top of the file. The *control stack (CS)* is used to keep track of function calls before the migration. During a migration, the name of the migration point where the migration occurs and the names of the migration points associated with every function called before the migration will be pushed into the control stack. We also need to maintain a *data stack (DS)* to store the necessary local and global data at the point where migration occurs and to keep the necessary data for the names of migration points stored in the CS. Although stored in different stacks, each control and data item must tightly correspond to one another.

The Execution Flag is a variable that stores a signal sent from the scheduler. Four types of flags are defined: normal (NOR), resuming (RES), migrating (MIG), and stack (STK). The normal flag (NOR) is the default flag. It tells the process to execute normally on the current machine. The resuming flag (RES) is assigned to the initialized process by the scheduler. The initialized process will wait for the connection from the migrating process. The migrating flag (MIG) notifies the migrating process to start the migration at the nearest coming migration point. Finally, the stack (STK) flag is used internally by the migrating process to keep information regarding function calls occurring before process migration.

After inserting variables on the MODF file, MCL will insert migration macros at various loca-

```
WAIT_MACRO

- get EF value from Scheduler;
if (EF == RES){
    - accept TCP connection and
      receive CS and DS from the
      migrating process;
}

JMP_MACRO

if (EF == RES){
    - pop Mp name from CS;
    - Jump to that Label in
      the function;
}

MIG_MACRO

if (EF == MIG){
    - push current Mp name to CS;
    - push NV to DS;
    - if ( the Mp label is in main() ){
            - request TCP connection and
              send CS and DS to the new
              process.
            - exit() the program;
      }
      else{
            - set EF to STK;
            - return() to the caller function;
      }
}
else if( EF == RES ){
    - pop NV set;
    - if( size of CS is zero ){
            - set EF from RES to NOR;
      }
}

STK_MACRO

if ( EF == STK){
    - push current Mp name to CS;
    - push NV to DS;
    - if ( the Mp label is in main() ){
        - request TCP connection and
          send CS and DS to the new
          process;
        - exit() the program;
      }
      else{
        - return() to the caller function;
      }
}

*** Note that NV stands for Necessary Variable set.
```

Figure 5. Pseudo code macros that support process migration.

tions in the source program. During a process migration, these macros will collect CS and DS stacks, transfer data across machines, and restore data to the appropriate variables in the destination process. These macros are WAIT_MACRO, JMP_MACRO, MIG_MACRO, and STK_MACRO. MCL will put WAIT_MACRO at the beginning of the main function to wait for the connection and the contents of CS and DS from the migrating process. JMP_MACRO is put right after WAIT_MACRO in the main function and at the beginning of the body of other functions. MIG_MACRO is inserted at every migration point. In case the migration point was inserted before the function call to keep track of subroutine calling instructions, a STK_MACRO will be inserted right after the function call associated by those migration points. The pseudo codes of these macros are shown in Figure 5.

At a migration event, the scheduler will initialize a process on an idle machine. The EF flag of the new process is set to RES. Then, the program will wait for a connection from the migrating process. After that the scheduler will send a signal to the migrating process to start migration operations.

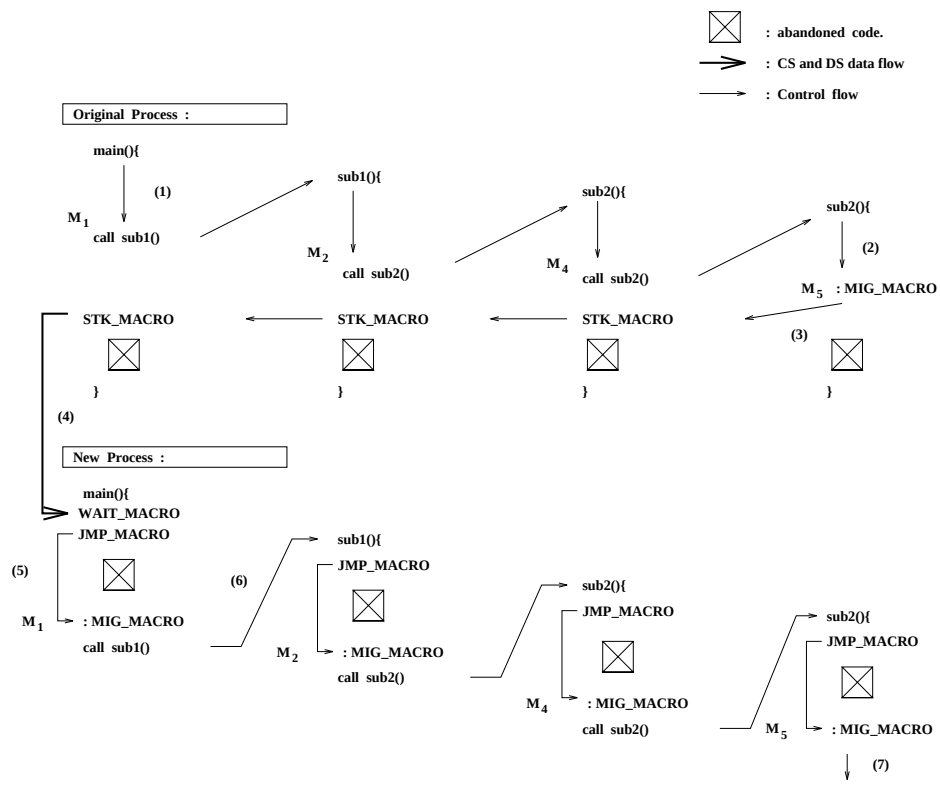

Figure 6. An example of how MpPVM conducts process migration.

To illustrate the migration operation, we will refer to the MODF file in Figure 4 and consider the situation in Figure 6. At (1) the process starts its execution from main() and then invokes sub1(), sub2(), and sub2() again as described in Figure 6. While executing instructions between M_4 and M_5, the scheduler signals the migrating process to change the EF flag from NOR to MIG (2). The process will continue its execution until it reaches M_5. Then it will enter MIG_MACRO.

Because the EF flag is MIG, the process will push the label M_5 to the CS stack and push *NV* (*necessary variables*) at M_5 to the DS stack. After that, the process will change the EF flag to STK and then abandon the rest of sub2() by returning to its caller function (3). At the caller function, the process will enter STK_MACRO in which it will save M_4 and its NV to CS and DS stacks respectively. Then, it will return to sub1(), its caller. The process will repeat the same action until it reaches the main() function. At the main() function, the instructions in STK_MACRO will be executed. However, at this point, not only the migration label M_1 and the NV at the main() function will be saved to the stacks, but the process will also send the CS and DS stacks to a new process in the destination machine (4).

After being initialized by the scheduler, the new process will start its execution and enter WAIT_MACRO. In this MACRO, since the default EF flag of the initialized process is RES, the process will wait for the connection and for the contents of CS and DS stacks from the migrating process. After getting this information, the process will enter JMP_MACRO. Then, the label name (M_1) will be popped from the CS stack. The execution will be transferred to the location of that label name by using a *goto* statement (5). At the migration point M_1, the process enters MIG_MACRO with the EF value of RES. It then pops appropriate values to variables in the NV. After that, the process will call sub1() as in (6). In sub1() the instruction in JMP_MACRO will pop M_2 from the CS stack, and then jump to M_2 . At M_2, the value of the NV are popped from DS stack and then sub2() is called. The JMP_MACRO in sub2() will transfer the execution to M_4. The process will pop DS stack and then call sub2() again. In the new sub2(), the execution flow will pop the migration point from the CS stack, jump to M_5, and then pop values of variables in NV set from the DS stack. Finally, the EF flag will be set to NOR and the process continues its execution (7).

5 Mpd, Mlibpvm, and Scheduler

Mpd is a modified version of pvmd. It has been modified to support process migration by adding two major features including:

- The migration table and message forwarding mechanism to handle point-to-point communication in a migration environment.
- Protocol to support data communications and signaling among the scheduler, the migrating process, and the new process on the migrated machine.

Mlibpvm is a modified version of libpvm. It is a programming library containing subroutines for supporting two different interfaces.

- *Scheduler Interface,* subroutines to monitor and control workload of the computing environment. These subroutines are used only by the scheduler. They include dynamic configuration subroutines such as pvm_addhosts() and pvm_delhosts(), and process migration subroutines

such as mppvm_initialize() to initialize a new process at idle or lightly–loaded machines and mppvm_migrate() to signal the migrating process to start its migration.

- *Application (or User) Interface*, subroutines that are modified to support efficient distributed computing in the migration environment. These are process control subroutines such as pvm_spawn() and pvm_kill(), and message passing subroutines such as pvm_send() and pvm_recv(). Since the process control subroutines can effect workload of the environment, every request of these subroutines should be approved and supervised by the scheduler. The message passing subroutines also need to be modified to cooperate with Mpd and to maintain reliable data communication in a process migration environment.

5.1 Assumptions about the Schedulers

In a migration environment, the scheduler is required to have functions to control the dynamic workload of the system. Therefore, additional assumptions regarding resource management in the migration environment must be discussed.

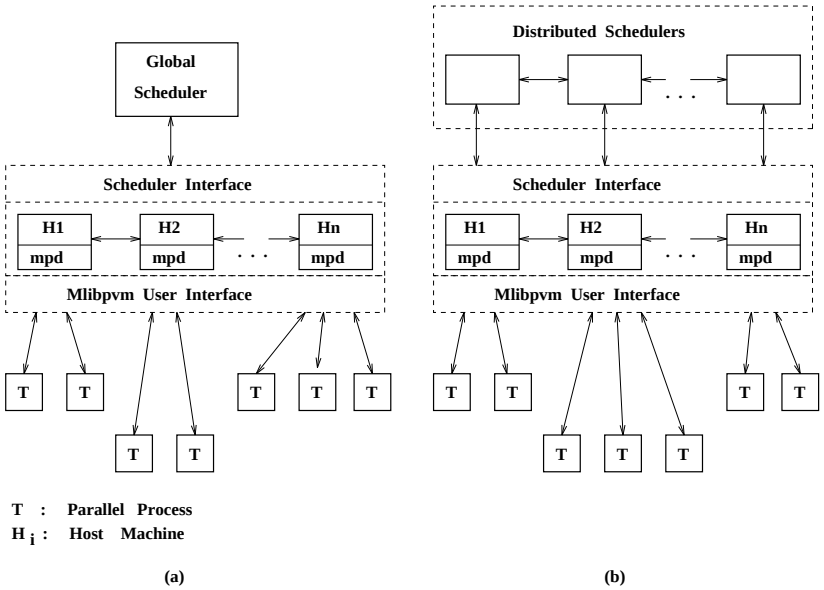

Figure 7. The overall structure of MpPVM virtual machine environment. (a) and (b) show two environments where global and local schedulers are used to control load balancing of the system.

In Figure 7(a), all load balancing calculation and process migration are controlled by the global scheduler. The scheduler could be either a server process implemented on a dedicated machine or a simple user process residing on one of the virtual machine's hosts. On the other hand, the server can also be implemented as a group of distributed processes residing on every host in the system as in 7(b). These processes will exchange information and work together to control load balancing of the system.

For the sake of brevity, the design and implementation of the scheduler are not discussed in this paper. In any case, we assume the scheduler has the following functionalities:

- The scheduler must classify parallel processes running under its supervision into two types: static and dynamic. A static process resides on one machine until its termination; whereas, a dynamic process can migrate from one machine to another.
- The scheduler must exhaustively seek idle or lightly-loaded machines in the heterogeneous environment and make underutilized machines ready for parallel application processes.
- To migrate a process, the scheduler must initialize a process at an underutilized machine before notifying the migrating process to make a connection and transfer its data to the new process. If the parallel process is already pre-initialized, the scheduler will send the identification of the pre-initialized process to the migrating process. In this case, we can substantially reduce the migration overhead.
- To maintain reliable data communication, in migrating a process, the scheduler must *inform* every Mpd of processes that had communicated with the migrating process in the past or will communicate with it in the future.

Based on these assumptions, the algorithm for the scheduler to perform process migration is given in Figure 8.

```
migrate ( task-id , task-name , current-host , destination-host ){
    - if  task-name is not pre-initialized then
        call mppvm_initialize() to initialize a new process in the
        destination host.
    - call mppvm_migrate() to send migration signal as well as
      the initialized task id, TCP socket id, and related
      information to the current Mpd and the migrating process.
    - wait until the migration finishes.
    - broadcast a pair of ( task-id, destination-host ) to every
        Mpd in a processing group.
    - wait until an acknowledgement signal is received
        from every Mpd in the processing group.
}
```

Figure 8. An algorithm for the scheduler to control a process migration.

5.2 MpPVM data communications

In an Mpd, a daemon for each host in the virtual machine and a data structure called *Migration Process Table (MPT)* are implemented. MPT, which contains a list of original task id's along with the corresponding current host id, is used mainly to support data communication in a migration environment. Every time a process sends messages to another process, the Mpd will scan through

its MPT table to find out the current machine of the target process. If it cannot find anything, Mpd will assume that the target process has never been migrated to other machines before. Then, the original PVM strategies of searching for a target machine will be employed.

Because MPT will be used extensively for data communication, the methodology to maintain this table is very important. It must be handled efficiently and immediately at every migration event. According to the algorithm in Figure 8, when migration occurs, the scheduler will check if the process with the same "task name" has already been pre-initialized in the destination machine. If not, an initialization will be performed on that machine. After that, the scheduler will send the migration signal and other information such as task id, TCP socket id of the new process to the migrating process and to the Mpd of the current machine. This current Mpd contacts the destination Mpd and the migrating process to update their MPTs. Then, the migrating process will open a TCP connection to the new task and start transferring its data.

At this point, the scheduler will wait until the migration is complete. After getting a "completed" signal from the Mpd of the new machine, the scheduler will broadcast the same migration information to the Mpd of every host in the processing group. Upon receiving the information every Mpd will modify its MPT. At the same time, the scheduler will wait for the acknowledgement signals. After getting the signals from all Mpd's in the processing group, the scheduler will continue its execution.

There are three possible communicating situations that may occur in this model during a migration. They are:

- If a message arrives at the current Mpd of the migrating process before the migration and has not yet been read by a pvm_recv() function call, the Mpd will store these messages in its buffer.

- If a message reaches the current Mpd of the migrating process during the migration, it will also be stored in the buffer. Once the migration is complete, the Mpd will flush this data to the destination machine.

- After migration, if the Mpd of the process that wants to send a message to the migrated process has got the broadcast migration information from the scheduler and has updated its MPT, the message will be sent directly to the Mpd of the new machine. Otherwise, the message will be sent to the Mpd of the last host and then forwarded to the new machine. Therefore, the faster we update the MPT of every Mpd in the same processing group, the less we have to forward messages.

Before its termination, each parallel process will send a termination signal to the scheduler. The scheduler will collect these termination signals until every process in the same processing group is terminated. Then, it will broadcast signals to every Mpd of that processing group to delete information of the terminated process from their MPT.

6 Preliminary Implementation and Experimental Results

A few prototypes and experimental programs have been developed to test the correctness and applicability of the design of MpPVM. Routines for transferring data and process states have also been developed. *pvmd* and *libpvm* of PVM have been modified to support the experimental MpPVM protocol.

In pvmd, we have implemented the MPT table and modified PVM point-to-point message passing mechanism to cooperate with the MPT. The Task-Pvmd and Pvmd-Pvmd protocols are also modified to support signaling among the migrating process, the new process, and the scheduler. In libpvm, a few subroutines for transferring data during process migration are implemented. These data must be encoded into the XDR format before being transferred across machines in a heterogeneous environment. The migration subroutines which are needed by the scheduler are also added to the libpvm library.

Based on the implemented prototype, we have conducted experiments to verify the feasibility and applicability of the migration point and necessary data analysis in the MpPVM design. A parallel matrix multiplication program with Master-Slave communication topology developed in [13] was chosen for our experiments. Both the PVM and MpPVM versions of this program are implemented. These two versions are different in that the MpPVM one contains a number of migration points at various locations in the program. Both experiments are performed on the same LAN environment consisting of four SUN Sparc IPC and fourteen DEC 5000/120 workstations.

The purpose of our first experiment is to verify the heterogeneous process migration of MpPVM and to examine the performance degradation of the PVM version when parallel tasks have to compete with local jobs for computing cycles. We assume that the new process at a destination of a process migration is pre-initialized. Since process initialization involves many factors such as disk accesses and NFS that might cause substantial amount of overhead, the availability of pre-initialization allows a faster process migration. However, this assumption requires the scheduler to have an efficient resource allocation policy in a migration environment.

In this experiment we ran four parallel processes on two SUN and two DEC workstations. One of the two DEC workstations was the test site that had continued requests from the machine owner (or other local users). When the workload on the test site increased, MpPVM migrated its process from the test site to one of the unused SUN workstations on the network. In the competing situation, we simulate the increasing workload of local computation by gradually adding light-weight processes, i.e., programs with few floating–point operations and small data size, to share more time slices from the CPU.

From Figure 9 we can see that MpPVM achieved a superior performance over PVM especially when the owner's jobs request more CPU time. In 9(a), a 3x400 and a 400x3 matrix are multiplied ten times. With competing local jobs, both parallel processing versions have an increased execution time with the requests from the owner. However, the execution time increase of MpPVM is about a one-half that of PVM when the machine owner actively uses the test site. This is because in this implementation only one migration-point is used and is located at the half-way point of the

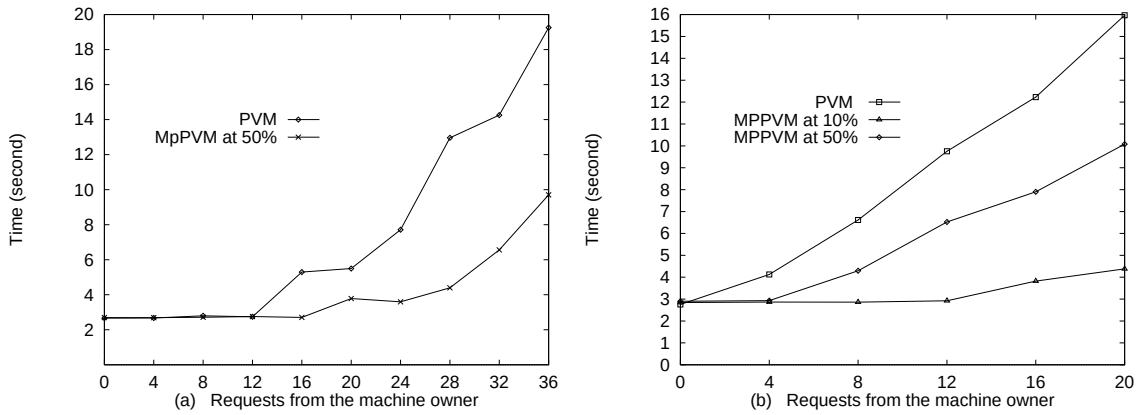

Figure 9. Comparisons of execution time between PVM and MpPVM matrix multiplication programs

total required computation. In 9(b), we still use the same configuration as 9(a) except that we increase the order of the matrices to 3x1600 and 1600x3 respectively. As a result, the parallel processing performance becomes more sensitive to the owner requests than in 9(a). In this test we have developed two MpPVM programs which will migrate from the overloaded machine after 10 percent and 50 percent of their executions, respectively. The performance of migration at 10 percent is better than that at 50 percent since the process spends less time competing with the owner computations. In general, the more migration points in a parallel program, the less the duration of competing with local computation will be and, therefore, a better performance is achieved.

In the second set of experiments, the goal is to investigate the effects of process migrations when both the problem and ensemble size scale up. Since we want to detect the degradation caused from process migrations only, the tests are conducted in a dedicated environment[1].

We scale the problem size of the MpPVM matrix multiplication program with the number of processors on 4, 6, 8, 10, and 12 workstations by following the memory-bounded scale-up principle[2] [14]. At each ensemble size, at most 3 and 5 slave processes are migrated when the program runs on 4 and 6 workstations respectively; whereas, 6 slave processes are migrated when the program uses a greater number of workstations. New processes of process migration were pre-initialized, and overlapping of process migrations was prevented in this experiment. The experimental results are depicted in Figure 10.

In Figure 10(a), each curve represents a relationship between execution time and the scaled problem size, indicated by the number of workstations, at a specific number of process migrations (m). At $m = 0$, there is no process migration. Likewise, the curves at $m = 1$ to $m = 6$ give the results with one to six process migrations, respectively. We can see that these curves increase approximately in parallel. For example, the curve of $m = 6$ (migrate six times) grows in parallel with the curve of $m = 0$ (no migration). Since the difference of execution time of $m = 6$ and $m = 0$

[1]The tests were conducted at midnight on Friday when no one else was on the system.

[2]For the matrix multiplication $A \times B$, the order of matrix A is 3x1600, 3x2400, 3x3200, 3x4000, and 3x4800 for ensemble size 4, 6, 8, 10, 12 respectively. The order of matrix B is symmetric with respect to the order of matrix A.

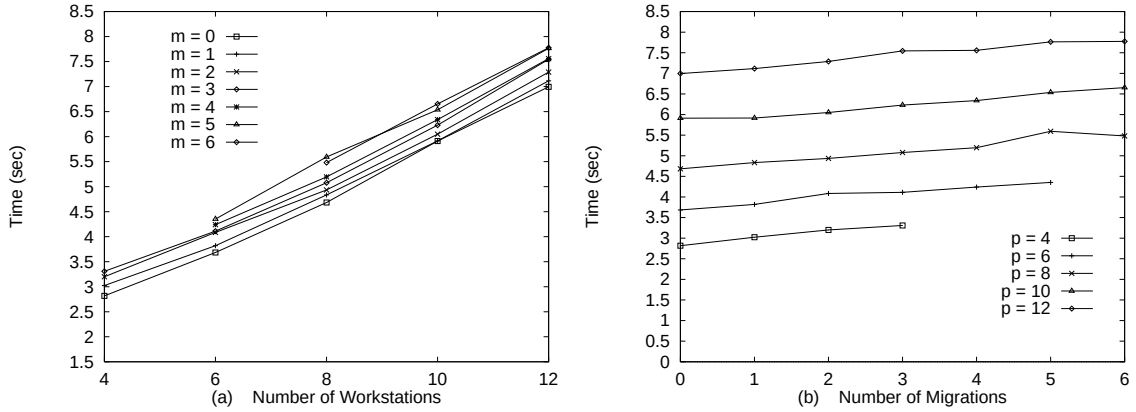

Figure 10. Comparisons of execution times of MpPVM matrix multiplication programs when the problem size scales up.

is the migration cost, we can conclude that the migration costs caused by six process migrations at the scaled problem size on 8, 10, and 12 workstations are approximately the same. Therefore, the migration cost is a constant and does not increase with the ensemble size in our experiments. The execution time increase is mainly due to the increase of computation and communication, not due to the migration cost.

Figure 10(b) further confirms that the migration cost does not increase with problem and ensemble size. In this figure, different curves depict the relation between the execution time and the number of process migrations at a specific ensemble size. We can see that the increase of execution time with the number of process migrations is consistently small compared with the computing and communication time. More importantly, the measured performance show that the process migration mechanism is very efficient. With six process migrations on a cluster of twelve workstations, for instance, the total process migration cost is less than one-half second. Thus, the experimental results show that the proposed process migration mechanism is implementable and practical in a distributed heterogeneous environment.

Process migrations have been prevented from being overlapped with each other or with computation and communication in our experiments. Process migration cost may be further reduced with this overlapping. On the other hand, while experimental results are very encouraging, the results are preliminary and based on certain assumptions. In our experiments, we assume the destination processes in a process migration are pre-initialized. Thus, at a migration event, the migrating process can transfer its data promptly to the destination process. Migration cost will be increased if the new process is initialized during process migration.

Efficient process migration needs the cooperation of many components such as the scheduler, the virtual machine (MpPVM), and the application processes. Performance of process migrations also depends on many factors including the inherited properties of the applications, communication topologies, network contentions, and efficiency of the scheduler. Since MpPVM is just one of the components with certain functionalities to support process migration, certain assumptions and limitations are unavoidable in current examination of the applicability and feasibility of its design.

The measured migration cost is only true for our application on our environment. It may vary with the computation/communication ratio of the application and hardware parameters of the underlying distributed platform, though we believe it represents the trend of general applications. Further experiments are planned for other parallel applications with different computing and communication structures.

7 Conclusion and Future Work

We have introduced a high-level mechanism and its associated methodologies to support efficient process migration in a non-dedicated, heterogeneous network computing environment. The newly proposed network process-migration mechanism is general. It can be applied to any distributed network environment. In particular, based on this mechanism, a software system named MpPVM has been designed and implemented to support efficient process migration for PVM application programs. We have studied the interaction between process migration and resource management and proposed modifications of pvmd and pvmlib of PVM to maintain reliable data communication among processes in a migration environment. Our implementation and experimental results confirm the applicability and potential of the proposed mechanism in a non-dedicated heterogeneous environment. Experimental results indicate the MpPVM software system is efficient and is scalable in the sense that it can carry out a process migration in tenth of milliseconds and the migration costs becomes less notable when the problem and ensemble size increase.

The implementation of MpPVM is still preliminary. The Mpd and Mlibpvm need to be improved to maintain high reliability and efficiency on various parallel applications. The MCL software and its supporting tools are also under development. To support an effective process migration environment, an efficient scheduler which controls workload of the distributed environment is inevitable. We plan to develop a scheduler that can efficiently exploit process migrations in a non-dedicated, heterogeneous computing environment. The performance evaluation and prediction according to effects of process migrations on various algorithms and heterogeneous platforms will also be investigated.

With the rapid advance in network communication, the concept of "virtual enterprise" network computing is becoming increasingly prevalent. That is, with certain agreement, computers on the network can use each others unused computing resources to speedup performance. Network process migration is indispensable for virtual enterprise computing. Our newly proposed process migration mechanism is important because it is the only mechanism available to support heterogeneous, point-to-point network process migration on top of existing distributed systems. The MpPVM software is a part of the SNOW project[3] (Scalable virtual machine environment for non-dedicated heterogeneous Networks Of Workstations) under development at Louisiana State University. We believe that the development of MpPVM and SNOW will have an important impact on the design of future software environments for virtual network computing.

[3]More information regarding the SNOW project can be found at http://bit.csc.lsu.edu/~scs.

Acknowledgment

The authors are grateful to A. Beguelin of Carnegie Mellon University for his help on understanding the implementation of PVM and on related work in process migration in a distributed environment.

References

[1] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam, *PVM: Parallel Virtual Machine – A Users Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[2] A. Beguelin, J. Dongarra, A. Geist, R. Mancheck, and V. Sunderam, "Recent enhancements to PVM," *The International Journal of Supercomputer Applications*, vol. 9, pp. 108–127, 1995.

[3] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, 1994.

[4] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor–a hunter of idle workstations," in *Proceeding of the 8th IEEE International Conference on Distributed Computing Systems*, pp. 104–111, June 1988.

[5] J. Casas, R. Konuru, S. Otto, R. Prouty, and J. Walpole, "Adaptive load migration systems for PVM," in *Proceedings of Supercomputing'94*, pp. 390–399, 1994.

[6] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty, and J. Walpole, "Mpvm: A migratable transparent version of PVM," Tech. Rep. CSE-95-002, Oregon Graduate Institute of Science and Technology, Dept. of Computer Science, Feb. 1995.

[7] G. Burns, R. Daoud, and J. Vaigl, "Lam: An open cluster environment for MPI." Available at ftp://tbag.osc.edu/pub/lam/lam-papers.tar.Z.

[8] F.-C. Cheng, P. Vaughan, D. Reese, and A. Skjellum, *The UNIFY system*, version 0.9.2 ed., Sept. 1994.

[9] J. N. C. Arabe, A. Beguelin, B. Lowekamp, E. Seligman, M. Starkey, and P. Stephan, "Dome: Parallel programming in a multi-user environment," Tech. Rep. CMU-CS-95-137, Carnegie Mellon University, School of Computer Science, Apr. 1995.

[10] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor–a hunter of idle workstations," in *Proceeding of the 8th IEEE International Conference on Distributed Computing Systems*, pp. 104–111, June 1988.

[11] B. Randell, "System structure for software fault tolerance," *IEEE Transactions on Software Engineering*, vol. 1, pp. 220–232, 1975.

[12] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Transparent checkpointing under UNIX." Appearing in USENIX Winter 1995 Technical Conference, Jan. 1995.

[13] A. Beguelin, E. Seligman, and M. Starkey, "Dome: Distributed Object Migration Environment," Tech. Rep. CMU-CS-94-153, Carnegie Mellon University, School of Computer Science, May 1994.

[14] X.-H. Sun and L. Ni, "Scalable problems and memory-bounded speedup," *J. of Parallel and Distributed Computing*, vol. 19, pp. 27–37, Sept. 1993.