A Runtime System for Autonomic Rescheduling of MPI Programs*

Cong Du, Sumonto Ghosh, Shashank Shankar, and Xian-He Sun
Department of Computer Science
Illinois Institute of Technology
{ducong, ghossum, shankar, sun}@iit.edu

Abstract

Intensive research has been conducted on dynamic job scheduling, which dynamically allocates jobs to computing systems. However, most of the existing work is limited to redistribute independent tasks or at the algorithm design level. There is no runtime system available to support automatic redistribution of a running process in a heterogeneous network environment. In this study, we present the design and implementation of a system that dynamically reschedules running processes over a network of computing resources via automatic decision-making and process migration. The system is implemented on top of MPI-2 and HPCM (High Performance Computing Mobility) middleware. Experimental and analytical results show that the runtime system works well. It makes dynamic rescheduling of running tasks possible and improves system performance considerably. While the implementation is for MPI programs and using HPCM, the design of the system is general and can be extended to other distributed environments as well.

1. Introduction

Runtime dynamic scheduling is a fundamental issue of parallel and distributed computing. The emergence of Grid [9], provides a promising platform for large-scale and resource intensive applications. The Grid provides the basic software infrastructure with mechanisms for resource sharing over a distributed heterogeneous network. The job scheduling and management system is an integrated component of Grid computing. When one system encounters some difficulties, such as a system failure, temporary resource unavailability, network outrage, system reconfiguration, or just performance degradation caused by preemption from high priority local jobs, the Grid jobs need to be rescheduled to another host to continue. In traditional job scheduling systems, task allocation is static. Once a task is assigned, it will stay where it is until it finishes or restarts at another site from the beginning. In these systems, a reassignment means the loss of all partial results. The nature of static allocation may cause dramatic performance loss in practice. Also, jobs may have to be rejected when a certain host cannot satisfy the jobs for required resources. However, if the mobility is supported and the decision is made automatically at runtime, a job can move from one host to another for both resource availability and performance gain. In this study, we intend to develop a novel system enabling dynamical reschedule of running processes over a network of computing resources, via automatic decision-making and process migration. We present the design and implementation of a runtime support system, which enables dynamic re-allocation of processes in a heterogeneous distributed environment. We also present a highly configurable and extensible rule-based mechanism for policy making that supports various system conditions in such environment.

Process migration is the act of transferring an active process from one computer to another. The process retains its execution sequence and memory state during a migration. The process is interrupted on the source machine and then is resumed at the break point with the same memory state on the destination machine [8]. HPCM is a heterogeneous process migration middleware [5, 6, 8].

The Message Passing Interface (MPI) is a standard library specification for message passing. It is proposed, developed broadly by vendors, implementers, and users. MPI-2 includes an extension to MPI-1 standard. It supports process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, and additional language bindings [17].

In this study, we design and implement a runtime system on top of HPCM and MPI-2, providing resource registration, resource monitoring, process registration, and soft-state management to support dynamic rescheduling of MPI tasks. The system has a rule-based decision-making component that coordinates with other components of HPCM as a commander, and invokes the migration when it reaches a migration decision according to a highly configurable and extensible rule-based mechanism. Though the system is implemented on top of the MPI and HPCM middleware, it is general and can be extended for

* This research was supported in part by National Science Foundation under NSF ACI-0130458, ANI-0123930, and EIA-0130673.

checkpointing-based or mobile computing systems, and for other distributed environments.

This paper is organized as follows. In Section 2, we give an overview of related works on process migration and job scheduling. In Section 3, we present the features of the rescheduler and describe the implementation details. We present the rule-based decision-making mechanism in Section 4. Section 5 shows the performance results and analysis. Section 6, finally, concludes this paper and presents future works.

2. Related Works

The existing computation and data Grid do not support dynamic task re-allocation in general. Traditional static task scheduling requires that before scheduling, the arrival time and execution time of tasks and their dependencies be known to the scheduling system. Usually their relationships are defined by a DAG [12]. Clearly, this condition cannot be met in Grid computing. Some researchers have developed new algorithms for dynamic scheduling with limitations. Some algorithms duplicate the tasks and issue them simultaneously on multiple computing resources [14], [16]. Some issue multiple copies of tasks to idle computing resources [10]. These algorithms have limitations on the usage of resources. They provide dynamic features at the expense of resources and performance. They may also cause mutual exclusion problems when utilized in the tasks requiring non-duplicable resources. Some of them are based on performance prediction. In this case, the system may pay a penalty in terms of prediction error or system failure. Once performance degradation or system failures occur, a reassignment is necessary. In practice, this can cause the loss of all partial results and dramatic performance degradation.

Due to its importance, intensive research has been done in process migration. Some of the early works, such as MOSIX [3], V [4] and Sprite [7], combine the migration functionality to cluster operating systems. Because these systems work on clusters with specific operating systems, they do not use or provide a general runtime re-allocating function. These systems cannot be used in a Grid environment.

There are several other systems implemented for widely used operating systems at user-level, e.g. Condor [15], or at kernel-level, e.g. Linux Zap [21] to support homogeneous process migration. Condor is geared towards High Throughput Computing, through resource and job matching. A centralized broker uses a Class-Ads mechanism to match jobs to resources and then, schedule jobs on the Condor machines. Condor uses a checkpointing-based mechanism to implement process migration at the user-level. It only supports homogeneous process migration. Although several job scheduling policies are proposed [1, 2], they do not support heterogeneous runtime process rescheduling. The Zap system supports migration of legacy applications through the use of loadable kernel modules and virtualization of both the hosts and processes with respect to each other [21]. It uses a checkpointing-based mechanism to support process migration and cannot migrate over heterogeneous environments. However, the Zap system has not implemented the mechanism for scheduling and re-allocation. Heterogeneous process migration has been studied by a few researchers [22, 24] who address several important problems and discuss how to build their prototype systems. They also propose some triggering conditions, but they do not develop those simple triggers into a runtime rescheduling system.

There are some other applications that obtain mobility via mobile agents or mobile codes. Compared to process migration, mobile agents are modularly designed applications that inherently need to “jump” or “goto” other machines. They are scheduled by the execution workflow itself. They work under certain predefined virtual environments, and work well only for certain special purpose applications [13].

3. System Design and Features

The basic functionalities of our rescheduler include resource monitoring and registration; process registration;

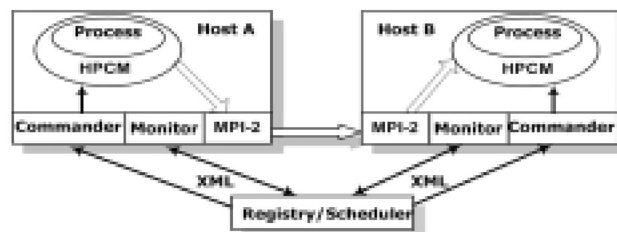

Figure 1. System Model

soft-state management; and a rule-based decision-making mechanism. It provides services that enable: effective communication between and within the hosts; transparent sharing, resources discovery and usage; and rule-based intelligent monitoring and decision-making.

Our system model consists of system state monitoring entities, commander entities and a process registration and decision-making entity. Figure 1 shows the system model.

A monitor and a commander entity reside on each host, including candidate destination hosts. There is also a central or hierarchical registry/scheduler, which can reside on any host with or without other entities. As shown in Figure 1, the monitor registers the host static information to the registry/scheduler and periodically gathers and updates the system status to it. The monitor determines the status of its local system resources as *free*, *busy*, *overloaded* or *unavailable*. It then reports its system status and other information to the registry/scheduler. The

registry/scheduler analyzes the data from monitors and makes a decision regarding which process to migrate and where to migrate it. The registry/scheduler sends a message to the source machine's local commander to initialize the migration. After receiving the message, the source machine's local commander issues a command to the migrating process to start the process migration. The migrating process, in turn, initializes a process at the destination machine through the dynamic process creation, and creates a communication channel between the migrating process and the initialized process. The initialized process, then, is ready to take over the computation on the destination machine. The migrating process transfers the execution, memory, and communication states to the initialized process on the destination host at the nearest poll-point (pre-defined possible point in the execution sequence where a migration can occur). The details of the process migration mechanisms can be found in [5, 6, 8].

3.1 Monitoring and Decision-Making

At each host, the monitor gathers system information and manages local system state based on the information. Local decisions of the system state are made and are reported to its registry/scheduler. Monitoring can be performed periodically or only when necessary. We chose the former for a better reaction time. Based on the rules used in monitoring, the monitor can be very light-weighted.

System information consists of static and dynamic information. Static information, such as host name, IP address, operating system, and memory size, is not likely to change during the lifetime of the specific monitoring entity. The static information is used only for one-time registration. On the other hand, dynamic information from computational resources, memory, networks, disks or applications is quite likely to change with time. We gather dynamic information either through the use of scripts (such as UNIX shell-scripts on *NIX, or batch-files on Windows), or through APIs provided (such as *sysinfo* on LINUX platform, or the *WMI* on Windows platform). We chose to implement a script-based mechanism, partially for the ease of implementation, and partially for its portability. We implemented the shell scripts using utilities like 'vmstat', 'prstat', 'ps' etc, on Sun Solaris 5.8 to gather system information. These mechanisms could be easily ported to LINUX where the shell scripts could read the system parameters from '/proc',

The monitors collect various types of information. They are:

- Processor utilization and load: parameters include the load average, CPU utilization and the number of processes per processor.
- Memory state: available memory and percentage of available memory for both virtual and physical memory, and the memory statistics regarding the process.
- Disk usage: parameters include disk space used, disk space available. It gathers the disk usage parameters of the various mount points.
- Communication: parameters include latency, and bandwidth.

Based on the information, the monitor determines the system status according to a rule-based mechanism. This decision is made locally and specifically according to the rules defined for a specific local system. We can accommodate our system into a large number of heterogeneous systems with very large gaps in both performance and resource availability.

3.2 Registry/Scheduler

Registry/scheduler is an entity that resides on any host in the HPCM system. It can also reside on a host without any other entities configured, and it is a global system state manager and decision-making entity. Although conceptually it is a centralized entity in the system, we can extend it as a service in hierarchy. Each local system has its own registry/scheduler and each registry/scheduler has its own upper level registry/scheduler. We can configure a local registry/scheduler on a local cluster and its upper level registry/scheduler to a specific organization, such as a Virtual Organization in a Grid environment. The lower level registry/scheduler has its own health condition, which indicates its overall workload and availability of each kind of resource. Usually, it is preferred that the migration destination is chosen inside one's control domain, which includes the systems registered to the same registry/scheduler entity. This hierarchical design solves the problem of a centralized bottleneck, thereby improving the performance and the system scalability.

Registration mechanisms can be either *pull* or *push* based. The good thing about the *pull* based registration mechanisms is that the registry/scheduler can decide when it needs the information and status of each host. It then queries the current information to make more optimized decisions. But, this also leads to the registry/scheduler having to make a query at runtime when a decision is expected, thus slowing down the process.

The other side is the *push* model where all the registrants are expected to refresh their status every once in a while. This model forces the clients to maintain timers and to constantly keep querying the status of the registry, thus guaranteeing a certain amount of traffic. In this model there are chances of flooding the registry, if all the registrants become synchronized.

In our system, the registration of resources is based on a soft-state mechanism, wherein clients have to regularly update their presence and state information to the

registry/scheduler through the *push* model, otherwise the registry/scheduler will consider them as *unavailable*.

Scheduling involves decision-making utilizing static information and runtime data. The registry/scheduler makes a decision on where to migrate a process based on "first fit" policy. From the machine list, the registry/scheduler chooses the first host, which is ready and owns all the resources required, as the migration destination host.

3.3 Communication

The monitor, commander and registry/scheduler of our system are components of communication. In addition to these, the migrating process and the initialized process are also involved in communication during process migration. Altogether, five kinds of communication parties coordinate and communicate with each other to form an automatic migration system as shown in Figure 1. We have developed several communication mechanisms, so that we can achieve high performance, scalability and extensibility. We discuss these mechanisms as follows:

- Migrating process and initialized process: The communication data between the migrating process and initialized process include the execution state and memory state. The amount of communication highly depends on the application. We have built up mechanisms to reduce the communication cost in process state transfer. We still need faster communication to improve the migration performance. In the following discussions, we use the communication channel of LAM MPI-2 [18] in process state transfer. Currently, we have tested several communication channels for the process state transfer including TCP/IP, MPI and PVM. We take advantage of the MPI-2 standard dynamic communicator management to support communication state migration over MPI-2. To enable process migration over MPI-2, we need to dynamically create a process with a communicator and join the communicators together, so that the migrating process and initialized process can communicate in one communicator. Fortunately, dynamic process management is defined in MPI-2 standard and LAM is one of the few MPI environments that support these functions. We cannot use other MPI such as MPICH-2 [19] and Sun MPI [20] because they do not support the dynamic process management, and the implementation is in their future schedule.
- Rescheduler, migrating and initialized process: The commander needs to issue a migration command to the migrating process. Then the address and the port of the destination machine are written to a temporary file and are read by the migrating process. We defined this command as a user-defined signal, which is simple, efficient and easy to bind to most systems and communication environments. The detailed application information, parameters, and resource requirements are encapsulated in an *application schema* in a XML format and sent to the destination machine to initialize the process on the destination machine. The *application schema* contains information such as: application characteristics, which include data, communication, or computing intensive; estimated communication data size; resources requirement; and estimated execution time on workstation with certain computing power. The *application schema* is initially provided by the users and is updated according to the statistics of actual executions.
- Entities of rescheduler: We combine a custom XML based protocol with TCP/IP sockets to form the communication subsystem of the rescheduler. The XML based protocol is used for communications between the monitor, registry/scheduler and commander entities. We chose this combination because its implementation can be easily extended, its protocol is simple to implement, and it is easy to debug. As its name suggests, XML is extensible and is transmitted using plain ASCII format and it is also transport independent. Even though we have chosen TCP/IP as our transport protocol, it could be changed in the future to another communication channel, such as various channels of MPI.

4. Rule-based Decision-Making Mechanism

We established a rule to describe the requirement of the system based on one or some specific performance or availability parameters. A rule is built to define the resource status of a system. We defined a policy as a group of rules. The policy defines the transformation mechanism of hosts or resources states. We classify the system states with a fine granularity using a series of numbers to support more complex migration rules and policies. Here we use a simplified three-state representation to introduce our mechanisms, which can be easily reconfigured to a finer granularity representation.

The relationship between the actions and the states is shown in Table 1. We define the system states as:

free: The host is willing and able to accept incoming HPCM-enabled applications.

busy: The host is no longer accepting any incoming applications. It is a state of "as is". The host does not try to migrate the migration-enabled applications out.

overloaded: This host needs to offload its applications onto other host, in order to switch either to *busy* or to *free*.

Table 1. System State Description

System state	Loaded	Migrate in	Migrate out
Free	No	Yes	No
Busy	Yes	No	No
Overloaded	Yes	No	Yes

We configure a time interval as Monitoring Frequency for each state. It indicates how often the system information is to be gathered. According to the established protocol, when a host reaches an *overloaded* state in the

monitor, it consults the registry/scheduler to get a recommended candidate of destination machine. At the moment this registry/scheduler simply checks for host environments that are in the *free* state, and if there is one, it recommends it as the move-to host. The registry/scheduler then sends this message to the commander of the *overloaded* host, thus ensuing the migration.

In our system architecture, a monitoring entity resides on each host. A monitoring entity is composed of system information gathering engines, the process selector, the monitoring information database, and the rule-evaluator. Each of these modules is configurable, thus it is possible to change the internal architecture of gathering information, process selection etc. A flow representation of the monitoring architecture is shown in Figure 2. The registry/scheduler selects the process to be migrated. In the current system, we selected a migration-enabled process based on the start time of the process and the application description information provided in the *application schema* for each application. We get the estimated execution time of the application from the *application schema*, and the start time of the application from the *pid* file time-stamp. The registry/scheduler tends to migrate a process that has the latest completing time to reduce the possibility of migrating multiple processes.

A rule file contains the rule name (*rl_name)*, the command to be fired to retrieve the system information (*rl_script)*, description of the rule (*rl_desc)*, the logic operator to evaluate the rule (*rl_operator)*, a list of parameters to be passed to the shell script to retrieve the system information (*rl_param)*, and the conditions for the system to be in *busy* state (*rl_busy)*, and *overloaded* state (*rl_overLd)*. The rules are shown in figure 3.

Rule 1 [processorStatus]: Makes decisions based on the process status, i.e. the idle time of the processor, and amount of time spent in executing idle process. It uses the Unix utility 'vmstat' to determine the processor status. This rule does not require any parameter. If the processor's idle time is higher than 45 but lower than 50 then the system is kept in *busy* state; if the processor's idle time is lesser than 45 then the system is kept in *overloaded* state; otherwise the system is put into *free*.

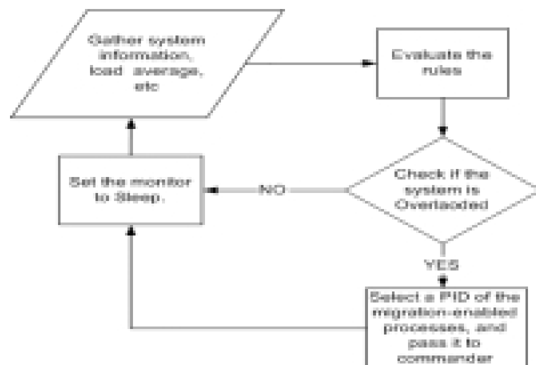

Figure 2. Rule Evaluation

Rule 2 [ntStatIpv4]: Determines the number of Ipv4 sockets currently open in the system. It uses the Unix utility 'netstat' to determine the number of sockets in a specified state. The rule takes as input a parameter specifying the state of the socket to be monitored, such as the sockets in the *ESTABLISHED* state.

rl_number: 1	rl_number: 2
rl_name: processorStatus	rl_name: ntStatIpv4
rl_type: simple	rl_type: simple
rl_script: processorStatus.sh	rl_script: ntStatIpv4.sh
rl_desc: This rule determines the processor status i.e. the idle time.	rl_desc: This rule determines the number of sockets in a give state.
rl_operator: <	rl_operator: >
rl_param:	rl_param: ESTABLISHED
rl_busy: 50	rl_busy: 700
rl_overLd: 45	rl_overLd: 900

Figure 3. Simple Rules

rl_number: 5
rl_name: cmp_rule
rl_type: complex
rl_desc: A Complex Rule.
rl_ruleNo: 4 1 3 2
rl_script: (40% * r 4 + 30% * r1 + 30% * r3) & r2

Figure 4: A Complex Rule

A complex rule evaluation determines the state of the system on the basis of a combination of rules. Figure 4 shows a Complex Rule. The *rl_type* determines the type of the rule to be complex. The *rl_script* specifies how the decision is made based on individual rules evaluated. It can be represented in an expression or a file name containing the expression. The *rl_param*, *rl_busy*, *rl_operator* and *rl_overLd* need not be specified in a complex rule.

Thus, as shown in Figure 4, rule numbers 4, 1, 3 and 2 are fired in sequence and the system is in *busy* state if both rule 2 and a combination evaluation of rule 4, 1 and 3 are in *busy* or one of them is in *busy* and the other is in *overloaded*. We can also define a complex rule as a weighted sum of several simple rules.

5. Evaluation

We implemented and tested the rescheduler working with other components of the HPCM system on a platform of 64-node cluster running on SunOS 5.8. Each workstation is a Sun Blade 100 with 1 UltraSparc-IIe 500MHz CPU, 256K L2 cache, and 128MB memory. We used the LAM/MPI [18] version 6.5.9 as the MPI-2 communication platform. The communication between the workstations is a 100Mbps internal Ethernet with exclusive use. We also used a computational intensive migration-enabled application named "test_tree", which creates binary trees with specified number of levels, assigns a random number to each node of the trees, sorts the trees and computes the sum of all the tree nodes. We

used NTP (Network Time Protocol) to synchronize the timing on workstations. The maximum error range is no more than 0.02 second.

5.1 Rescheduler Overhead

We monitor the host performance with or without the rescheduler using a standalone performance sensor, named *"sysinfo"*, for performance data collection. We configure the monitor, the commander and the registry/scheduler on one workstation. Another workstation is configured with a monitor and a commander and is registered to the registry/scheduler. Several performance parameters, including load average, CPU utilization, and communication cost, are collected. The comparisons of the measured results are shown in Figure 5 and Figure 6.

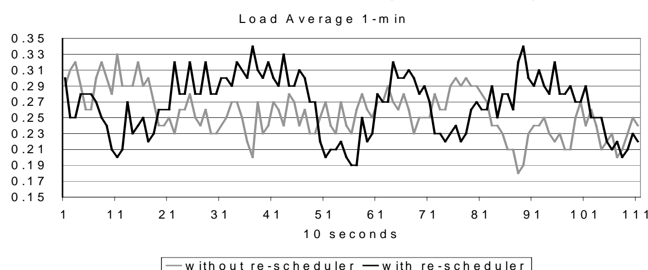

Figure 5. Overhead – Load Average 1-min

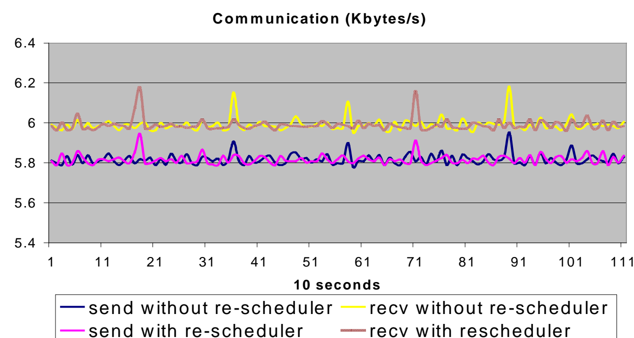

Figure 6. Overhead – Communication

The performance data is gathered at an interval of 10 seconds. The load average value is 0.256 for 1-minute without the rescheduler and 0.266 with the rescheduler. The overhead is 3.9%. The load average value is 0.262 for 5-minute without the rescheduler and 0.263 with the rescheduler. The overhead is 0.4%. The 1-minite load average is shown in Figure 5. The CPU utilization average is 0.263 and 0.260 for with and without the rescheduler and the overhead is 3.46%. The communication load with or without rescheduler is 5.82 KB/s for sending and 5.99KB/s for receiving as illustrated in Figure 6. The upper two curses are for receiving and the lower two curves are for sending. We can see clearly that there is almost no overhead for communication. Through the testing, we see that the overhead of the rescheduler operation is usually less that 4%. This testing is to explore only the overhead of the rescheduler. The overhead of process migration is also small. Details of the migration overhead on both homogeneous and heterogeneous platforms can be found at our prior publication [8].

5.2 System Efficiency

Figure 7 and Figure 8 illustrate the efficiency of our system by another experiment. The tests are performed on two workstations and the whole duration is recorded in a 10-second time interval.

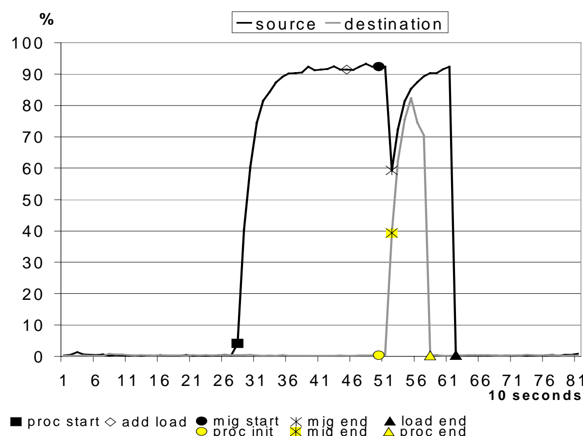

Figure 7. Efficiency – CPU Utilization

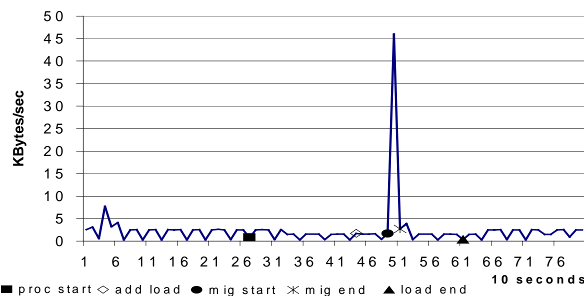

Figure 8. Efficiency – Communication

We start a migration-enabled process at the time point 28 (280 seconds from the beginning of test). We then add an additional application, which causes a dramatic load increase on this workstation and the rescheduler decides to migrate the migration-enabled process to another workstation. The migration decision is made at point 50. It takes 72 seconds, from the time that additional task is loaded, for the system to warm up and for the monitor to find out that this is a long task and determine that the system is overloaded. If the additional load is a short task, this period of time can avoid the fault migration caused by small system performance variations. It is a configurable parameter of the rescheduler and can be optimized for different type of workstations. We did not configure the system to be more sensitive because we tried to avoid false migration, which may reach a wrong decision. Then it takes 0.002 second to make a migration decision and within 0.3 seconds, the initialized process is started on the destination machine. The initialization is performed using

the LAM/MPI dynamic process management. Though the LAM/MPI dynamic process management operations are slow, currently we do not have other choice because MPICH-2 [19] and Sun MPI [20] do not support dynamic process management operations at this time. We can use other MPI-2 implementation in future to reduce the initialization time. We can also choose to improve this performance by pre-initializing the processes on the candidate destination machines. In this example, we do not use pre-initialization because we want to show clearly the entire process of the decision-making and migration. It takes the migrating process 1.4 seconds to reach its nearest poll-point. The initialized process starts data restoration and then resumes its execution within 1 second. After of 7.5 seconds, the process is migrated to another system completely, the CPU utilization drops down as shown in Figure 7, and the CPU begins to serve the addition task until it exits. Figure 8 illustrates the communication caused by the migration. The migration occurs when source machine is quite busy. The data restoration is started almost at the same time on the destination machine, and the initialized process resumes execution in parallel with the data collection and restoration. That is, the process resumes execution at the destination before the migration ends. This testing is to explore the time consuming on each phrase of process migration in decision-making and rescheduling.

5.3 Rescheduling and Policies

We defined 3 migrating policies to examine the effectiveness of the rescheduler. Table 3 compares the performance of the application under these 3 different migration policies, which are described as follows:

- Policy 1: No Migration.
- Policy 2: Migrate when any of the following conditions are met: 1) 1-min load average is greater than 2; 2) the number of active processes is greater than 150. The destination machine must meet all of the following conditions: 1) 1-min load average is lower than 1; 2) the number of active processes is less than 100.
- Policy 3: Migrate when any of the following conditions are met: 1) 1-min load average is greater than 2; 2) the number of active processes is greater than 150; 3) the current incoming/outgoing communication flow is no more than 5MB/s. The destination machine must meet all of the following conditions: 1) 1-min load average is lower than 1; 2) the number of active processes is less than 100; 3) the current incoming/outgoing communication flow is no more than 3MB/s.

We performed the tests using 5 workstations. The 1st is the source machine where the process is originally started. The 2nd is busy in communication with the 5th machine. The communication speed is from 6.71MB/s to 7.78MB/s when applying policy 2 and policy 3. The 3rd workstation has a CPU workload of 2.52. The 4th workstation is free. As shown by Table 2, for each policy, we start the same MPI application on the 1st workstation. Then additional tasks are loaded to the 1st workstation and the system becomes busy. Under Policy 1, the application does not migrate and it takes 983.6 seconds to finish. Under Policy 2, the rescheduler does not consider the communication state of each workstation. At that time the load of the 2nd workstation is 0.97, which is below the threshold, so the rescheduler chooses the 2nd workstation as the destination machine. The total execution time for the application is 433.27 seconds. Under Policy 3, the rescheduler chooses the 4th workstation as the destination machine, and the total execution time is 329.71 seconds.

Table 2. Comparison of Policies

Policy	total exec time (sec)	start at	migrate to	source (sec)	destination (sec)	migration time (sec)
1	983.6	1st	-	983.6	0	-
2	433.27	1st	2nd	242.68	198.98	8.31
3	329.71	1st	4th	221.28	115.13	6.71

The rescheduler improves the performance of an application by choosing a good destination host. In this case, the execution time is reduced to 33.5%. The migration policy of the rescheduler is very important. The communication cost is also an important factor in the decision-making. Similarly, data access locality is another important issue that should be considered in the process of decision-making. If a process involves a lot in a local data access, the process is not to be migrated for slight performance degradation. These features have been enclosed in the *application schema*. An optimized policy can greatly improve the accuracy of migration decision.

6. Conclusion

Runtime dynamic scheduling is a fundamental issue of parallel and distributed computing. In parallel computing, it is conventionally instigated by load balancing and performance optimization. In a distributed Grid environment, it becomes more essential and can be applied for fault tolerance (reschedule when the machine will shut down, intrusion is detected, etc.); resources availability (reschedule when special hardware and software are required); data locality (reschedule the process close to the data); etc. in addition to load balance. In this study, we have successfully designed and implemented a runtime rescheduling support system, which triggers rescheduling automatically, and carries the dynamic rescheduling via process migration for MPI programs. We have addressed the technical hurdles of integrating rescheduling decision-making methodology with the heterogeneous process migration mechanisms, and verified the feasibility of MPI-2 [17] and HPCM (High Performance Computing Mobility) middleware [11] in supporting runtime dynamic scheduling. With the assistance of the runtime system and the support of

HPCM, a MPI subtask, written in traditional languages such as C or Fortran, can automatically migrate from one machine to another, searching for required computing resources or for a better performance. By setting up a rule-based decision-making and scheduling mechanism, the system is extensible and flexible to various heterogeneous computing platforms. We experimentally tested the system for overhead and efficiency, as well as autonomics under MPI environments. Experimental and analytical results show that the rescheduling system works well and is a complement of existing work on dynamic scheduling, which mostly focuses on redistribution of independent new tasks instead of reschedule of running tasks.

HPCM is supported by the NSF Middleware Initiative (NMI) program and is released under NMI software release [11]. The current prototype implementation of the runtime system, as well as HPCM, is only for the proof of concept. Many issues remain open. We plan further improving the reschedule supported system with the ability of self-configuring and self-adjustment, so that the system can take feedbacks from the scheduling and performance history, and automatically improve its accuracy and efficiency. This study focuses on system and technical support to carry dynamic scheduling. It provides a system that can carry different decision-making and rescheduling algorithms, but does not intend to introduce any new algorithm. Interested readers may refer to [23, 25] for newly proposed rescheduling algorithms.

References

[1] J. H. Abawajy, "Job scheduling policy for high throughput computing environments", in the *Proceedings of 9th conference of Parallel and Distributed Systems*, pp. 605-610, Dec. 2002.

[2] J. Basney and M. Livny, "Managing Network Resources in Condor", In the *Proceedings Of the HPDC9,* pp. 298-299, 2000.

[3] A. Barak and R. Wheeler, "Mosix: An Integrated Multiprocessor UNIX", in the *Proceedings of Winter 1989 USENIX Conference*, pp. 101-112, San Diego, CA, Feb. 1989.

[4] D. Cherton, "The V Distributed System", *Communications of the ACM*, 31(3): 314-333, March 1988.

[5] K. Chanchio and X.-H. Sun, "Communication State Transfer for Mobility of Concurrent Heterogeneous Computing", in the Proceedings of *the International Conference on Parallel Processing* (ICPP 2001, Best Paper Award), Sep. 2001.

[6] K. Chanchio and X.-H. Sun, "Data collection and restoration for heterogeneous process migration", *SOFTWARE--PRACTICE AND EXPERIENCE*, 32:1-27, April 15, 2002.

[7] Frederick Douglis, "Transparent Process Migration in the Sprite Operating System", *PhD thesis*, University of California, Berkeley, Sep. 1990.

[8] C. Du, X.-H. Sun and K. Chanchio, "HPCM: A Pre-compiler Aided Middleware for the Mobility of Legacy Code", in the *Proceedings of IEEE Cluster Computing Conference*, Hong Kong, Dec. 2003

[9] Ian Foster, Carl Kesselman, "The Grid 2: Blueprint for a New Computing Infrastructure," *Morgan-Kaufman*, ISBN 1558609334, Nov 2003.

[10] N. Fujimoto and K. Hagihara, "Near-Optimal Dynamic Task Scheduling of Independent Coarse-Grained Tasks onto a Computational Grid", in the *Proceedings of IEEE International Conference on Parallel Processing*, Kaohsiung, Oct. 2003.

[11] HPCM: High Performance Computing Mobility, http://meta.cs.iit.edu/~hpcm/.

[12] Y. K. Kwok and I. Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors", *ACM Computing Surveys* Vol.31, No.4, 1999, pp.406-471.

[13] Danny B. Lange and Mitsuru Oshima, "Seven Good Reasons for Mobile Agents", *Communications of the ACM*, Vol.42, No.3, March 1999.

[14] G. Li, D. Chen, D. Wang and D. Zhang, "Task Clustering and Scheduling to Multiprocessors with Duplication", in the *Proceedings of 2003 IEEE International Parallel and Distributed Processing Symposium* (IPDPS 2003), Nice, France, April 2003.

[15] M. Lizkow, M. Livny, and T. Tannenbaum. "Checkpoint and Migration of UNIX Processes in the Condor Distributed Environment", Technical Report 1346. University of Wisconsin-Madision, April 1997.

[16] G. Manimaran and C. Siva Ram Murthy, "A Fault-Tolerant Dynamic Scheduling Algorithm for Multiprocessor Real-Time Systems and Its Analysis", *IEEE Transactions on Parallel and Distributed Systems*, vol 9, No. 11, p137, 1998.

[17] The Message Passing Interface (MPI) standard, http://www-unix.mcs.anl.gov/mpi/.

[18] LAM/MPI Parallel Computing, http://www.lam-mpi.org.

[19] MPICH2, MPI-2 Home Page, http://www-unix.mcs.anl.gov/mpi/mpich2/.

[20] Sun MPI-2, Sun MPI 6.0 Software Programming and Reference Manual. http://www.sun.com/products-n-solutions/hardware/docs/html/817-0085-10/index.html

[21] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh, "The Design and Implementation of Zap: A System for Migrating Computing Environment", in the *Proceedings of the 5th Operating System Design and Implementation OSDI'02*, Dec. 2002.

[22] P. Smith and N. Hutchinson, "Heterogeneous Process Migration: The Tui System", *Software -Practice and Experience*, Vol 28, No.6, pp.611-639, 1998.

[23] X.-H. Sun and M. Wu, "GHS: A Performance Prediction and Task Scheduling System for Grid Computing," in the *Proceedings of 2003 IEEE International Parallel and Distributed Processing Symposium* (IPDPS 2003), Nice, France, April 2003.

[24] M. M. Theimer and B. Hayes, "Heterogeneous Process Migration by Recompilation", in Proceedings of *the 11th IEEE International Conference on Distributed Computing Systems*, Jun. 1991.

[25] M. Wu and X.-H. Sun, "A General Self-adaptive Task Scheduling System for Non-dedicated Heterogeneous Computing", in the *Proceedings of IEEE Cluster Computing Conference*, Hong Kong, Dec. 2003.