G-JAVAMPI: A GRID MIDDLEWARE FOR TRANSPARENT MPI TASK MIGRATION*

LIN CHEN†, TIANCHI MA†, CHO-LI WANG†, FRANCIS C.M. LAU† AND SHANPING LI‡

Abstract. Resources in a grid are dynamic, heterogeneous, and widely distributed. End users need a simple and efficient way to aggregate and utilize these diverse resources. We introduce a grid middleware called *G-JavaMPI*, which combines a high-level message passing interface with the Java language to support portable messaging-passing programming in a grid. Different from traditional MPI implementations, it supports transparent migration of MPI processes during execution. This feature facilitates more flexible task scheduling and more effective resource sharing. The migration mechanism is implemented by exploiting the JVM Debugger Interface (JVMDI) functions, and minor bytecode modification. This method is portable and does not require modification of the JVM. To guarantee continuous MPI communication during process migration, a message redirection mechanism is employed. As a process could be moved across multiple grid points that are under different control policies for cross-organization resource sharing, we develop an instance-oriented delegation mechanism that can provide strict protection on multi-hop delegations. We use a BLAST application to evaluate the performance of process migration and the effectiveness of dynamic task scheduling. The results attest to the effectiveness and efficiency of the middleware.

Key words. grid, middleware, JavaMPI, process migration, delegation, scheduling.

1. Introduction. *Grid*, or *computational grid*, is evolving rapidly as a practical extension to distributed computing technology, with the vision of dynamic and diverse resource sharing across organizations [1]. Resource sharing over a grid helps existing computing resources to be utilized in a more cost-effective manner, provides ways to solve large-scale problems requiring an enormous amount of computing power, and introduces flexibility in resource coordination.

Grid resources are usually distributed widely and managed under different local policies. The types of resources in a grid can be highly diverse and their availabilities can change dynamically. A new grid middleware is therefore needed, to hide the heterogeneity of the computing resources and to bridge the gaps between different local policies. The grid middleware should support easy programming and be able to adapt the applications to the changing environment for efficient execution.

This research aims at the design of a grid middleware called *G-JavaMPI*, which can support parallel execution of MPI (Message Passing Interface) programs written in Java, and location-transparent computations in a grid. It combines the platform portability of Java with the easy programming of the message-passing paradigm. Using G-JavaMPI, the programmer can write and execute an MPI program without having to care about the locations of computing resources and the network topology. Different from many existing MPI implementations, G-JavaMPI supports transparent Java process migration and message redirection between "mobile" distributed Java processes. Our portable migration mechanism exploits the JVM Debugger Interface (JVMDI) functions to control the execution and capture the runtime states of Java process. It restores the process states through an exception handler which is inserted in the Java source code at the preprocessing time. The message redirection mechanism makes the physical locations of the processes transparent to the user by supporting

*This work was supported in part by HKU Foundation Seed Grant 28506002, the China 863 National Grid project, and a HKU grant for the HKU Grid Point.

†Department of Computer Science, The University of Hong Kong, Hong Kong (`Contact person: Lin Chen (lchen2@cs.hku.hk)`).

‡College of Computer Science, Zhejiang University, China (`shan@cs.zju.edu.cn`).

logical ranks in the program and redirects buffered messages to updated location of the migrating process during process migration.

Another special feature of G-JavaMPI is its instance-oriented delegation mechanism for supporting multi-hop process migration, in which a process could be migrated for multiple times across grid nodes. With our instance-oriented delegation mechanism, the user grants his/her privileges via a security instance instead of the hosts. The security instance contains the description of the resource access operations, the conditions under which the process can perform these operations, as well as a signature of the user to certify the above contents. Permission to access the resource in the destination host can then be granted by simply checking the signature in the security instance and the validity of the specified resource access operations. Whereas in existing delegation mechanisms, the destination host has to verify all the signatures recorded in the delegation document which were created during migration of the process through a series of hosts.

With these supports, a process can be migrated transparently and safely between grid points to avoid running hotspots, to utilize available resource or to move closer to the data source. The process migration feature also gives the administrator certain flexibility in managing and deploying execution resources in response to load imbalances and fluctuations. As the result, better resource utilization can be achieved in the grid environment, which makes it possible to solve large-scale problems more cost-effectively.

The rest of the chapter is organized as follows. Section 2 gives an overview of the design of the middleware. Sections 3 and 4 describe our solutions to JavaMPI binding and transparent Java process migration respectively. Section 5 introduces the instance-oriented delegation mechanism. Section 6 presents the experiment of dynamic scheduling of a BLAST application and the performance results. Section 7 discusses the related work. Section 8 concludes the chapter.

2. The G-JavaMPI Middleware. Figure 2.1 displays the main components of G-JavaMPI in a single grid node, and their interactions. The components on the left side, including the migration modules and the MPI daemon, mainly perform execution-related functions. Modules on the right are for decision-making and control, including the rescheduler module and the information collection module. The delegation module sits in the middle to provide protection and security related information to the other modules.

Four main operations make up the execution and scheduling of an application: (1) JavaMPI binding; (2) process migration; (3) instance-oriented delegation; and (4) rescheduling. The following discusses the role of each of the modules concerned, and how they interact with one another to carry out the four operations.

Initialization and MPI communication During initialization, the local MPI daemon collects information about the processes from remote daemons, creates the MPI communicator, and then ranks the processes (labelled (1) in the figure). At the same time, the security module generates for each process a delegation document which is to be used to prove the identity and to record the migration history of the process (2). During MPI communication, the daemons check the location of the destination process according to the MPI communicator and redirect messages to it (3).

Process migration At start time, Java processes are augmented with library code for state capturing. When migration takes place, the library code suspends the execution at a migration safe point and captures the process state through

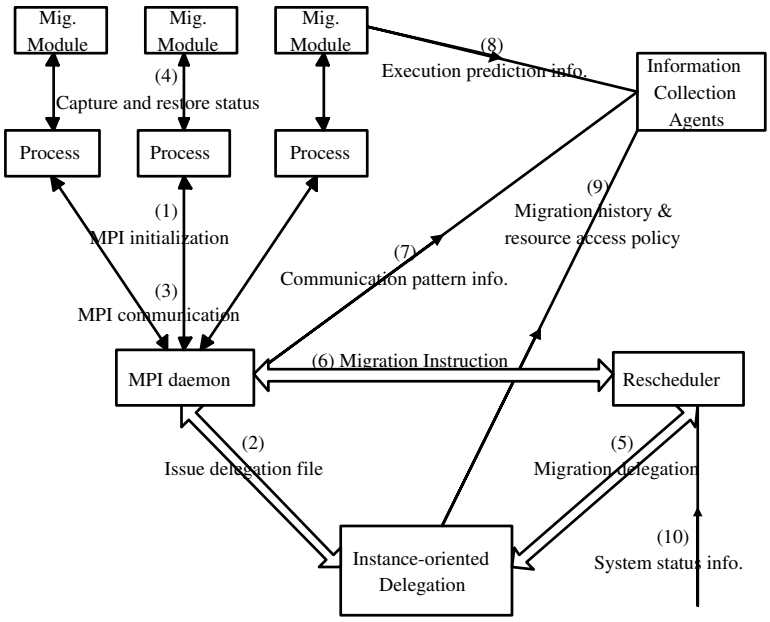

FIG. 2.1. *Main components of G-JavaMPI middleware*

JVMDI (4). The delegation module records the event of migration and appends it as a new entry to the trace list in a delegation document. The restoration exception that was inserted into class source code at preprocessing time is invoked to read the process states from the process dump and restore them into the new process (4). If migration is successful, all the concerned entities are required to sign on the recorded migration event in an entry of the trace list in order to prove the validity of this event; otherwise, the entry would be deleted (5).

Migration control Migration control is mainly done by the rescheduler. The rescheduler needs to check with the security module whether a process is allowed to migrate, whether the destination host is in the trust list, etc. (5), before it can issue the instruction to the remote daemon in the destination host (6). The daemons involved will update the MPI communicator structure, and communication between the migrated process and other processes can be resumed (1).

Rescheduling decision The rescheduler makes decisions based on information about application performance and the system status. Three types of application performance information are collected by the information collectors: communication pattern records from the daemons (7); execution prediction information from the migration module (8); and migration history and resource access policy information (such as the trusted hosts list, etc.) from the security module (9). System status information including CPU utilization and network availability is obtained by the rescheduler (10).

3. MPI Communication.

3.1. Enabling MPI Communication in Java. We opt for a client-server design for transferring MPI messages for Java processes. MPI daemons are running

as servers in all the grid nodes; the Java processes are clients. We follow the MPI standard to implement the JavaMPI API. The API works as an interface to link Java processes with the MPI daemons which run the native MPICH-G2 library. A Java process sends MPI messages to the local MPI daemon by calling the API functions. Several message queues are reserved by the API functions to transfer requests and replies between clients and servers. The message data, which are usually arrays of primitive data types, are packaged into byte streams using the object serialization mechanism. In addition to primitive data types, messages with Java object types can also be transferred. The daemon then delivers the messages on behalf of the process. In the destination host, the bytes are transformed back to the original data of primitive or object type.

In order to provide efficient MPI communication, native MPI is used by the MPI daemons to transfer messages. Instead of tightly binding the native MPI library to the Java language as is done in some previous JavaMPI implementations such as mpiJava [2] and JavaMPI [3], our client-server design clearly separates the JVM execution environment from the environment in which the native library is executed. This approach avoids conflicts on the use of system resources (e.g., conflicts on the system signals) between the native MPI library and the JVM. It also requires no modification to the native MPI library.

3.2. Message Redirection and Restorable Communication. The MPI daemons maintain a table called the global communicator mapping table. The table records information of all the processes including their logical ranks in the MPI communicator and physical locations. When a process is migrated from one location to another, the local daemon serving the process notifies the other daemons of the change of physical location. All daemons including the local daemon update the mapping table. The communication channels are reconstructed via the global communicator mapping table at the low level. Although its physical location changes after migration, the process can still be reached as before through its logical rank number.

4. Transparent Java Process Migration.

4.1. State Capturing Through JVMDI. The Java Virtual Machine Debugger Interface (JVMDI) [23] is a native interface available since the introduction of Java 2, and is used typically by debuggers. It defines the standard services that a JVM must provide for debugging. Table 4.1 summarizes the JVMDI functions used in process migration, especially in state capturing. The first group of functions are used to control thread execution and detect events such as method exit/entry, frame pop-up and breakpoint. The other functions are used to capture the runtime state of loaded classes (including their fields and methods), threads, objects, frames and local variables.

In G-JavaMPI, the migration module is implemented as an external library. It is loaded into the JVM at startup and runs as a debugging thread. When the migration module receives a migration instruction, it suspends the execution of the Java process. All information related to loaded classes and their static fields are saved in bytecode format using object serialization. All frames are saved in the order that they are invoked. For each frame, local variables, referenced objects, information of associated method, and the program counter are captured and saved. All the actions performed by the migration module are transparent to the application. In addition, the capturing mechanism is all on top of an ordinary JVM so that no modification of the JVM is required.

TABLE 4.1
Some JVMDI functions used in process migration

Functions	Descriptions
SuspendThread(), ResumeThread()	control the execution of threads
SetEventHook(), SetEventNotificationMode()	set events to be detected and the actions responding to the events, mainly the JVM initialization event and breakpoint event
SetBreakpoint(), ClearBreakpoint()	set and clear breakpoints in migration safe points and starting points in invoked methods
GetLoadedClasses()	gives all loaded classes in JVM
GetSourceFileName(), GetClassModifiers()	get the source file name to check whether it is main class, get access flag to check it is static class
GetClassFields()	get static fields in a class to be captured and restored
GetFieldName(), GetFieldModifiers()	get the access flag of field to check whether it is static, get its name to restore its value
GetMethodName(), GetMethodModifiers()	get the name and access flags of invoked method which are used to set breakpoint during restoration
GetLineNumberTable()	get line numbers to check whether the current breakpoint is at the start location of a source line (migration safe point)
GetLocalVariableTable()	get list of local variable of a method to be used during capturing and restoration
GetBytecodes()	get bytecode to inspect possible branches for searching for next migration safe point
IsMethodNative()	check whether the invoked frames belong to native methods which are not migrate safe point
GetAllThreads()	get all threads, get reference to main thread
GetThreadStatus(), GetThreadInfo()	check whether the thread is daemon thread which will not be suspended during migration
GetFrameCount()	get the number of invoked frames
GetCurrentFrame(), GetCallerFrame()	get latest invoked frame and its caller frame, and the references to all frames can be obtained
GetFrameLocation()	get the location in the method code where the execution reaches in the frame
GetLocalObject(), GetLocalInt(), ...	get the value of local variables with various types such as Int, Object, Long and etc.
SetLocalObject(), SetLocalInt(), ...	restore the value of local variables

There are no JVMDI functions for extracting and rebuilding operand stacks. Moreover, when the execution point is inside a native method (frame), the local data in the frame are machine-dependent. These factors make it difficult to capture and restore the complete process state and may destroy the portability of the system. Our solution is to restrict a migration to happen only when all operand stacks are empty and the execution point is outside of a native method. This is achieved through setting migration safe points and bytecode rearrangement.

Migration safe points are defined only for the current frame. The current frame must be a Java method other than a native method, and no ancestor frame which invokes the current frame directly or indirectly is a native method. With these conditions, the migration safe points are the execution points at the first bytecode in-

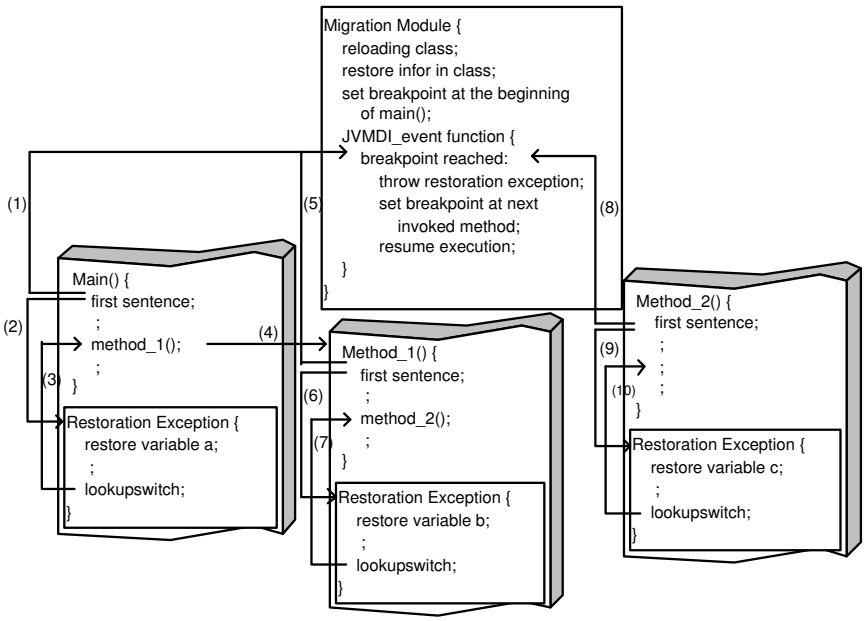

FIG. 4.1. *Frames and runtime states restoration*

struction of each source code line in the current frame. This definition is based on the observation that the operand stack of the current frame is always empty immediately after the completion of the execution of a Java source code line. If the process is suspended at a point which is not a migration safe point, the execution will be resumed immediately and the migration will be delayed until reaching the next migration safe point.

For the operand stacks of all the frames other than the current frame, bytecode rearrangement is used to make these operand stacks always empty during migration. The rearrangement is done in the preprocessing stage before execution. It introduces new local variables to store the intermediate values. Consider for example the statement $y = f(x) + g(x)$. During the valuation $f(x)$, the intermediate value of $f(x)$ is stored in the operand stack. To prepare for possible migration, this is transformed into: $tmp1 = f(x)$; $tmp2 = g(x)$; $y = tmp1 + tmp2$. Here is another frequent statement: $y = f(a + g(x))$, which is transformed into: $tmp1 = g(x)$; $y = f(a + tmp2)$. Therefore, intermediate values are stored in local variables rather than the operand stack.

4.2. State Restoring. During restoration at the destination node, the migration module reloads all the saved classes and restores their static field values by calling several JNI functions, including $FindClass()$, $SetStaticIntField()$, $SetStaticObject Field()$. To restore the runtime state, we resort to the exception handler. During bytecode rearrangement at preprocessing time, restoration functions are inserted as an exception handler in the exception area of each method. In the handler code, local variables of the called methods are reset with the saved information, and a "lookupswitch" instruction would pass the execution to where the process was suspended previously.

Figure 4.1 outlines the procedure of re-establishing stack frames and restoring

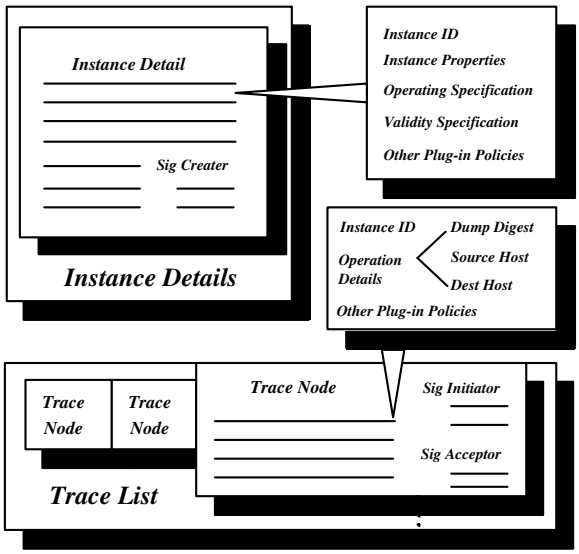

FIG. 5.1. *Delegation document*

runtime state. The migration module in the destination host has a breakpoint at the start of *main*() method. The breakpoint is reached immediately after the execution begins (1). The migration module then throws a restoration exception in the current method. If there are other frames, it sets a breakpoint at the start of the next invoked method (frame). The process catches the exception immediately, and jumps to the restoration exception handler (2). The runtime state of the current method is then restored. The process then continues execution from the suspension point and invokes the next method *method*_1() (3). The second frame is created (4). The restoration of its runtime state is the same as the *main*() frame (5)(6)(7). These steps are repeated until the final frame is restored (8)(9)(10).

5. Security Support During Migration.

5.1. Instance-oriented Delegation Mechanism. To tackle the security issues arising from multiple organizations, the Grid Security Infrastructure (GSI) [18] has established a trust framework based on Public Key Infrastructure (PKI) in which general authentication and authorization can be carried out. The user grants his/her privilege to the hosts. To ensure the trust protection, when a process is about to move, the host currently hosting the process will be asked to delegate the process' privilege to the new host to which the process will move. Because authorization is bounded to hosts, this is called host-oriented delegation. Therefore, to authenticate the privilege in a multi-hop migration, the destination host has to verify the signatures recorded in the delegation document which were issued by all the hosts along the process's migration path. This chain-delegation in fact is not most desirable because if a privileged host is cracked, the damage can easily spread across the whole network. Moreover, there is significant overhead in the authentication of chain-delegation.

To allow for more efficient and safe delegations, we introduce the instance-oriented delegation mechanism [17] which is also based on PKI. A security instance is an encapsulation of a set of authorizations from the user for resource access at the destination host and the relevant conditions for validating these authorizations. The conditions

can be created with respect to the process' specific code segments, states or resource requests. Therefore the user has the flexibility of defining different priorities or policies on processes, which have to do with resource requirements, maximum handover time, etc. The user can then delegate his/her privilege to a security instance instead of to certain hosts. Access privilege at the destination host can be approved by only verifying one single signature issued by the user in the security instance and validating the authorizations according to the specified conditions. This avoids having to verify all delegation signatures in a chain-delegation, and so the authentication procedure is greatly simplified. To support process migration, the security instance is specified to protect process state information including code and runtime states and the user's privileges.

The security instance is embedded in the delegation document, as shown in Figure 5.1. The first part of the delegation document records the instance details, including the identity of the instance, instance properties such as process code, operation specification such as its policy on trusted and untrusted resources, and the validity specifications such as the maximum number of handovers, etc. Handover is the operation of transferring the delegation document from a source host to a destination host during process migration. To prevent unlimited diffusion of any potential damage, the maximum number of handovers should be specified. The second part is a trace list, which records the history of handovers. Each entry in the trace list records the operation details including a digest on the process states, source host, destination host, etc. The instance ID is also recorded to prove that the operation is relative to the exact instance. The initiator and acceptor should sign on the operation, to show their approval on the entry's content. For the handover operation, the initiator is the source host and the acceptor is the destination host. Both the initiator and the acceptor will hold a copy of the signed record in the entry to prevent any denial on the operation in future. The handover history in the trace list is an important reference for trace back purposes should security leaks happen in the future.

5.2. Process Migration in Action. Figure 5.2 shows all the essential operations and communications during the entire migration procedure. Before migration, the delegation document is checked whether the proposed migration is allowed. For example, if the maximum handover time is reached or the destination host is not in the trusted list, the migration will be denied. If the migration is permitted, the rescheduler issues a migration instruction to the MPI daemon. The Java process will find the next migration safe point and suspend the execution at that point. The local daemon is notified and it in turn notifies all the remote daemons. The destination node creates a new process instance (a new JVM) that waits, when at the same time the source process captures the process states. When capturing is finished, a digest which is an array of 16 bytes is produced from the captured process states and reported to the delegation module. The delegation module creates a new entry describing this migration event based on the digest and appends it to the trace list in the delegation document. The process states are then transferred to the destination node. Before the new process restores the suspended process states to the new process, the delegation module computes the digest of the received process states and compares it with that recorded in the trace list of the delegation document. When restoration finishes, the delegation module signs on the entry in the trace list to prove the validity of this migration event.

6. Experimental Results. We divide the experiments into three parts: performance of MPI communication, performance of process migration, and effectiveness

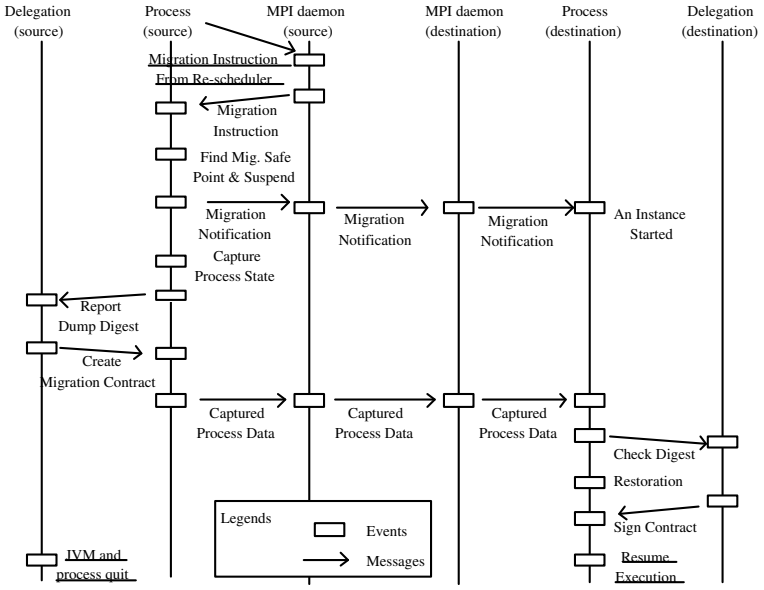

FIG. 5.2. *Delegation procedure during migration*

of scheduling support. The version of G-JavaMPI we use in the experiments is implemented based on: (1) Java Development Kit: Sun JDK 1.4.2_04; (2) MPI library: MPICH-G2 [6]; (3) Grid toolkit: Globus Toolkit 2.4 [22]. The experiments are conducted on a cluster of PCs, each equipped with a Pentium 4 2.0 GHz processor and 512 MBytes memory, running Linux 2.4.18. Java programs are executed in full-speed debugging mode provided by Sun Java Hotspot JVM in Sun JDK1.4.2_04.

We collect experiment data from the execution of a G-JavaMPI BLAST (Basic Local Alignment Search Tool [20]) application that realizes a distributed bioinformatics computation. We set up a simulated grid environment consisting of two grid points, which is sufficient to demonstrate the features of G-JavaMPI. Each grid point has some computational machines and a copy of the gene database. The two groups are denoted as A (machines 0, 1, 2, 5; database 0) and B (machines 3, 4, 6; database 1).

6.1. Performance of MPI Communication and Process Migration. The performance of MPI communication, shown in Figure 6.1 and 6.2, is evaluated by benchmarking the time of ping-pong message transfer between two nodes. The steady bandwidth for long message in G-JavaMPI is about 6.6MB/s. Comparing with MPICH-G2's 11.3MB/s, this represents a loss of 42% in efficiency. The additional overhead is mainly caused by message packaging and copying performed during client-server message transferring in the JavaMPI binding.

We investigate process migration overheads in the BLAST application under different CPU workloads. We use a collection of computation-intensive calculating-PI programs with different sleeping times and calculation precisions to produce various background workloads on the computational machines. The value of background workload is taken as the CPU utilization when the background program is executed alone in a machine. The various types of workload used in the experiments are 98%, 70%, 50% and 0% CPU usage. The detailed timing breakdowns during migration

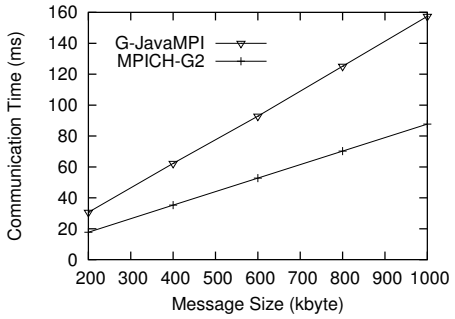

FIG. 6.1. *MPI communication times of long messages in G-JavaMPI and MPICH-G2*

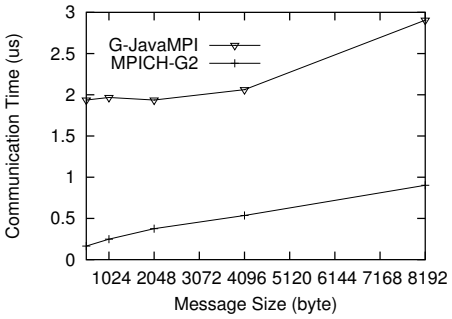

FIG. 6.2. *MPI communication times of short messages in G-JavaMPI and MPICH-G2*

under different background workloads are recorded in Table 6.1. The total migration cost consists of three parts, "Migration-Out", "Dump-Transfer"and "Migration-In". In "Migration-Out", the "Pre-Migration" operation includes those operations that must be done before migration, such as closing files or socket connections. For our experimental application, the gene files must be closed before migration can be carried out. "Dump" is to extract process state through the migration module of G-JavaMPI and record it in a dump file. The size of a dump file in our BLAST experiment is about 2.1 Kbytes. The "Generate-Delegation" operation is to record the migration event in the contractual history and append it to the process' delegation file. After "Migration-Out" is finished, the dump file will be transferred to the destination machine, and the operation is called "Dump-Transfer". The "Migration-In" part consists of four operations. In the "Entrance-Check" operation, the identity in the delegation of the source JavaMPI process and its resource access privilege are verified. Then the process state is restored through G-JavaMPI's process restoration mechanism in the "Restore" operation. In the "Post-Mig" operation, some files may need to be reopened and socket connections re-connected. Finally, if the migration is completely successful, the contractual history in the delegation document will be signed by all concerned entities to prove its legality during the "Sign-Delegation" operation.

As Table 6.1 shows, the overheads of all the operations during migration are not significant (from several milliseconds to hundreds of milliseconds). The time of "Dump-Transfer" may vary with the available network bandwidth and the size of the dump file. In our experimental environment, the total migration cost is less than 3.5 seconds, even with very high background workload. The overhead is acceptable for most long-running applications.

Table 6.2 presents the percentage of overhead due to delegation-related operations in the total overhead. The delegation-related operations include the "Gen-Dele", "Entrance-check" and "Sign-Dele" operations. Their costs are about 15.9%–33.6% of the total overhead. The overhead of delegation operations is almost independent of the scales and characteristics of grid applications, because there are only a fixed number of operations on the delegation document. As the number of migration hops increases, the size of the delegation document increases because new entries are being appended. This may increase the time spent by the operations. But the amount increased is not significant as the appended trace data is of very small size. Therefore the weights of the delegation operations would tend to diminish in applications of large problem sizes, which are very common among grid applications. It can be amortized

TABLE 6.1
Migration cost breakdowns (ms)

Background Workload	Migration-Out			Dump-Transfer	Migration-In			
	Pre-Mig	Dump	Gen-Dele		Entrance-Check	Restore	Post-Mig	Sign-Dele
0%	3	194	61	349	189	63	6	61
50%	3	194	78	547	187	62	6	62
70%	3	404	55	844	192	65	6	72
98%	3	740	258	1909	194	62	6	63

TABLE 6.2
Delegation costs (ms)

Background Workload	Delegation Total Cost		Migration Total Cost	Delegation Percentage
	In Migration-Out	In Migration-In		
0%	61	251	931	33.6%
50%	78	250	1,144	28.8%
70%	55	265	1,645	19.5%
98%	258	257	3,239	15.9%

by the migration benefits.

6.2. Dynamic Scheduling. The BLAST application adopts the database segmentation approach, i.e., the database is divided into many independent segments and each process compares the query sequence against the individual segments. As the segments are independent, there is little communication between processes. The scheduling of this application is naturally an instance of the classical scheduling problem of mapping independent tasks to heterogeneous machines. In related literature [21], either iterative heuristics or searching-based algorithms can be used to solve this problem. We select the min-min heuristic (Min-min) and genetic algorithm (GA) in our experiment. The estimated completion time (ETC) of a process in a computational node is calculated as $ETC = \frac{TotalLoad - FinishedLoad}{ProRate} + MigCost$. An ETC matrix records the ETCs of all pairs of processs and nodes. The algorithms take the matrix as input and calculate the new mapping solution.

The experiment concurrently runs three BLAST programs: X (8 processes, each searching about 172MB data), Y (6 processes, each searching about 228MB data), and Z (4 processes, each searching about 344MB). We evaluate and compare the execution performance of these three applications with different scheduling algorithms. The following scenario is used to test the performance.

Stage 1 At the beginning, machine 5 of grid point A and machine 6 of B are not available to compute. Each of the 18 processes is mapped to one of the five machines which are ranked in a predefined order.

Stage 2 About 60 seconds later, the system is notified that machines 5 and 6 join the grid and can provide computational resources. This event is called extension. The scheduler will make decisions for process re-mapping.

Stage 3 About 300 seconds later, machines 5 and 6 leave the grid and processes executing there must be migrated. This event is called shrink. The scheduler will make decisions for process re-mapping again.

In the above formula for estimating ETC, $MigCost$ is obtained from experimental result. In reality, however, a large amount of simultaneous migrations may cause some additional overhead. Part of the overhead is the management cost of the daemon which has to do information update and process management. As migrations involving the

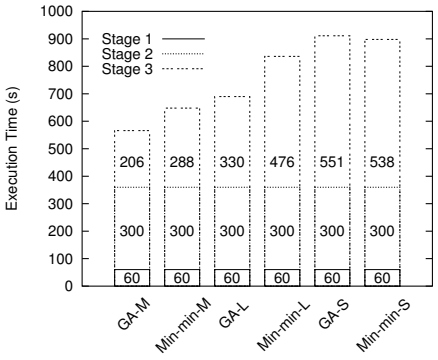

FIG. 6.3. *BLAST execution time breakdowns with different scheduling algorithms*

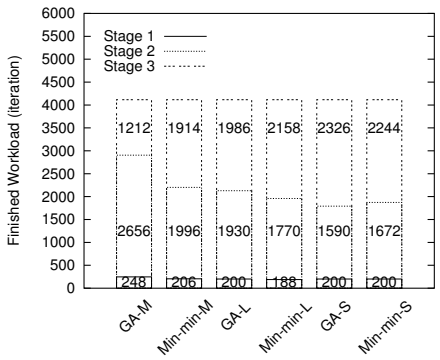

FIG. 6.4. *Finished workload distributions of BLAST execution*

same machine cannot be overlapped for data consistency reasons, the other part of the overhead is caused by the sequential execution of certain migrations. In order to truly reflect the migration overhead and also to avoid excessive migrations, we introduce cost penalty to the ETC matrix. For the case of extension, penalties are added to the ETCs of processes on machines which are not newly added. The ETCs of processes on their own machines are not added the penalties as there are no migration costs. And the ETCs of processes on newly-added machines would not be added the penalties because their migrations are desired. We apply a similar policy to add penalties for the case of shrink. Penalties are added to the ETCs of processes on machines which will not be removed. The ETCs of processes on their own machines are not added the penalties. Penalties are not added to the ETCs of processes which originally resided on removed machines. Finally the ETCs of processes on removed machines should be set to a very large value because processes are not allowed to migrate there after a shrink. By adding different penalties, we get six different scheduling algorithms including GA-M (GA algorithm with moderate penalty), GA-L (GA with large penalty), GA-S (GA with no penalty), Min-min-M (Min-min algorithm with moderate penalty), Min-min-L (Min-min with large penalty) and Min-min-S (Min-min with no penalty).

Figure 6.3 and 6.4 report the execution times and finished workloads of the three applications with different scheduling algorithms in three stages. The amount of workload is presented by the number of finished iterations. One iteration of workload is equivalent roughly to searching 1Mbyte database. In stage 1, represented as the bottom box, the execution times are fixed to be 60 seconds. As the mappings are the same, the finished workloads are almost the same for all the scheduling algorithms. In stage 2, the processes are re-mapped after the extension event is triggered. Although the execution times are also fixed to be 300 seconds, the finished workloads are different. The applications using the GA-M and Min-min-M algorithms finish more workloads than the others. In stage 3, the processes are re-mapped again after the shrink event is triggered. The applications with GA-M and Min-min-M finish earlier than the others. This is because they had less remaining workloads and they get better mappings. Based on the execution times and finished workload, we can obtain the execution efficiencies in each stage. In stage 2, applications using GA-M have the highest efficiency of searching about 9 Mbytes per seconds. They also achieve the highest average efficiency (more than 7 Mbtyes per seconds) of the whole execution. For the whole execution, GA-M and Min-min-M make the applications run fastest.

TABLE 6.3
Number of migrations

Algorithm	GA1	Minmin1	GA2	Minmin2	GA3	Minmin3
After Extension	6	4	4	4	13	8
After Shrink	7	7	4	4	7	12

However, GA-S and Min-min-S make the execution worse as many migrations produce much additional overhead which is not estimated in the algorithms. In the applications with GA-L and Min-min-L, migrations only happen between the two newly added machines and other machines, and the workload actually is not fully balanced among the machines. Therefore they experience worse performance than GA-M and Min-min-M. Table 6.3 records the number of migrations in each remapping. Algorithms with no penalty cause as many as 20 migrations in total. Algorithms with large penalty allow the processes to move to or from machines 5 and 6, and therefore the number of migrations is only 8. In addition to some necessary migrations to newly added machines, algorithms with moderate penalty also relocate some processes in the other machines to achieve better load balancing. The number of migrations for GA-M and Min-min-M are 13 and 11 respectively. Both are less than those of GA-S and Min-min-S, and larger than those of GA-L and Min-min-L.

Through these experiments, the migration facility of G-JavaMPI has been tested and verified. With the mapping algorithms, the efficiency and flexibility of applications are improved through dynamic process migration. In addition to the algorithms, a practical cost model which accurately measures factors affecting the performance is very important. In our case, the introduction of additional penalties augments the overhead estimation in the ETC matrix, which leads to improvements in the performance reported.

7. Related Work. There have been efforts to provide MPI for the Java language. Existing approaches can be grouped into two types. One is using native MPI bindings where some native MPI library is called by Java programs through Java wrappers. Examples include mpiJava [2] and JavaMPI [3]. The other approach is pure Java implementation as used in jmpi [4] and DOGMA [5]. Our method can be classified into the first group. However, instead of tight integration, our method seperates the functionalities of native MPI and Java wrappers. Native MPI is realized in MPI daemons which cooperate with Java wrappers through inter-process communication (IPC). Therefore, G-JavaMPI is portable to different native MPI implementation, and it can support communication restoration with the help of daemons when the process is migrated.

There is also research on making existing MPI implementations grid-enabled or grid-aware through modification or the addition of special features. MPICH-G2 [6] and MagPIe [7] are some existing implementations optimized for grid environments through modifications such as network optimization, topology-aware collective operations, etc. In [8], an add-on mechanism to the MPI paradigm called MPI process swapping is presented. Through adding some code to iterative MPI applications and allocating redundant MPI processes, it can improve performance by dynamically swapping processes to the best available resources. However, there is no implementation that provides support for transparent MPI process migration.

There are grid middleware or frameworks developed to provide useful abstractions to the grid programmer. Several systems, including Netsolve [9] and Ninf/G [10], facilitate the whole or parts of the NES/GridRPC (Network-Enabled Server/Grid-based

RPC) paradigm which supports server-client programming on grid. The Cactus code and computational toolkit [11] provides application programmers with a high level set of APIs which hide features such as the underlying communication and data layers. During runtime, the best available layer, which is implemented in modules, can be chosen for a give resource. Cactus modules cover almost all aspects including compilation, programming, I/O, parallel checkpointing, dynamic steering, etc. DataCutter [12] is an application framework that supports the development of data-intensive applications requiring access to remote databases. Based on its filter-streaming programming model, the application's processing structure is implemented as some distributed processes called filters, with which queries and data transformations can be carried out.

There are many research projects targeting at the development of application schedulers for dynamic and heterogeneous environments. Most of them would tie applications to specific programming models, and derive a performance model from the programming model to guide the scheduling decisions. The programming model ranges from task dependency graphs in SEA [13] and communicating tasks having their own resource requirements in AppLeS [14] to the master-slave model in [15]. AppLeS has little limitation on the programming model. Instead, it provides some templates for users to specify the performance model. For different performance models, there are different scheduling algorithms that would fit, including iteration-based distribution in MARS [16], maximum-flow algorithm in [15], and many other heuristics. There are two unique features in our G-JavaMPI. First, it targets at a broad spectrum of applications which can be implemented using the MPI paradigm. The extracted models include independent subtasks, master-slave, tasks with communication and dependency, and resource-centric applications. Second, it focuses on rescheduling in space, while most of the other related projects consider only initial scheduling.

8. Concluding Remarks. We aim at developing a grid programming environment which can achieve easy-of-use, high performance, and flexibility in task scheduling for grid computing. We introduce a new grid middleware called *G-JavaMPI* to support programming and optimized execution of JavaMPI applications in the grid environment. The middleware enables people to write MPI-style programs in the Java language conveniently. The transparent Java process migration mechanism and the delegation mechanism provide a basis for developing and realizing dynamic scheduling. Through the experiments, we have demonstrated the usefulness of process migration and dynamic scheduling in extension and shrink scenarios. There are however other types of grid applications, such as tightly cooperative applications, community-centric applications, and interaction-centric applications, that have yet to be examined for their MPI programmability or rescheduling strategy. Some tools or methods to recognize the application characteristics may be extremely useful. What is also worth some serious study is the issue in integrating migration-based rescheduling with other system-level or application-level services including time-based scheduling, QoS mechanisms, etc. To integrate the middleware in existing grid infrastructures, a standard interface needs to be developed to facilitate easy and efficient access to G-JavaMPI.

REFERENCES

[1] I. Foster and C. Kesselman, *The Grid 2: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers, 2004.

[2] M. Bake, *mpiJava: A Java interface to MPI*, The 1st UK Workshop on Java for HKCN, 1998.

[3] Sava Mintchev and Vladimir Getov, *Towards portable message passing in Java: Binding MPI*, In Recent Advances in PVM and MPI, volume 1332 of Lecture Notes in Computer Science, pages 135-142. Springer Verlag, 1997.

[4] K. Dincer, *A Ubiquitous Message Passing Interface Implementation in Java: jmpi*, The 13th International and 10th Symposium on Parallel and Distributed Processing, 1999

[5] Glenn Judd, Mark Clement and Quinn Snell, *DOGMA: Distributed Object Group Metacomputing Architecture*, Concurrency: Practice and Experience, 10(11/13):977-983, 1998.

[6] Nicholas T. Karonis, Brian Toonen, Ian Foster, *MPICH-G2: a Grid-enabled implementation of the Message Passing Interface*, Journal of Parallel and Distributed Computing, Volume 63, Issue 5, May 2003

[7] T. Kielmann, R.F.H. Hofman, H.E. Bal, A. Plaat, and R.A.F, *Bhoedjang: MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems*, ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), Atlanta, GA, pp. 131-140, May 1999

[8] O. Sievert, H. Casanova, *A Simple MPI Process Swapping Architecture for Iterative Applications*, to appear in the International Journal of High Performance Computing Applications (IJHPCA), Fall issue, 2004.

[9] H. Casanova, J. Dongarrg, *NetSolve: A network server for solving computational science problems*, *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2), 1997.

[10] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura and S. Matsuoka, *Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing*, Journal of Grid Computing, Vol. 1, No. 1, pp. 41-51, Kluwer Academic Publishers, June, 2003.

[11] Cactus Webmeister, *The cactus code website*, http://www.CactusCode.org, 2000.

[12] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz, *Distributed Processing of Very Large Datasets with DataCutter*, Parallel Computing, Vol.27, No.11, pp.1457–2478, 2001.

[13] M. Sirbu and D. Marinescu, *A scheduling expert advisor for heterogeneous environments*, the 6th Heterogeneous Computing Workshop (HCW' 97), pp.74-87, 1997.

[14] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, D. Zagorodnov, *Adaptive Computing on the Grid Using AppLeS*, IEEE Transactions in Parallel and Distributed Systems, vol. 14, no. 4, pp 369-382 - April 2003.

[15] G. Shao and R. Wolski and F. Berman, *Master/Slave Computing on the Grid*, Proc. the 9th Heterogeneous. Computing Workshop, Cancun, Mexico, May 2000, pp. 3-16.

[16] J. Gehring, A. Reinefeld, *MARS – a framework for minimizing the job execution time in a metacomputing environment*, Future Generation Computer Systems, v.12 n.1, p.87-99, May 1996.

[17] T. Ma, S. Li, *An Instance-Oriented Security Mechanism in Grid-based Mobile Agent System*, Proc. IEEE Cluster Computing (Cluster 2003), Dec. 2003, Hong Kong.

[18] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke, *A Security Architecture for Computational Grids*, Proc. 5th ACM Conference on Computer and Communications Security Conference, pp. 83-92, 1998.

[19] Lin Chen, Cho-Li Wang, and Francis C.M. Lau, *A Grid Middleware for Distributed Java Computing with MPI Binding and Process Migration Supports*, Journal of Computer Science and Technology (China), Vol. 18, No. 4, July 2003, pp. 505-514.

[20] S.F Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman, *Basic local alignment search tool*, J. Mol. Biol., 215:403, 1990.

[21] T.D. Braun, H.J. Siegel, N. Beck, L.L.Boloni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, D. Hensgen and R.F. Freund, *A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems*, Journal of Parallel and Distributed Computing, 61(6):810-837.

[22] *Globus Toolkit*: http://www.globus.org

[23] *Sun Java: Java Virtual Machine Debugger Interface*: http://java.sun.com/j2se/1.4.2/docs/guide/serialization/index.html