Dynamic Scheduling with Process Migration*

Cong Du, Xian-He Sun, and Ming Wu
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616, USA
{ducong, sun, wuming}@iit.edu

Abstract

Process migration is essential for runtime load balancing. In Grid and shared networked environments, load imbalance is not only caused by the dynamic nature of underlying applications, but also by the fluctuation of resource availability. In a shared environment, tasks need to be rescheduled frequently to adapt the variation of resources availability. Unlike conventional task scheduling, dynamic rescheduling has to consider process migration costs in its formulation. In this study, we first model the migration cost and introduce an effective method to predict the cost. We then introduce a dynamic scheduling mechanism that considers migration cost as well as other conventional influential factors for performance optimization in a shared, heterogeneous environment. Finally we present experimental testing to verify the analytical results. Experimental results show that the proposed dynamic scheduling system is feasible and improves the system performance considerably.

1. Introduction

Many distributed environments have been developed to meet the demand for more computation power. Some of the well-known distributed systems are Condor, NetSolve, Nimrod, and the Grid environment [14]. Resources in these systems are heterogeneous and are shared among different user communities. Each resource or organization may have its own resource management policies and resource usage patterns. Central control does not exist in resource management. To harvest Grid computing in these environments requires a continued dynamic rescheduling of Grid tasks to adapt to the availability of locally controlled computing resources. In addition, besides load balance, migration-based dynamic scheduling also benefits dynamic Grid management [19] in the cases of new machines joining or leaving, resource cost variation, and local task preemption.

An appropriate rescheduling should consider the migration costs. This is especially true in distributed and heterogeneous environments, where plenty of computing resources are available at any given time but the associated migration costs may vary largely. An effective and broadly applicable solution for modeling and estimating migration costs, however, has been elusive. Even if an estimate is available, integrating migration cost into a dynamic scheduling system is still a challenging task. Based on our years of experience in process migration [8] and task scheduling [24], we propose an integrated solution in this study.

The design of a migration-based dynamic scheduling is fourfold: reschedule triggering, migration cost modeling, task scheduling, and parameter measurement. We have proposed a reschedule triggering system [10]. In this paper, we focus on the three remaining problems. We choose to analyze the migration cost based on our HPCM (High Performance Computing Mobility) middleware [12]. HPCM is a middleware released under the NSF middleware initiative. It has a complex structure to support reduced process states and pipelined communication/execution for efficient process migration. All the parameters of the migration cost model are measured by monitoring the system and application running status at runtime. Due to the sophistication of HPCM, the analytical results presented in this study can be extended to other existing migration and checkpointing systems as well. Based on the estimated migration cost, we develop an integrated dynamic scheduling system to optimize application performance.

In the next section, we give an overview of related work. In Section 3, we briefly describe the process migration mechanisms and then model the migration cost. A dynamic scheduling algorithm is introduced in

* This research was supported in part by national science foundation under NSF grant SCI-0504291, CNS-0406328, EIA-0224377, and ANI-0123930.

Section 4. Experiments and the parameter measurement methodologies are presented in Section 5. Conclusions and future work are discussed in Section 6.

2. Related work

Different task scheduling policies have been used in distributed shared environments. Condor system [20] uses a matchmaking mechanism to allocate resources with ClassAds. The scheduling strategy is based on the match of the users' specification of their job requirements and preferences, with the machines' characteristics, availabilities, and conditions. The process migration is implemented based on a checkpointing-based mechanism. However, it does not support run-time process migration in heterogeneous environments. AppLeS [5] is a well-known task scheduling system in Grid computing. It uses a loop of task events to schedule subtasks of a meta-task dynamically. While it can reschedule un-started subtasks, it does not support checkpointing or process migration. Projects like Mosix [3], and OpenSSI [21] support Single System Image (SSI) clustering, and hence support process migration over the nodes within the cluster. Because SSI technologies assume a tightly coupled cluster environment, these systems cannot be applied to massive message-passing based parallel applications or a general loosely coupled Grid environment. Virtual Machine Migration [6] may also be used in load balancing. However, because it requires the migration of the entire running environment, including the operating system, it is heavy-weighted in nature and only works in local-area clusters with fast communication channels. The Linux Zap [22] supports migration of legacy applications through the use of loadable kernel modules and virtualization of both hosts and processes. It uses a checkpointing-based mechanism to support process migration on Linux. The Zap system, as well as some heterogeneous process migration systems [23], has not implemented any mechanism for dynamic scheduling and reallocation. Their migration costs have never been studied in depth and their migrations are conducted manually. The benefit of rescheduling may not reach its full potential if it does not consider the migration cost.

3. Migration cost analysis

Process state collection, transmission and restoration are of general importance in process migration. While migration has potential performance gain for running tasks, the scheduling must be aware of the migration cost, which is the cost to migrate a running process to its new location. In this section, we first present a general migration cost model. Then, to provide feasible runtime prediction, we conduct in-depth analysis on the HPCM middleware [12].

HPCM is a user-level middleware supporting heterogeneous process migration of legacy codes written in C, Fortran or other stack-based programming languages via denoting the source code. It consists of several subsystems to support the main functionalities of heterogeneous process migration, including source code pre-compiling, execution state collection and restoration, memory state collection and restoration, communication coordination and redirecting, and I/O state redirecting. We have developed several optimization mechanisms to reduce the migration cost, including communication/execution pipelining, and live variable analysis. To make correct decisions and achieve precise scheduling, it is important that the migration cost, as well as the amount of process state, is analyzed and measured at runtime.

The input of HPCM is the source code of an application. The pre-compiler or the users choose some points (called poll-points) in the source code. A poll-point is a point where a migration can occur. The pre-compiler annotates the source code and outputs the migration capable code, namely the annotated code. The annotated code is pre-initialized on the destination machine before a migration. When a migration is demanded, the migrating process first transfers the execution state, I/O state, communication state and partial memory state to the initialized destination. The pre-initialized process resumes execution while the remaining memory state is still in transmission. That is, the process states are transferred in a pipelined manner. The concurrency saves significant time in a networked environment, especially when a large amount of state data needs to be transmitted. The pipelining, however, imposes difficulty in estimating the migration cost.

To migrate an application over heterogeneous systems, we represent the application's memory space by a Memory Space Representation (MSR) model [7], which is a machine-independent logical representation of memory space. The *snapshot* of an application's memory space is modeled as a MSR directed graph. Each vertex in the graph represents a memory block. Each edge represents a relationship between two blocks when one of them contains a pointer, which points to a memory location within another memory block. MSRLT (MSR Lookup Table) is a global mapping table between application memory space and the conceptual MSR model. Each memory block that may be referenced in the MSR, including a dynamic memory block, has an entry in the table. To represent a pointer, which contains a machine-specific address, the

MSRLT is searched for the memory block that contains the address. The pointer is then represented in MSR by an edge to the referenced memory block. The pre-initialized process restores the pointer to the correct address allocated to the referenced memory block.

3.1. Migration cost

The task scheduling system is based on the statistical information gathered by the system monitoring and the estimated migration cost.

Similar to a checkpoint/restart system, the migration is separated into three phases: data collection, data transmission and data restoration. The times spent on these phases are represented as T_c, T_t, and T_r, respectively. The source machine and the destination machine are represented as m_s and m_d. For a general process migration system without any optimization, the cost to migrate a running process from m_s to m_d is:

$$C_{sd} = T_c + T_t + T_r. \quad (3\text{-}1)$$

However, estimating the migration cost based on this general migration cost model shown in (3-1) is not practical. First, a precise estimation of each parameter in (3-1) highly depends on the implementation of the migration system, which is not general. Second, (3-1) cannot be applied to optimized process migration systems such as the HPCM system, where the phases are overlapped to reduce the migration cost.

In the following, we derive a migration cost model to estimate the migration cost of a process at a given *migration point* (the break point in the execution sequence where migration occurs). Though we use HPCM middleware throughout the analysis, the model is general and can be extended to provide accurate estimation of other migration systems.

Given an application *App* running on a machine, at time $t = 0$, it reaches a poll point P. If *App* does not migrate at time t, it finishes on m_j at t_{jj}. If *App* is scheduled to migrate to another machine at time t, it finishes on m_i at t_{ji}. For convenience, t_j is used instead of t_{jj} in the following. The available communication bandwidth from m_j to m_i is b_{ji}, which can be estimated with existing network performance prediction tools such as [25] and [13]. The available computing capacity of m_j for application *App* is τ_j. In Section 4, we will discuss how to measure and predict this parameter.

The migration cost C_{ji} is defined as the time spent to migrate *App* from m_j to m_i. So, $t_{ji} = C_{ji} + t_j\tau_j/\tau_i$. If m_i has the same computing capacity as m_j, that is $\tau_j = \tau_i$ (in most cases, this means m_i is identical to m_j), then the migration cost is

$$C_{ji} = t_{ji} - t_j. \quad (3\text{-}2)$$

3.2 Process state

A process's state is represented as $S = <App, P, M, IO, Comm>$. They are the execution state P, memory state M, I/O state *IO,* and communication state *Comm*. *f(S)* is the size of S. In HPCM, data collection, transmission and restoration overlap with each other. We assume the migration is from m_j to m_i and C is the abbreviation for C_{ji}. So the migration cost,

$$\max(T_c, T_t, T_r) < C < T_c + T_t + T_r$$

The complexity of data collection and restoration is application-specific. Based on the data collection and restoration algorithm, we can define the data collection time as:

$$T_c = search(MSRLT) + encode(S) + copy(S)$$

and the data restoration time as:

$$T_r = update(MSRLT) + decode(S) + copy(S).$$

where *search(MSRLT)* is the time searching the MSRLT data structure; *update(MSRLT)* is the time updating the MSRLT data structure with machine-specific address; *encode(S)* is the time encoding the data to a machine-independent format; *decode(S)* is the time decoding data; *copy(S)* is the time copying data to or from a buffer. For homogeneous migration, it is not necessary to encode data, so *encode(S)* and *decode(S)* can be omitted from the formula.

Suppose there are n fully-connected nodes in MSR graph. Because the MSRLT is searched in a depth-first manner, *search(MSRLT)* has the upper bound complexity of $O(n \log n)$. To update the references in MSRLT, *update(MSRLT)* takes *O(n)* time complexity. *encode(S)*, *decode(S)* and *copy(S)* state have the complexity of $O(f(S))$. The transmission time T_t also takes the time complexity of $O(f(S))$. Putting them all together, the migration cost is represented as:

$$C \approx \alpha_0 + \mu f(S), \quad (3\text{-}3)$$

α_0 is a small migration overhead that is application specific and depends on the number and size of fully-connected subgraphs in the MSR model. If the application has a large amount of referenced or dynamic allocated memory blocks, α_0 is bigger. For other systems without the MSR model and performance optimization, α_0 is a constant. μ is called migration processing rate, and it is represented as seconds per byte. μ is proportional to the reciprocal of the current available bandwidth, b_{ji}, between the source and destination node, that is $\mu = \frac{\alpha_1}{b_{ji}}$. α_1 is an application dependent constant that reflects the overlapping factor. α_1 can be analytically computed

without consider overlapping. The overlapping factor can be measured directly.

Experiments shown in Figure 1 confirm formula (3-3). The linear increase of migration cost shows that the migration cost is proportional to the size of the state for a given application *App* and migration point *P*. The migration cost is determined by the amount of state to be collected, transmitted and restored during the migration.

We measure α_0 and α_1 by experiments. As shown in Figure 1, $\alpha_0 = 0.0004$ seconds, and $\mu = 0.025$ second/MB when there is no other traffic on the experimental platform. So $\alpha_1 = 2.23$. The migration cost *C* increases from 0.01 to 15.46 seconds when *f(S)* increases from 0.415 to 635.22 MBytes.

In the following, we estimate the size of the process state. A process state consists of execution state, memory state, communication state, and I/O state.

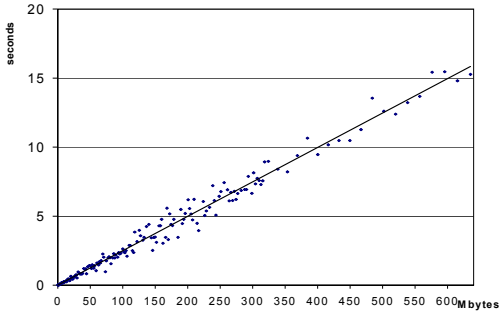

Figure 1. Process State and Migration Cost

Execution State. A stack containing the functions in the calling sequences and their current execution locations represents the execution state *P*. The migration point is represented by <function, location> couples in a stack, which are maintained by macros inserted at the beginning and the end of each function. For example, the execution state of a process is $P = <$ $<main, main\_L_2>$, $<foo1, foo1\_L_1>$, $<foo2, foo2\_L_4>$, $<foo1, foo1\_L_2>$, ...>, where $main\_L_2$ represents the second poll point in the function main. So,

$$f(P) = \beta \bullet depth(P) \quad (3\text{-}4)$$

where the *depth(P)* is the depth of the stack at the migration point *P* and β is the amount of data for a single entry in the execution stack.

Memory State. There are three memory spaces in a running process: heap space, global space, and stack space. For programming languages like C and C++, variables are defined at different locations and with different storage type modifiers to indicate their storage idiosyncrasies and locations. Blocks in heap space are allocated at runtime and referenced by pointers. Also there are some variables residing in the register file for fast access. The global variables and static variables reside in global space and can be accessed after a function call. So, all the variables in the global space are collected, transmitted and restored as a part of the memory state. The dynamic allocated memory blocks reside in the heap space with pointers referencing to these blocks. A dynamic memory block may lose all its references during execution, and then it is called "garbage" and cannot be accessed afterward. Only valid memory blocks need to be collected and transmitted. Each function in the running sequence has a stack segment and a local address space in stack. All the parameters, including local auto variables and temporary variables (variables defined by a compiler to store the intermediate computing values) are stored in the stack. They are only valid before the function return to its caller. We perform live variable analysis on these variables. That is, only variables that are *live* (may be accessed after the migration point) at the migration point are collected and transmitted. The amount of memory state is represented as:

$$f(M) = M_g + M_s(P) + M_h(P) \quad (3\text{-}5)$$

where M_g is the size of variables in global address space; $M_s(P)$ is the size of live variables at migration point *P* in stack space; $M_h(P)$ is the size of the dynamic memory blocks at migration point *P* in the heap space.

I/O State. Distributed applications may use many approaches to store and access their data. Commonly used approaches include network file systems such as NFS, distributed file systems such as DFS and Coda [4], and data transfer protocols such as FTP and GridFTP [16]. The data migration cost highly depends on these storage systems and data transfer protocols. Estimating the performance of these systems is out of the scope of this paper. In the following, we assume the presence of the globally accessible storages. The application data are not moved during the migration. The I/O state of a process is registered into a data structure called I/O Information Table. This table is created and maintained at runtime. An entry in the I/O Information Table is initialized when an I/O instance is created. The I/O information table is transmitted to the destination process and restored accordingly. The amount of the I/O state is:

$$f(IO) = \gamma \bullet iob(P) \quad (3\text{-}6)$$

where γ is the size of each I/O entry in the I/O Information Table; *iob(P)* is the size of I/O Information Table at the migration point *P*.

Communication State. The communication state of a process is composed of all its active connections established with other processes [8]. For each

connection, the migrating process coordinates with its communication partners to direct new messages to the destination process, drains the message queue and saves the incoming message to the received-message-list. The communication state is forwarded to the destination machine with its received-message-list. The destination process then restores the communication connections. It first checks the received-message-list for incoming messages in future execution. The size of communication state is:

$$f(Comm) = \sum_{i \in N(P)} length\ (msg\ (i)) \qquad (3\text{-}7)$$

where *N(P)* is the set of all the established connections of the process *App* at the migration point *P*; *msg(i)* is the size of the received_message_list of connection *i*.

Putting them all together, the size of process state is:

$$f(S) = \beta \bullet depth(P) + M_g + M_s(P) + M_h(P) + \gamma \bullet iob(P) + \sum_{i \in N(P)} length\ (msg\ (i)) \qquad (3\text{-}8)$$

where *f(S)* is the data size of state *S*; β is the size of an entry in the running stack; M_g is the size of variables in global address space; $M_s(P)$ is the size of live variables at migration point *P* in stack space; $M_h(P)$ is the size of the dynamic memory blocks at migration point *P* in the heap space; γ is the size of each I/O entry in the I/O Information Table; iob(P) is the size of I/O Information Table at the migration point *P*; *N(P)* is the set of all the established connections of the process *App* at the migration point *P*; *msg(i)* is the size of the received_message_list of connection *i*.

4. Dynamic task scheduling

We have developed a dynamic task scheduling system to reallocate applications dynamically at runtime. To choose a machine as the destination machine, we calculate the expected application execution time after migration and the cost of migration. Let m_j denote the source machine and m_i is the magration destingation machine. As shown in Section 3, C_{ji}, the time of migrating a process from m_j to m_i, is

$$C_{ji} = \alpha_0 + \mu f(S) \qquad (3\text{-}3)$$

Let $w_j^{'}$ denote the unfinished workload on m_j. The completion time of the unfinished workload $w_j^{'}$, T_{ji}, is calculated as follows.

$$T_{ji} = C_{ji} + T(S_j^{'}) \qquad (4\text{-}1)$$

where $T(S_j^{'})$ denotes the execution time of the application with unfinished workload on machine m_i.

When the destination machine m_i is dedicated to the execution of the migrated application, we can calculate $T(S_j^{'}) = w_j^{'} / \tau_j$ where τ_j denotes the computing power of the machine m_i. If the destination machine is in a shared environment, for example, in a Grid environment, its resource availabilities may vary with time. To estimate the application execution time on a shared resource, we need to identify the availability of computing resources and its influence on the application performance.

Assumption: an application is located on machine, m_0.

Objective: dynamically reallocate an application when an abnormality is noticed

---

Begin

Receiving the triggering signal

List a set of idle machines that are lightly loaded over an observed time period, $M = \{m_1, m_2, \ldots m_q\}$;

$p' = 1$;

For each machine m_k $(1 \le k \le q)$,

Use Formula (3-3) to calculate the migration cost, C_{0k};

Use Formula (4-2) to calculate the mean of the remote task execution time, $T(S_k^{'})$;

Use Formula (4-1) to calculate the application completion time, T_{0k}

If $T_{0p'} > T_{0k}$, then $p' = k$;

End For

Migrate the application from m_0 to $m_{p'}$;

End

Figure 2. Dynamic task scheduling algorithm

We use parameters λ, ρ, σ to describe the dynamic resource usage pattern of a shared machine. λ is the local job arrival rate, σ is the standard deviation of job service time, and ρ is the resource utilization. We assume the arrival of local jobs follows a Poisson distribution with λ. The service time of local jobs follows a general distribution with mean $1/\mu$ and standard deviation σ. These assumptions are based on the observations of machine usage patterns reported by researchers in Wisconsin-Madison, Berkeley, Maryland et. al. [2]. The cumulative distribution function of the application completion time on a machine can be calculated as [15]:

$$\Pr(T \le t) = \begin{cases} e^{-\lambda w/\tau} + (1 - e^{-\lambda w/\tau}) \Pr(U(S) \le t - w/\tau \mid S > 0), & if\ \ t \ge w \\ 0, & otherwise \end{cases} \qquad (4\text{-}2)$$

where $U(S)$ is the sum of busy periods of local jobs on the machine. τ is the computing capacity of the machine and w is the workload of the application.

After identifying the cumulative distribution function of $T(S_j')$, we can decide which machine should be selected as the destination machine. The basic idea is given below. First, we list a set of idle machines that are lightly loaded over an observed time period. Then for each machine, we calculate the migration cost and the expected application execution time with formula (3-3) and (4-2) respectively. The machine which has the minimum expected application completion time will be chosen as the destination. Figure 2 gives the detailed dynamic task scheduling algorithm.

5. Experiment results

We implemented the dynamic scheduling algorithm to verify the correctness of the migration cost model. We performed experiments on the sunwulf Computer Farm in the Scalable Computing Software (SCS) laboratory at the Illinois Institute of Technology. Sunwulf is a heterogeneous cluster in both computing and communication capacity. In our previous work, HPCM has been proven to work well on both heterogeneous and homogeneous ISAs. In this paper, we focus on the heterogeneities in computation and communication capacity and their impact on the scheduling mechanism. ISA heterogeneity, which does not affect our model and scheduling mechanism, is not discussed. Sunwulf is composed of one Sun Enterprise 450 server node (sunwulf node), 64 Sun Blade workstations 100 (hpc-1 to hpc-64) and 20 Sun Fire V210R (hpc-65 to hpc-84) compute nodes. The Sun Enterprise 450 server has four CPUs, 8M cache and 4GB memory. Each CPU is 480 MHz. The Sun Blade compute node has one 500-MHz CPU, 256K L2 cache, and 128M memory. The Sun Fire V210R compute node has two 1GHz CPUs, 1M L2 cache and 2GB memory. All the systems are running SunOS 5.9 operating system. All the Sun Fire 210R servers are connected with a Gigabits Ethernet. The maximum bandwidth is 89.1M bytes/s. Other communication channels within the workstations or between the servers and the workstations are 100Mbps internal Ethernet. The maximum bandwidth is 11.8M bytes/s. The workstations are organized as a "fat tree" structure. The computation and communication heterogeneities make sunwulf a good test bed for our system.

In the first experiment, we have tested four applications to verify the migration cost model. The first one is the linpack C sequential program, which solves a dense system of linear equations with Gaussian elimination [11]. The second is the bitonic program written by Joe Hummel [18], which builds a random binary tree and then sorts it. The third is gzip, a popular compression utility. The last application xlintims is from CLAPACK [9] with single precision real timing routines. CLAPACK provides routines for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems.

According to formula (3-3), the migration cost is proportional to the data size of the process state. Because the data collection overlapped with data communication, α_0 is very small. The maximum value we observed in our experiments is less than 0.01 seconds for bitonic program which has a large amount of dynamically allocated memory blocks. In the following experiments, we assume it is a constant. α_1 is an application dependant constant and b_{ji} is the

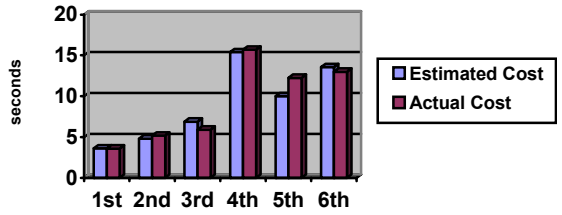

Figure 3. Migration Cost Estimation (Linpack)

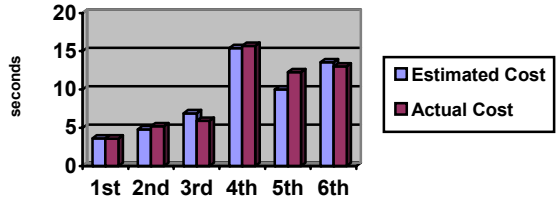

Figure 4. Migration Cost Estimation (Bitonic)

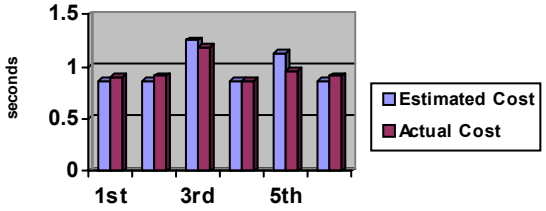

Figure 5. Migration Cost Estimation (gzip)

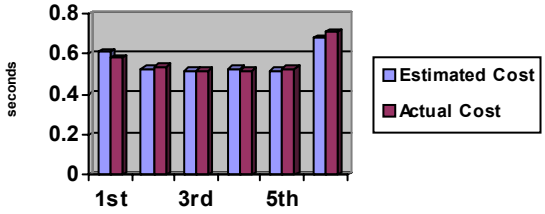

Figure 6. Migration Cost Estimation (xlintims)

available bandwidth from m_j to m_i. We perform our tests on Sun Fire nodes with Gigabit Network and randomly generate communication load to simulate the traffic in Grid. Since the source and destination machine are identical and have the same computing capacity, the actual migration cost can be measured by formula (3-2). The parameters in formula (3-8) are measured at runtime by querying the running applications for current status. After receiving a migration request, the dynamic scheduling system sends a query to the running process asking for current size of state by a user-defined signal. The running process checks its current execution state, memory state, I/O state and communication state and generates a process description schema in XML language describing current data size of each process state. The process schema includes the static and dynamic information about a running process such as application name, type (computational intensive, data intensive, dynamic memory management intensive, etc.), and data size of each state. β, γ and the size of global memory state M_g are provided by a precompiler. In our experiments, $\beta = 4$ *and* $\gamma = 28$. The depth of execution stack *depth(P)*, the memory size of stack space $M_s(P)$, heap space $M_h(P)$, and the size of I/O information table *iob(P)* are measured and maintained by HPCM middleware. The number of live communication connections *N(P)* and the length of the received message list *length(msg(i))* are measured at runtime through querying. Based on the information provided by the process schema, the dynamic scheduling system calculates the size of process state using formula (3-8) and then estimates the migration cost using formula (3-3).

Table 1. Comparisons of scheduling Strategies

seconds		DSA	MCM	ETM	RS
linpack	1	2395.33	2578.52	2468.12	4174.63
	2	494.91	619.49	553.46	526.54
	3	65.83	89.42	81.29	90.82
	4	21.52	23.60	25.02	76.81
xlintims	1	557.78	1359.56	557.81	557.72
	2	57.18	67.45	69.54	92.83
	3	615.97	1489.73	616.15	1466.6
≤5%		5%-20%		20%-50%	>50%

As shown in Figure 3-6, the migration cost can be estimated with error ranging from 0%-18%. Each application is tested 6 times. The migration is triggered randomly so the running state varies each time. As shown in the figures, the migration cost may vary according to the application's current running state. The average error is 8.19% for linpack, 5.76% for bitonic, 6.51% for gzip and 2.51% for xlintims. The overall average error for these tests is 5.74%. This experiment shows that the migration cost can be precisely predicted by our migration cost model.

To evaluate the efficiency of the proposed dynamic scheduling algorithms (DSA) in selecting a destination machine for process migration, we conduct experiments to compare its performance with other machine selection strategies: migration-cost-minimum (MCM), execution-time-minimum (ETM), and random selection (RS). The migration-cost-minimum approach chooses a machine to which the migration cost is the minimum. The execution-time-minimum approach is to find a machine where the unfinished workload of the application will be executed in a minimum time. We have mentioned that the sunwulf cluster is heterogeneous in terms of the nodes' computer power and the underlying communication infrastructure. To further increase the heterogeneity of our test platform to simulate a Grid environment, we generate synthetic traffic on the network and workload on the nodes. In our simulation environment, the arrival rate of local jobs on each machine follows Poisson distribution. The local jobs' lifetime is simulated with $2.0/x$ [1], which follows the observation of real-life processes in [17]. x is a random number between 0 and 1. The local job arrival rate and the job service rate on each machine are randomly generated in an adjustable range. The resource utilization of each machine is thus different. We randomly generate the network traffic so that the end-to-end network performance among those nodes is different. Table 1 shows the application execution time with different machine selection approaches. The job execution time of each selection strategy is compared with the minimum completion time of all strategies and is marked with different grey levels for ≤5%, 5% to 20%, 20% to 50% and >50% higher than minimum completion time respectively. The experiment results show that with considering the migration cost, the proposed scheduling algorithm (DSA) is the best in performance. ETM may find acceptable destination for the process for light process migration. However, as shown in Table 1 for Linpack C applications, when the communication channel is busy, ETM cannot avoid performance degradation caused by increased migration cost.

6. Conclusion and Future Work

In this paper, we study dynamic scheduling in a shared distributed environment. We have introduced a migration cost model, derived a dynamic scheduling algorithm that considers migration costs as a decision

factor, and implemented an automatic dynamic scheduling system that integrates the model, algorithm, and a triggering/monitor subsystem. Experimental results show that the model is precise, the scheduling algorithm is more appropriate than existing scheduling algorithms, and the dynamic scheduling system is effective and practical. The proposed dynamic scheduling system has a real potential to positively impact parallel and distributed computing.

We have considered worst-case scenarios in our analysis and implementation. The Grid environment is heterogeneous and shared, and the HPCM migration system supports the transfer of runtime, memory, and communication states. The analysis and implementation can be extended to other less powerful migration systems or to dedicated environments. Since checkpointing and migration mechanisms differ mostly in communication state, the results can be also applied to checkpointing systems.

We have proposed and implemented a prototype of the dynamic task scheduling system to reallocate processes dynamically. Currently, we select the destination machine based on an estimate of the completion time of the migrated process. When an application consists of multiple processes running concurrently on different machines, we need to consider the overall application completion time as a selection criterion. We plan to extend our current work to this more complicated scenario in the future.

References

[1] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, et al, "An opportunity cost approach for job assignment in a scalable computing cluster," *IEEE Trans. on Parallel and Distributed Systems*, vol. 11, no. 7, 2000.

[2] A. Acharya, G. Edjlali, and J. Saltz, "The utility of exploiting idle workstations for parallel computation," in *Proc. of SIGMETRICS*, 1997, pp. 225-236.

[3] A. Barak, S. Guday, and R. G. Wheeler. "The MOSIX Distributed Operating System: Load Balancing for UNIX", vol. 672. Springer-Verlag, 1993.

[4] P. J. Braam. The Coda Distributed File System. *Linux Journal*, June 1998.

[5] F. Berman, R. Wolski, H. Casanova, W. Cirne, et al. "Adaptive computing on the Grid using AppLeS," *IEEE Trans. Parallel Distrib. Systems*, 14 (2003) 369-382.

[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, etl "Live Migration of Virtual Machines". In Proc. 2nd Symp. on Networked Sys. Design and Impl. Boston, MA May 2005.

[7] K. Chanchio and X. H. Sun, "Data collection and restoration for heterogeneous process migration," *Software--Practice and Experience*, 32:1-27, April 15, 2002.

[8] K. Chanchio and X-H. Sun, "Communication State Transfer for the Mobility of Concurrent Heterogeneous Computing," *IEEE Trans. on Computers*, Vol. 53, No. 10, pp: 1260-1273, 2004.

[9] http://www.netlib.org/clapack/

[10] C. Du, S. Ghosh, S. Shankar, and X.-H. Sun, "A Runtime System for Autonomic Rescheduling of MPI Programs," in *Proc. of Intl. Conf. of Parallel Processing*, Montreal, Canada, August 2004.

[11] J. Dongarra, "The Linpack Benchmark: An Explanation," in *Proc. of 1st Intl. Conf. on Supercomputing*, pp. 456-474, Athens, Greece, June 8-12, 1987, Springer-Verlag 1988.

[12] C. Du, X.-H. Sun and K. Chanchio, "HPCM: A Pre-compiler Aided Middleware for the Mobility of Legacy Code," in *Proc. of IEEE Cluster Computing Conference*, Hong Kong, Dec. 2003. Software available at: http://archive.nsf-middleware.org/NMIR4/contrib/download.asp.

[13] A. Eswaradass, X.-H. Sun, and M. Wu, "A Neural Network Based Predictive Mechanism for Available Bandwidth," in *Proc. of 19th Intl Parallel and Dist. Processing Symposium*, Denver, CO, April., 2005.

[14] I. Foster, C. Kesselman, *The Grid2: Blueprint for a New Computing Infrastructure*, Morgan-Kaufman, 2004.

[15] L. Gong, X.-H. Sun, E. F. Waston, "Performance modeling and prediction of non-dedicated network computing," *IEEE Trans. Comput.* 51 (2002) 1041-1055.

[16] B. Allcock, L. Liming, and S. Tuecke, "GridFTP: A Data Transfer Protocol for the Grid."

[17] M. Harchol-Balter and A. Downey, "Exploiting process lifetime distributions for dynamic load balancing," in *Proc. of ACM Sigmetrics Conf. Measurement and Modeling of Computer Systems*, 1996.

[18] J. Hummel, L. Hendren, A. Nicolau. "A language for conveying the aliasing properties of dynamic, pointer-based data structures," in *Proc. International Conference on Parallel Processing*, 1992.

[19] K. Keahey, M. Ripeanu, and K. Doering. "Dynamic creation and management of runtime environments in the Grid," *Workshop on Designing and Building Web Services (GGF 9)*, Chicago, October, 2003.

[20] 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.

[21] http://openssi.org/

[22] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The Design and Implementation of Zap: A System for Migrating Computing Environment," in *Proc. of the 5th Operating System Design and Implementation*, Dec. 2002.

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

[24] X.-H. Sun and M. Wu, "Grid Harvest Service: A System for Long-Term, Application-Level Task Scheduling," in *Proc. of IPDPS 2003*, Nice, France, Apr 2003.

[25] R. Wolski, N. T. Spring, and J. Hayes, "The Network Weather Service: a Distributed Resource Performance Forecasting Service for Metacomputing," *Journal of Future Generation Computing Systems,* vol. 15, no. 5-6, pp. 757-768, Oct. 1999.