Memory Conscious Task Partition and Scheduling in Grid Environments

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

Abstract

While resource management and task scheduling are identified challenges of Grid computing, current Grid scheduling systems mainly focus on CPU and network availability. Recent performance improvement of CPU and computer network has made memory usage a significant factor of overall performance. In this study, we consider memory availability as a performance factor and introduce memory conscious task partition and scheduling. Three task partition policies are discussed. They are CPU-based, memory-based, and CPU-memory combined partition. We first investigate the three task partition policies on dedicated resources and verify the effectiveness of the CPU-memory combined partition algorithm in finding an optimal solution. We then extend the task partition policies in non-dedicated environments with the consideration of resource sharing. Analytical and experimental results show that the CPU-memory combined scheduling approach outperforms either the CPU-based or memory-based scheduling approach considerably for memory-intensive applications in Grid environments.

1. Introduction

In order to provide high performance computation power to serve the increasing need of large applications, researchers strive to construct distributed systems for resource sharing and collaboration. Some of well-known existing heterogeneous distributed systems include Condor [1], NetSolve [2], Nimrod [3]. In the late 1990's, a much more complex computation environment, Grid computing, was emerged [4]. The national scale Grid consists of various virtual organizations and has the potential to bring unprecedented computing power. However delivering this unprecedented computing power to the users is still an elusive problem. One of the major issues is how to schedule a large application in this complex infrastructure [5]. Many heuristic scheduling algorithms have been proposed for traditional high performance computing. Unfortunately, most of them are for dedicated systems and do not consider memory constraint in task scheduling. By dedicated, we mean the resources are dedicated for a given application. In contrary current distributed systems are composed of shared resources, where dedicated usage can only be achieved through reservation under certain environments. An effective scheduling system that can harvest shared resources is central to the success of Grid computing [4].

The application performance not only depends on CPU speed, but also on memory access speed. Compared to the rapid development of CPU technology, memory speed improvement lags behind the improvement of CPU speed, which hinders the further improvement of application performance. Some efforts have been made recently on efficiently managing memory resources in a distributed system. Load sharing policies were proposed to assign individual jobs to each node of a cluster system according to the combined load index of both CPU cycles and memory requirement [6-7]. They aimed to reduce the average slowdown of all the jobs where slowdown is the ratio between the wall execution time and the CPU execution time of a job. In this study, we consider, instead, how to partition a Grid application and schedule it on a cluster of distributed heterogeneous resources to obtain a minimum application execution time with the consideration of both CPU resource availability and memory resource availability. Research of Grid task scheduling is still in its infancy. The effect of CPU availability is studied in [5,8-10]. The effect of memory availability on task partition and scheduling, has not received appropriate attention at this time. This study intends to change this situation.

The rest of this paper is organized as follows: Section 2 describes the related work. Section 3 introduces the system and application model. Three task partition policies on dedicated resources are investigated in this section. These policies, then, are extended into a Grid environment with the consideration of resource sharing.

Experimental results are presented in Section 4. Finally we conclude and summarize our work with Section 5.

2. Related work

Traditional work in task scheduling policy was mainly focused on dedicated systems, which cannot be assumed in non-dedicated computing environment like the Grid. New scheduling systems considering resource availability are under development [5,8-10]. A resource reservation strategy [11] was proposed in GARA to deliver end-to-end Quality of Service for high-end applications. However, reservation tends to lead to low system utilization and may conflict with local resource management policy. Condor system [1] provides a matchmaking mechanism to allocate resources with ClassAds. However, its major goal is to improve the system throughput instead of parallel application performance. Scheduling algorithms in the AppLeS [5,8] project are designed for parallel applications. They are supported by short-term resource prediction provided by NWS services [12]. These algorithms consider CPU and Network resource availability. However, their scheduling decisions are based on determined estimation of resource availability and apt for short-term applications. In contrast, a long-term, application-level performance prediction and task scheduling system, namely Grid Harvest Service (GHS) system, is proposed recently [9]. However, as AppLeS, GHS scheduling system ignores the impact of memory resource availability on the scheduling decision-making. In Nimrod/G, the economy is emphasized instead of application performance [13].

Several studies have been reported on task allocation for load balance in a cluster computation environment considering memory resource constraints. An opportunity cost approach proposed in [6] converts the usage of resources including CPU and memory to a single homogeneous cost. Based on the cost, task is assigned or reassigned to each node for load balance. Load sharing policies with the consideration of effective usage of global memory were studied in [7]. They consider two types of application workload, known memory demands and unknown memory demands. However, both systems mainly target at load balance in cluster computing. Their major concern is how to reduce the average slowdown of all individual jobs in the system, instead of how to schedule a parallel application to achieve its best performance. A page replacement policy has been studied in [14] to impose the hard bounds on the available memory space for guest processes. This work provides technical support for memory resource reservation.

3. Task partition and scheduling

In this section, we first introduce the system and application model. Second, we investigate CPU-based, memory-based, and CPU-memory combined partition policies for dedicated systems. These policies are then extended to shared environments.

3.1. System and application model

We assume the system is composed of a set of machines $\{m_1, m_2, ..., m_n\}$ connected through general network infrastructure. Each machine in the system has its own computing capacity and memory space. It could be either a dedicated resource or a shared resource. If a machine is a shared resource, we name the machine owner's sequential jobs as local jobs. The task to be scheduled is named as a Grid application. We assume that local job processing follows M/G/1 queuing system. This assumption is based on the observation of shared machine usage pattern [15-16]. We list the notion to be used through out this paper in the following.

τ_i : Computing capacity of machine m_i .

a_i : Available physical memory space for a Grid application at machine m_i .

λ_i : Arrival rate of local jobs at machine m_i .

ρ_i : System utilization at machine m_i .

σ_i : Standard deviation of service time of local jobs on machine m_i .

$\gamma_{i,j}$: link bandwidth between machine m_i and machine m_j .

In this study, we focus on iterative parallel applications. In this application model, the application workload is distributed to a group of subtasks during each iteration. Subtasks are executed on a set of processors. After all subtasks are completed, the results are collected for the calculation of the application workload at the next iteration. To adapt the possible application workload change or system dynamics, the application workload may be redistributed among processors. This process is repeated until the completion of the application. The execution of an iterative parallel application usually follows a master-worker paradigm. A master process iteratively generates worker processes and assigns them to available resources. Iterative parallel applications are receiving increased attention because they have the potential to be executed adaptively in a dynamic environment such as Grid computing. Iterative parallel application have been widely used in solving a number of problems such as image processing applications, N-body simulations and computational fluid dynamics [17]. The application execution time on each machine in an iteration is composed of two parts: computation cost and communication cost. The computation cost is the time required to finish the subtask workload and the communication cost is the time required for data transfer between the master process and worker process. A specific example of master-worker application that has been widely studied in Grid environments is meta-task, which is composed of a set of independent indivisible subtasks. The task partition and scheduling of a meta-task will be discussed in details in subsection 3.4. In this paper, we assume a subtask cannot be executed on a machine if its memory demand can be satisfied. This assumption could be lifted with a complex memory model [7] but it is out of the scope of this paper.

3.2 Task partition on dedicated resources

Scheduling of a single task in a Grid environment with consideration of memory constraint is simple: find all machines satisfying the task memory demand and then choose the machine whose expected task completion time is the shortest. In this study, we focus on parallel processing. In parallel processing, when there is a set of resources available, it is desirable to partition a large application into a group of relatively small subtasks and then assign subtasks to different available resources so that the application execution time can be minimized. How to distribute the application workload to subtasks and then map subtasks to available resources is the issue we need to address.

A natural task partition strategy is to assign each machine a certain amount of workload so that subtasks on different machines are finished at the same time. However, this may be constrained by physical limitations, such as the available memory space on each machine. Our objective is to balance the workload of subtasks as much as possible while satisfying the memory constraint. The task partition problem of an iterative parallel application can be formulated as minimizing the difference between the maximum subtask completion time and the minimum subtask completion time over all possible partition plans subject to $r_i < a_i$, $(1 \le i \le q)$ where r_i denotes the subtask's memory usage on machine m_i and q is the number of machines. In this subsection, we assume that the application workload can be arbitrarily partitioned in each iteration. In subsection 3.4, we will discuss how to extend these policies to meta-task partition.

CPU-only partition: the CPU-only method is a mean-time partition. It partitions the workload of a parallel application so that subtasks on different machines are finished at the same time. Let b denotes the subtask completion time. It includes the subtask execution time and the data transfer time between the master process (we assume it is on m_0) and the worker process that executes the assigned subtask. We assume that the size of the transferred data (D_i) is proportional to the subtask workload w_i ($D_i = cw_i$) and the communication delay is ignorable in data transfer cost. Thus, $b = w_i / \tau_i + cw_i / \gamma_{0,i}$ $(1 \le i \le q)$. w_k is the subtask's workload on machine m_i. Because $W = \sum_{i=1}^{q} w_i$. So the subtask workload on m_k is calculated as

$$w_k = \frac{W}{\sum_{i=1}^{q} \frac{\tau_i \gamma_{0,i}}{\gamma_{0,i} + c\tau_i}} \frac{\tau_k \gamma_{0,k}}{\gamma_{0,k} + c\tau_k} \qquad (1)$$

After getting the subtask workload w_k, we calculate the subtask memory usage requirement, r_k, on each machine. If there exists a machine m_i where $r_i > a_i$, $(1 \le i \le q)$ in a task partition plan on a given machine set, this partition plan is considered invalid. A failure signal is returned to the task scheduling algorithm.

Memory-only partition: the memory-only method partitions the workload of an iterative parallel application according to each machine's memory availability in a given machine set. A machine is assigned with a workload proportional to its available memory space. The workload of a subtask on a machine can be calculated as

$$w_k = \frac{W}{\sum_{i=1}^{q} a_i} a_k \qquad (2)$$

After obtaining the subtask workload w_k on machine m_k, we also check whether the subtask memory usage requirement r_k can be satisfied. If there is a machine m_i where $r_i > a_i$, $(1 \le i \le q)$, the task partition plan is considered as invalid. The scheduling algorithm will choose other possible resource sets for task partition.

CPU-memory combined partition: the CPU-memory combined partition method for an iterative parallel application is given in Figure 1. Let M_r denote the machine set where $r_i < a_i$ is satisfied for all machines**.** The basic structure of this algorithm is a loop of two steps. The first step is to apply a mean-time partition to calculate the subtask workload on a machine m_k. We then check the memory demand of its subtask. If the subtask memory demand cannot be satisfied by the available memory space on the machine, the workload of its subtask will be reduced to the maximum workload that m_k can support in its available memory space. We then set $r_k = a_k$ and remove m_k from M_r. In this way, all machines are assigned with appropriate workloads. After that, we calculate the sum of assigned workloads. The left workload will be redistributed among machines in M_r in the next time-step. This process will be repeated until the sum of left workloads is equal to 0 or $|M_r| = 0$ which indicates no machine is available for task partition in the next time-step. If the left workload is larger than 0 and $|M_r| = 0$ after the loop ends, there is no solution for task partition on this machine set. A failure signal will be returned to the task scheduling algorithm. $f(w_k)$ in Figure 1 is a function calculating the memory usage of a subtask with a given workload. $F(a_k)$ is a function calculating the maximum workload that a machine can support in its available memory space.

We now proceed to prove that the CPU-memory combined partition algorithm given by Figure 1 can find an optimal solution for the defined task partition problem.

LEMMA 1. Let q be the number of a machine set $\{m_1, m_2, ..., m_q\}$ and $\psi^* = \{w_1^*, w_2^*, ..., w_q^*\}$ denote a partition plan where w_i^* is the worker process workload on machine m_i. Let t_{m_i} denote the worker process execution time on machine m_i under partition ψ^*, $\psi = \{w_1, w_2, ..., w_q\}$ be the task partition plan we obtain using the proposed algorithm and $\{m_1', m_2', ..., m_p'\}$ denote those machines where $r_i < a_i$ under the partition plan ψ. If a task partition plan ψ^* is optimal, then $t_{m_1'}^* = t_{m_2'}^* = ... = t_{m_p'}^*$.

PROOF: If $t_{m_1'}^* = t_{m_2'}^* = ... = t_{m_p'}^*$ does not hold, suppose $t_{m_i'}^* = \min\{t_{m_1'}^* = t_{m_2'}^* = ... = t_{m_p'}^*$ and $t_{m_j'}^* = \max\{t_{m_1'}^* = t_{m_2'}^* = ... = t_{m_p'}^*$. It is obvious that $t_{m_i'}^* \neq t_{m_j'}^*$. According to the CPU-memory combined algorithm, the worker process completion time on those machines $\{m_1', m_{2,}', ..., m_p'\}$ is the same because we use the mean-time partition method at each step. We denote it as t_p. If ψ^* is optimal, $t_{m_i'}^* \le t_p$. Thus $w_{m_i'}^* \le w_{m_i'}$. So $r_{m_i'} < a_{m_i'}$ also holds under partition ψ^*. Then we can relocate part of workload from the worker process on machine m_j' to the worker process on machine m_i' to get a new task partition plan, ψ'. In this way, the difference between the maximum worker process execution time and the minimum worker process execution time in ψ' is less than ψ^*. This is against the assumption that ψ^* is an optimal partition plan. Q.E.D.

THEOREM 1: The CPU-memory combined partition algorithm shown in Figure 1 can find an optimal task partition plan on a given machine set if it exists.

It is obvious that if there exists an optimal task partition plan, we can obtain a task partition plan using the proposed algorithm because it fails only on the condition that all machines available memory are used up and there is still some workload left. Suppose ψ_o is an optimal task partition plan for a parallel program with workload W and $\psi = \{w_1, w_2, ..., w_q\}$ is the task partition plan we obtain using the proposed algorithm. $\{\overline{m}_1, \overline{m}_{2,} ..., \overline{m}_r\}$ denotes those machines with $r_i = a_i$ under ψ which indicates no more workload can be assigned without breaking the memory availability constraint. Obviously, the workloads on machines, $\{\overline{m}_1, \overline{m}_{2,} ..., \overline{m}_r\}$, under ψ and ψ_o are the same. The left machines, $\{m_1', m_{2,}', ..., m_p'\}$, are machines where $r_i < a_i$. From Lemma 1, we know that the worker process completion time on these machines is the same under ψ_o. Suppose it is $t(\psi_o)$. According to the CPU-memory combined partition algorithm, for $\{m_1', m_{2,}', ..., m_p'\}$, their worker process completion time are also the same. Suppose it is $t(\psi)$. $t(\psi_o) = t(\psi)$. Otherwise $W(\psi_o) \neq W(\psi)$. So the workloads on these machines under ψ and ψ_o are also the same. $\psi = \psi_o$. ψ is an optimal task partition plan for a given machine set. Q.E.D.

3.3. Task partition on shared resources

Utilizing only dedicated resources for Grid applications confines the fully delivery of Grid potential computing power because many resources in Grid computing are shared resources.

How to evaluate the effect of local jobs on a Grid application execution in the situation of resource sharing? In subsection 3.1, we define three parameters, λ, ρ, and σ to describe a machine's local jobs usage pattern. When the memory usage of a Grid application can be satisfied by the available memory space of a resource, the cumulative distribution function of the application execution time on a machine can be calculated as [18]:

$$\Pr(T \le t) = \begin{cases} e^{-\lambda w/\tau} + (1 - e^{-\lambda w/\tau}) \Pr(U(S) \le t - w/\tau \mid S > 0), & \text{if } t \ge w/\tau \\ 0, & \text{otherwise} \end{cases} \quad (3)$$

The mean of the Grid application execution time is:

$$E(T) = \frac{1}{1-\rho} \frac{w}{\tau} \quad (4)$$

To partition a Grid application in a shared environment, the same strategy as that of dedicated environment is applied. Our objective is to balance the workload of the Grid application as much as possible while satisfying the memory constraint. For an iterative parallel application, the task partition problem can be formulated as minimizing the difference between the maximum expected subtask completion time and the minimum expected subtask completion time over all possible partition plans subject to $r_i < a_i$, $(1 \le i \le q)$. The three task partition policies considering resource sharing are given as follows:

```
Assumption: The application workload, $W$, can be arbitrarily partitioned. Each machine in $\{m_1, m_2, ..., m_q\}$ has an available memory space $a_k$ $(1 \le k \le q)$.
Objective: find the subtask workload on machine $m_k$ $(1 \le k \le q)$
------------------------------------------------------------------------
Begin
$W_l = W$, $w_{ck} = 0$, $(1 \le k \le q)$;
$M_r = \{m_1, m_2, ..., m_q\}$
While $W_l \neq 0$ and $|M_r| \neq 0$
    For all machine in $M_r$
        Using formula (1) to calculate $w_k$;
        $w_{ck} = w_{ck} + w_k$;
        Calculate $r_k = f(w_{ck})$;
        If $r_k > a_k$, Then
            $w_k = F(a_k)$, $r_k = a_k$, $M_r = M_r - \{m_k\}$;
    End For
    $W_{assign} = \sum_{k=1}^{q} w_{ck}$; $W_l = W_l - W_{assign}$;
End While
If $W_l \neq 0$ and $|M_r| = 0$, Then return failure signal;
End
```

Figure 1. The CPU-memory combined partition algorithm for an iterative parallel application

CPU-only partition: the CPU-only method partitions the workload of an iterative parallel application so that the expected subtask completion time on different shared machines is the same. Let b denote the worker process completion time. From formula (4), we know $b = \frac{w_i}{(1-\rho_i)\tau_i} + \frac{cw_i}{\gamma_{0,i}}$ (Here $\gamma_{i,j}$ denotes the available link bandwidth between machine m_i and machine m_j). Using the same deriving method, we can obtain the subtask workload on m_k:

$$w_k = \frac{W}{\sum_{i=1}^{q} \frac{(1-\rho_i)\tau_i\gamma_{0,i}}{\gamma_{0,i} + c(1-\rho_i)\tau_i}} \frac{(1-\rho_k)\tau_k\gamma_{0,k}}{\gamma_{0,k} + c(1-\rho_k)\tau_k} \qquad (5)$$

Please notice that formula (1) is a simple form of formula (5). If $\rho_i = 0$ is put (dedicated machine's utilization) into formula (5), formula (1) is generated. NWS [12], a network prediction tool, is applied to estimate the available link bandwidth $\gamma_{i,j}$.

Memory-only partition: the memory-only method partitions the workload of an iterative parallel application according to each machine's available memory space in a given machine set. It is the same as the memory-only partition method in a dedicated environment. Formula (2) is used to calculate the subtask workload on m_k.

CPU-memory combined partition: The CPU-memory partition algorithm discussed in subsection 3.2 is modified to calculate the subtask workload on each machine considering resource sharing. In the while loop, formula (5) is used to calculate the subtask workload instead of formula (1). Since we use the same CPU-memory combined partition strategy, THEOREM 1 also holds for the task partition problem on shared resources.

```
Assumption: a meta-task is composed of a number of independent subtasks, $S_T = \{t_1, t_2, ..., t_p\}$. Each subtask is has a memory usage requirement $r_k$ $(1 \le k \le p)$.
Objective: find a task group $G_k = \{t_{k_1}, t_{k_2}, ..., t_{k_n}\}$ mapped on machine $m_k$ $(1 \le k \le q)$
------------------------------------------------------------------------
Begin
/* $C_k$ denotes the current estimated completion time of the assigned subtasks on machine $m_k$ */
$C_k = 0$, $G_k = \phi$, $(1 \le k \le q)$; $i = 1$;
While $S_T \neq \phi$
  For each $t_i \in S_T$ Do
    $j = 1$;
    While $j < q$
        If $r_i \le a_j$ $E(T_{i,j}) = w_i / [\tau_j * (1-\rho_j)] + \delta_{i,j}$;
        /* $E(T_{i,j})$ is the expected execution time of task $t_i$
        on machine $m_j$ */
        $j = j + 1$;
    End While
    Find machine $m_k$ where $C_k + E(T_{i,k})$ is minimal;
     Use formula (6) to calculate $PRIORITY(t_i, m_k)$;
  End For
  Find a map of $(t_u, m_v)$ where $PRIORITY(t_u, m_v)$ is the maximum.
  $G_v = G_v \cup \{T_u\}$; $C_v = C_v + E(T_{u,v})$;
  $a_k = a_k - r_i$, Update if necessary
End While
Return $G_k$ and $W_k$ $(1 \le k \le q)$;
End
```

Figure 2. CPU-memory combined partition algorithm for a meta-task

3.4. Task partition of meta-task

In Grid computing, task scheduling of a specific class of distributed application, meta-task, has been widely studied [5,8-10]. A meta-task is composed of independent indivisible subtasks, which may share some input files [8]. To adapt the dynamics of Grid environments, a meta-task is iteratively scheduled until its completion. The scheduler itself can be viewed as a master process. The question for meta-task scheduling is how to group subtasks into clusters and assign clusters of subtasks to available machines. The key to task partition of a meta-task is also balancing the workload of subtasks on each machine as much as possible while memory constraint is satisfied. However, because subtasks cannot be further partitioned, we cannot directly apply the above optimal CPU-memory combined partition method, which is based on the assumption that the application workload can be arbitrarily partitioned during each iteration. In general, task scheduling of a meta-task is NP-complete. A heuristic algorithm is thus proposed. Figure 2 gives the max-min CPU-memory combined partition algorithm for a meta-task. The intuition behind this heuristic is two-fold: a long subtask scheduled in the end would delay the whole execution; a subtask with a large memory demand scheduled in the end would introduce uneven workload distribution. The basic process of this algorithm includes two steps. In the first step, for each subtask, t_i, we find its favorite machine, m_k, where the subtask can be completed earliest. In the second step, we find a map of (t_u, m_v) where $PRIORITY(t_u, m_v)$ is the maximum. The $PRIORITY(t_i, m_j)$ is defined as:

$$PRIORITY(t_i, m_j) = \alpha * T_{i,j} + \beta * r_i \qquad (6)$$

where $T_{i,j}$ is the expected completion time of t_i on m_j and $\alpha + \beta = 1$. In this algorithm, $\delta_{i,j}$ denotes the file transfer time of subtask t_i on machine m_j. The file transfer time depends on the file size and the available bandwidth between machine m_0 and machine m_i. For more details on the calculation of $\delta_{i,j}$, people can refer to [8]. We implement the CPU-only partition policy for a meta-task by extending the min-min task group algorithm [10] with the addition of memory usage check after each subtask assignment. In the memory-only partition method for a meta-task, each machines is assigned with a cluster of subtasks. The sum of these subtasks' workloads is proportional to the machine's available memory space.

4. Experiment Result

We verify the efficiency of the proposed scheduling algorithms in a simulation environment and an actual distributed system. The application is scheduled in both environments with different task partition approaches. We use the application completion time to evaluate the performance of task scheduling algorithm.

In this simulation environment, the arrival rate of local jobs on each machine's follows Poisson distributions. The local job's lifetime is simulated with $2.0/x$ [6], which follows the observation of real-life processes in [19]. x is a random number between 0 and 1. The job arrival rate and service rate are randomly generated on each machine so that different machines have different CPU usage patterns. We simulate three categories of machines with different ranges of available memory space: machines with high available memory space (400M~2G), machines with medium available memory space (100M~400M), and machines with low available memory space (0~100M). A machine's available memory space is randomly generated in these three ranges. Because the scheduling process during each iteration of a Grid application execution is the same, we test the performance of task scheduling algorithm in one iteration in our environment.

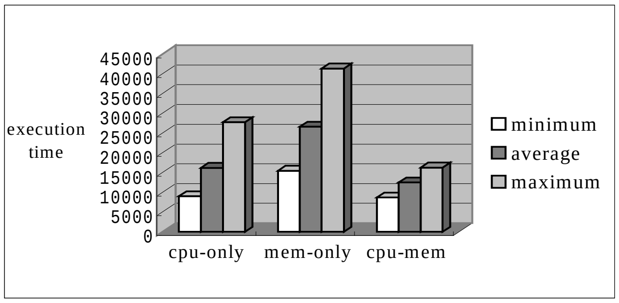

Figure 3. The Average, Minimum, and Maximum Performance of an iterative parallel application

Figure 3 presents the average of the application execution time of an iterative parallel application scheduled with different task partition policies. The number of machines is 20. The simulation is executed 20 times. The min and max of the application running time are given. We can see that the CPU-memory combined scheduling approach outperforms either the CPU-based or memory-based scheduling approach considerably. The performance gain of the CPU-memory combined approach is around 23% and 53% compared to CPU-based and the memory-based approach, respectively.

The effect of memory availability on the performance of task scheduling algorithm is shown in Figure 4. In the above simulation, the three different ranges of memory space availability can be represented with a triple (100, 400, 2000). We adjust the range of available memory space and measure the average of the application execution time with the three partition policies. The numbers on the X-axis stand for five different memory resource availabilities: (50, 200, 1000), (75, 300, 1500), (100, 400, 2000), (150, 600, 3000), and (200, 800, 4000). The experiment results show that CPU-only partition is much sensitive to memory resource availability compared with other partition policies. When the available memory space is reduced to some extent, the average application running time is increased dramatically. However, with the increase of available memory space, the performance of the scheduling algorithm with the CPU-only partition is approaching to CPU-memory combined scheduling approach while the performance of the memory-only partition approach is still low. This conforms our analysis that when the available memory space on each machine is large enough to accommodate any assigned workload, the CPU-only approach is the same as the CPU-memory approach.

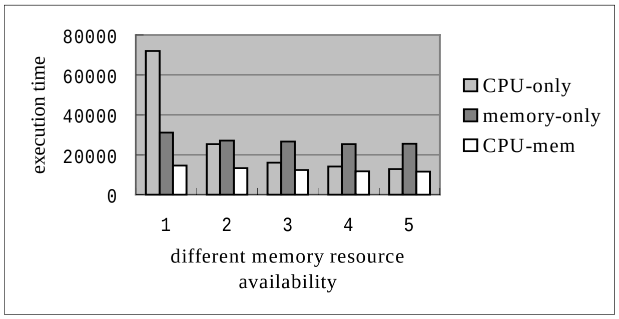

Figure 4. Comparison of Average Execution Time

Experiments are conducted to evaluate the meta-task scheduling in a simulation environment and an actual system. We measure the completion time of a meta-task scheduled with the three partition approaches in a simulation environment similar to that in [8]. We assume that the system is composed of a group of clusters. Each cluster has a file server. Each subtask of a meta-task has some input files. The input files of a subtask must be transferred to the machine where the subtask is assigned before its execution. Because the effect of link bandwidth heterogeneity on the application scheduling decision is not the focus of this work, we assume that all link capacity is the same in our simulation environment. The map of files and subtasks is randomly generated during each simulation. We also observed that CPU-memory combined scheduling approach outperforms either the CPU-based or memory-based scheduling approach considerably in two classes of memory availabilities: (100, 400, 2000) and (200, 800, 4000). Table 1 shows that the average performance gain of the CPU-memory combined approach is around 49% and 57% compared to the CPU-based and memory-based approach, respectively.

Table 1. Average, Minimum, and Maximum Execution Time of a Meta-task

Application completion time	(100, 400, 2000)			(200, 800, 4000)		
	CPU-only	Mem-only	CPU-mem	CPU-only	Mem-only	CPU-mem
Minimum	9702	9139	7172	9702	9139	5425
Average	31351	36850	15856	19813	22980	10038
Maximum	69268	142069	45479	62544	59914	27265

An actual heterogeneous computing environment is used to examine the performance of three scheduling methodologies for meta-task scheduling in different system configurations. The distributed system is composed of two clusters (Sunwulf and Ares) and two servers (Scala and Meta). Sunwulf is composed of a server (Sun-enterprise 450, 4G physical memory space, SunOS 5.8) and 64 computational nodes (Sun-blade 100, 128M physical memory space, SunOS 5.8). The server and 5 nodes are used in our experiment. Ares is an IBM-xSeries 1350 on Linux OS, composed of a server and 14 computational nodes. All of them have 4G physical memory spaces. The server and 5 nodes are used in our experiment. The two other servers are Scala (Sun-ultra 10, 256 physical memory space, SunOS 5.7) and Meta (Sun-fire 280, 1G physical memory space, SunOS 5.8). The available memory space on each of these machines is random generated within their physical memory spaces. The meta-task to be scheduled is composed of a series of NAS Benchmarks (BT, CG, LU, MG, IS and SP). The class types of these benchmarks are, in general, “B”, “A”, and “W”. We assume that 10 input files are needed for these benchmarks. The available bandwidth from the meta-task submission machine to Sunwulf, Ares, Scala, and Meta are around 110Mbps, 110Mbps, 560Mbps, and 110Mbps respectively. Figure 5 shows the execution time of a meta-task with three scheduling approaches. In all the three system scenarios, dedicated, shared, and mixed (two cluster servers are dedicated while others are shared), the CPU-memory combined scheduling approach outperforms either the CPU-based or memory-based scheduling approach.

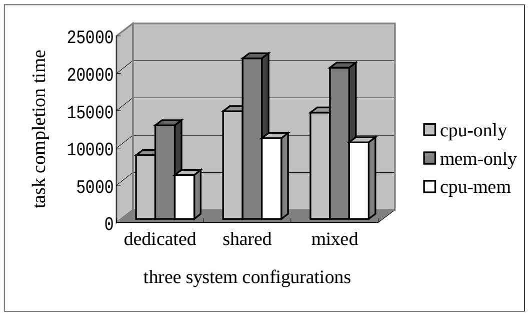

Figure 5. Meta-task completion time under three system configurations

In our experiment, the task partition and scheduling algorithm are executed on a Pentium 2.66GHz PC. The cost of task partition is too small to be noticeable. We measure the cost of task scheduling of an iterative parallel program in finding an optimal solution with a branch-and-bound method. The cost is around 11 seconds when the system size is 20. When the system size is beyond 20, a heuristic algorithm proposed in our previous work [9] can be applied to further reduce the scheduling cost. Experiment result shows that the heuristic task scheduling cost is only 4.3 seconds when the system size is 80.

5. Conclusion

In this study, we investigate memory-conscious task scheduling for iterative parallel applications in Grid environments. A specific class of iterative parallel applications, meta-task, is also investigated. Three task partition policies are discussed. Analytical analysis is given to confirm the correctness and effectiveness of the proposed CPU-memory combined partition. In our experiments in both a simulation environment and an actual distributed system, we observed that the average performance gain of the CPU-memory combined approach is around 23% and 49% compared to CPU-based and 53% and 57% compared to the memory-based approach for task scheduling of a parallel application and a meta-task, respectively.

The newly proposed memory-conscious task scheduling approach is a complement of existing task scheduling systems. It can be integrated into existing toolkits [5,9] for more appropriate task scheduling for memory intensive applications. Like most existing task scheduling systems, the current implementation of the proposed scheduling system has its limitations. For instance, the estimate of communication costs for data transfer among processes is relatively simple. More analysis of system communication architecture is needed to improve the prediction accuracy. To enable memory-conscious scheduling in Grid environments, we plan to integrate the proposed task partition and scheduling approaches into Grid resource management and job submission services [20] to investigate it further on large engineering applications.

Acknowledgment

This research is supported in part by National Science Foundation under NSF grant EIA-0130673, ANI-0123930, and by Army Research Office under ARO grant DAAD19-01-1-0432.

Reference:

[1] M. Litzkow, M. Livny, and M. Mutka, “Condor - a hunter of idle workstations,” *Proc. of the 8th International Conference of Distributed Computing Systems*, pp. 104-111, June 1988.

[2] H. Casanova and J. Dongarra, “NetSolve: a network server for solving computational science problems,” *The International Journal of Supercomputer Applications and High Performance Computing*, Vol. 11, No. 3, pp. 212-223, Fall 1997.

[3] Abramson D., Sosic R., Giddy J., and Hall B., “Nimrod: a tool for performing parametised simulations using distributed workstations,” *The 4th IEEE Symposium on High Performance Distributed Computing*, Virginia, Aug. 1995.

[4] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure.* ISBN 1-55860-475-8, July 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] Y. Amir, B. Awerbuch, A. Barak, R. S. Borgstrom, and A. Keren, “An opportunity cost approach for job assignment in a scalable computing cluster,” *IEEE Trans. Parallel Distrib. Systems,* 11 (2000) 760-768.

[7] L. Xiao, S. Chen, and X. Zhang, “Dynamic cluster resource allocations for jobs with known and unknown memory demands,” *IEEE Trans. Parallel Distrib. Systems*, 13 (2002) 223-240.

[8] Henri Casanova, MyungHo Kim, James S. Plank, and Jack Dongarra, "Adaptive scheduling for task farming with grid middleware," *The International Journal of High Performance Computing*, Vol. 13, No. 3, pp. 231 - 240, Fall 1999.

[9] X.-H. Sun and M. Wu, “Grid Harvest Service: A System for Long-Term, Application-Level Task Scheduling,” *Proc. of 2003 IEEE International Parallel and Distributed Processing Symposium*, Nice, France, April 2003.

[10] M. Wu and X.-H. Sun, “A general self-adaptive task scheduling system for non-dedicated heterogeneous computing,” *Proc. of IEEE Cluster Computing Conference*, Hong Kong, 2003, pp. 354-361.

[11] I. Foster, A. Roy, and V. Sander, “A quality of service architecture that combines resource reservation and application adaptation,” *International Workshop on Quality of Service*, pp. 181-188, June 2000.

[12] 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.

[13] R. Buyya, D. Abramson, and J. Giddy, “Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid,” *HPC Asia 2000*, Beijing, China, May 2000.

[14] K. D. Ryu, J. K. Hollingsworth, and P. J. Keleher, “Mechanisms and policies for supporting fine-grained cycle stealing,” *International Conference on Supercomputing*, Rhodes, Greece, pp. 93-100, 1999.

[15] Mutka, M. and M. Livny, “The available capacity of a privately owned machine environment,” *Performance Evaluation*, Vol. 12, No 4, pp. 269-284, 1991.

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

[17] Z. Lan, V. Taylor, and G. Bryan, “Dynamic Load Balancing of SAMR applications on Distributed Systems”, *Proc. of SC'2001*, Denver, CO.

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

[19] M. Harchol-Balter and A. Downey, “Exploiting Process Lifetime Distributions for Dynamic Load Balancing,” *ACM Trans. Computer Systems,* 15 (1997) 253-285.

[20] J. Reich, I. Foster, D. Gannon, and H. Kishimoto, “Open Grid Services Architecture Use Cases”, GGF documents, 2004.