Gang Scheduling in Heterogenous Distributed Systems

Khaled Al-Saqabi, Steve W. Otto and Jonathan Walpole[1]

Abstract — This paper presents an algorithm for scheduling parallel applications in large-scale, multiuser, heterogeneous distributed systems. The approach is primarily targeted at systems that harvest idle cycles in general-purpose workstation networks, but is also applicable to clustered computer systems and massively parallel processors. The algorithm handles unequal processor capacities, multiple architecture types and dynamic variations in the number of processes and available processors. Scheduling decisions are driven by the desire to minimize turn around time while maintaining fairness among competing applications. For efficiency, the virtual processors (VPs) of each application are gang scheduled on some subset of the available physical processors.

Index Terms— Scheduling, distributed systems, heterogeneity, process migration, concurrent processing, idle cycle stealing.

1. INTRODUCTION

Recent years have witnessed rapid advances in the performance of commodity micro-processor and network hardware. From a parallel computing perspective, one significant impact of these advances is that general purpose computer networks are already becoming viable platforms for running high performance parallel applications [1-6]. The Parallel Virtual Machine (PVM), P4, Linda and others [5] are examples of software systems that support such functionality.

While the problem of scheduling parallel applications on distributed computing systems is already well-explored [1], most existing approaches focus on dedicated, homogeneous environments, such as massively parallel processors (MPPs). From a scheduling perspective, general purpose computer networks differ from MPPs in two key respects: (a) they are usually composed of heterogeneous processors, and (b) individual processors are usually owned by a specific user or

1. Khaled Al-Saqabi is with the University of Kuwai, and is on sabbatical at the Oregon Graduate Institute, Portland, OR 97006; e-mail saqabi@cse.ogi.edu.
Steve Otto and Jonathan Walpole are with the Department of Computer Science and Engineering, Oregon Graduate Institute, Portland, OR 97006; e-mail: otto@cse.ogi.edu, walpole@cse.ogi.edu.

group of users. Both of these characteristics add significant complexity to the scheduling problem.

Heterogeneity complicates the scheduling problem in several ways. First, different processors can have unequal processing capacities and hence an even distribution of work among the available processors will not usually result in correct load-balancing. Second, variations in architecture and instruction set among the available processors impose hard constraints on the choice of targets for creating new virtual processors (VPs) or migrating existing ones.

The concept of ownership further complicates the scheduling problem because allocation decisions made by the scheduler may be dynamically invalidated by processor owners. In workstation networks, for example, a processor's availability for running parallel jobs typically depends on it being otherwise idle [2]. When the owner of an idle processor returns to use it again it may be necessary to invoke the scheduler to evict any currently resident VPs and reassign them to other processors. Responsive eviction is an important requirement for unobtrusive idle cycle stealing [7]. Consequently, scheduling decisions must be made quickly and it must be possible to migrate VPs dynamically.[2] Scheduling algorithms for environments in which ownership is an issue must be capable of handling dynamic and independent variations in the number of available processors and VPs.

This paper presents a scheduling algorithm which satisfies the above heterogeneity and ownership constraints and allocates processing resources to parallel jobs such that the job's turn around time is minimized and fairness among competing jobs is maintained. For practical reasons, we do not assume that a job's execution time requirements are known in advance.

The remainder of the paper is organized as follows. Section 2 compares our work with existing research in distributed scheduling. Section 3 outlines the model on which our scheduling algorithm is based. The scheduling algorithm is presented in Section 4. Finally, Section 5 concludes the paper.

2. RELATED WORK

Existing scheduling algorithms can be categorized as either shared memory multiprocessor

2. We assume the existence of a migration mechanism that is capable of migrating VPs among equivalent processor architectures at any stage during their execution [8-10]. However, we also assume that such a mechanism is heavyweight and should not be invoked frequently.

approaches [11, 12] or distributed systems approaches [1, 13, 14]. This paper is concerned with distributed systems approaches. Distributed systems approaches can be subdivided into approaches for homogeneous or heterogeneous environments. Most existing research falls into the homogeneous category in which all processors are assumed to be equivalent in terms of processing capacity and architectural characteristics. This paper addresses heterogeneous environments. Finally, schedulers can be further categorized according to whether they make allocation decisions statically or dynamically. Static approaches map VPs onto processors based on a set of characteristics defined at job submission time. These characteristics include, for example, the number of VPs in the job and the number and configuration of the available processors. Dynamic approaches do not require a priori knowledge of the system or application characteristics. Instead, the scheduler adapts to changes by dynamically re-mapping VPs to processors.

The algorithm presented in this paper addresses the problem of how to reassign VPs to processors dynamically in a heterogeneous distributed environment. That is, we are concerned primarily with the question of how to choose a good mapping following a change in the number of VPs or processors. The main motivation for this work is the development of a real-world distributed scheduler for use in our migratable PVM system [10]. This paper does not directly address the problem of distributing the scheduling algorithm itself. However, many of the ideas, discussed below, for distributing the scheduling decision are applicable to our scheduling algorithm.

Most existing dynamic homogeneous scheduling approaches target load-balancing as the main motivation for dynamic reassignment and differ according to their accuracy and the amount of processor load information they exchange [15]. Zhou's algorithm [16] balances load by periodically requiring each processor to inform other processors of load changes. The scheduler is invoked whenever a new VP is submitted. If the local load is below a threshold value Th_1 the VP is executed locally. Otherwise the least loaded node in the system is examined. If its load is less than the local load by at least a threshold Th_2, then the VP is scheduled on that processor. Otherwise, it is executed locally. Four heuristics, presented by Xu [17], reduce the overhead of this scheme at the expense of accuracy, by allowing either neighboring processors or all processors in the system to contribute to the setting and adjustment of the threshold values for a given node. Kremien et al [18] improve the scalability and stability of Zhou's algorithm by subdividing the

system into domains and only exchanging load information among processors in the same domain.

Ahmad's algorithm [19] is a compromise between the centralized and the distributed load balancing. System processors are assigned to independent symmetric regions, called spheres. In each sphere, a processor equadistant from all other processors is selected as the scheduler for that sphere. Cumulative load information for each sphere is exchanged among the independent schedulers. Using local threshold values, as well as cumulative load figures from other spheres, a task is either scheduled locally where submitted, or transferred to a less loaded sphere. Simulation results indicate an improvement in average response time compared with the fully distributed version of the same balancing algorithm.

Suen [20] proposed a balancing algorithm and communication protocol which improve average response time and reduce communication overhead. Each processor communicates its load directly to only $N^{1/2}$ of the N processors in the system. Each processor has two sets of processors, its sending set and its receiving set, that it sends to and receives from respectively. The sets are constructed such that load balancing information is propagated either directly or indirectly to all processors in the system. Other researchers [21] capitalize on multi access local area networks in order to implement the search for the minimum/maximum loaded node in constant time, regardless of the number of processors on the network.

Finally, Willebeek-LeMair [22] presented four scheduling policies: (i) sender/receiver initiated distribution which performs balancing based on information from neighboring processors. (ii) Hierarchical balancing methods which organize systems into a hierarchy of subsystems within which balancing is performed. (iii) The Gradient model which guides migration between overloaded and under loaded nodes through a proximity gradient which eventually transfers load from heavily to lightly loaded regions, and (iv) The Dimension Exchange method which first requires a synchronization phase, then balancing is performed iteratively.

More centralized, static approaches to scheduling in heterogeneous environments are supported in the Load Sharing Facility (LSF), Utopia, [23], Distributed Queuing System (DQS) [24], and Prospero Resource Manager (PRM) [25]. These systems are widely used in practice, and support the mapping of VPs to processors at job submission time. However, they do not support the con-

cept of dynamic migration. Consequently, they are more appropriate for dedicated, or otherwise idle, environments than for continual use in general purpose multiuser networks.

Condor [7] is the only system we are aware of that supports dynamic heterogeneous scheduling. However, it is oriented towards scheduling single VP jobs in distributed systems. We believe that research is underway to integrate Condor with PVM in order to schedule parallel applications. However, details of the scheduling algorithm used were not available at the time of writing.

In this paper, we present an algorithm for scheduling parallel applications in multiuser heterogeneous systems. The scheduler is invoked dynamically to reassign VPs to processors when processors become available, are reclaimed by their owners, and when VPs are created and destroyed. In the current version of the algorithm, scheduling decisions are made on a single processor. Further work to distribute the scheduler is underway, but is outside the scope of this paper.

3. MODEL

To illustrate the basic principles underlying our scheduling algorithm, we start out by presenting a simplified model in which all processors are of the same architecture and equal processing power. The complete algorithm presented in Section 4 extends this model to heterogeneous environments.

The scheduler is invoked in response to four kinds of event: *processor_exit*, *new_processor*, *new_VP* and *VP_exit*. A processor_exit event occurs, for example, when a processor is reclaimed by its owner and all its VPs must be migrated to other processors. The effect of a processor_exit is to remove the processor from the pool of processors managed by the scheduler. A new_processor event is the inverse of a processor_exit event. In other words, it is an event that signals the addition of a new processor to the pool of processors managed by the scheduler. A new_VP event occurs when an application creates a new VP. A VP_exit occurs when an application terminates a VP. In the remainder of the paper we refer to the collection of VPs belonging to the same application as a *job*.

At any point in time, the state of the system can be illustrated using a two-dimensional *allocation map* in which one dimension represents the available processors and the other represents time. Each entry in the allocation map is either occupied by one or more VPs of a job, in which

case the corresponding processor is assigned to that job during that time slice, or it is free, in which case the corresponding processor is unused during that time slice. For example, Figure 1 represents the state of a system with six processors and eight jobs. The VPs of job J1 are allocated time slice T1 on all processors, whereas jobs J2 and J3 are assigned to disjoint subsets of the available processors during time slice T2. Figure 1 also shows that processors P5 and P6 are free during time slices T3 and T4.

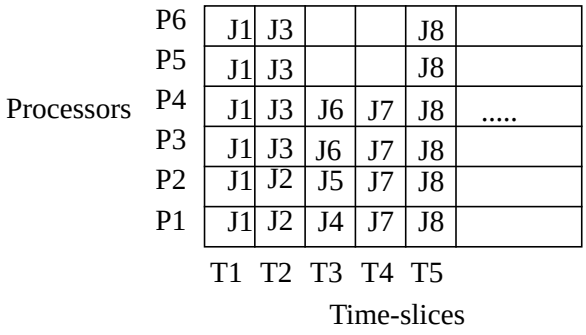

Figure 1. Model for gang scheduling multiple jobs on a set of processors.

Using this model, a processor_exit event corresponds to the removal of a row from the allocation map, and a new_processor event corresponds to the addition of a row to the allocation map. A new_VP event corresponds to the assignment of a new VP to an entry in the allocation map and may cause the addition of a new column if it is the first VP of a new job. Similarly, a VP_exit event corresponds to the removal of a VP from an entry in the allocation map and may cause the removal of a column if none of its entries are assigned. The role of the scheduling algorithm is to decide how to manipulate the allocation map in response to these events.

A number of assumptions influenced the design of our scheduling algorithm. First, we assume that the VPs of a single job communicate frequently and hence will benefit from gang scheduling [26]. Gang scheduling requires all the VPs of a single job to execute at the same time. Hence, we use time slices that extend across processors and ensure that all the VPs of a single job are allocated in the same time slice. In terms of the allocation map, this approach implies that if one VP from a job resides in a column, all other VPs of that job must also reside in the same column. Note that if a single job occupies multiple columns, all its VPs must be replicated in each of those columns.

When a new_VP event occurs a processor must be selected as the target for executing the newly created VP. Similarly, when a processor_exit event occurs the scheduler must select new processors as targets for migration of the displaced VPs. If no free processors are present in the required time slices, the scheduler may choose to assign the new or displaced VPs to one or more of the other processors assigned to the same job. In the following discussion we refer to this procedure as *doubling up*.

If VPs are doubled up on a processor it may be possible to double up on several other processors, hence freeing processors, without further impacting the turn around time of the job. For example, consider a processor_exit event on processor P1 in Figure 1. If we place two VPs of job J1 on processor P2 in time slice T1 the net effect will be to double the turn around time for job J1[3]. Taking job J1's VPs from processors P5 and P6 and placing them on P3 and P4 will not further increase the turn around time, but it will free up processors P5 and P6 in time slice T1. These processors could then be used by other jobs. We refer to this concept as *compressing* the job's processor set.

On a new_processor event the scheduler must determine whether to migrate existing VPs to the new processor. Similarly, on a VP_exit event the scheduler must decide whether to use the resulting free capacity for other jobs. From the discussion in the previous paragraph it can be seen that both processor_exit and new_VP events can also cause processors to be freed through compression. Therefore, all four scheduling events can potentially cause the scheduler to consider migrating existing VPs to new processors. We refer to the migration of existing VPs as *expanding* a job's processor set.

Since the value of expanding a job's processor set depends on the relationship between the cost of migration and the remaining execution time of the job, and since we assume no fore knowledge of a job's execution time, it is impossible to know whether expansion will be worthwhile. For example, if the remaining execution time for a job was very small and the migration cost was very high, the speed up resulting from expansion would not amortize the migration cost. Therefore, we take the following simplistic approach: if a job can not be speeded up through expansion we do not change its current allocation. If the job can be speeded up we assume that it will run for long

3. This example assumes an initial ssignment of one VP per processor for job J1.

enough to offset the migration cost.

Compression and expansion are relatively straight forward in homogeneous environments, but become complex in heterogeneous environments where processors have unequal processing capacity and incompatible architectures. The discussion so far has assumed a homogeneous environment. The algorithms presented in the next section extend this basic approach to heterogeneous environments with the following characteristics. First, the architecture types and relative processing capacities of all processors are known. In practice, we would obtain an estimate of the processing capacity by periodically running a simple benchmark on each processor and passing the results to the scheduler. Second, the choice of target processors for VP migration and the creation of new VPs is constrained by architecture type.

4. THE SCHEDULING ALGORITHM

The overall scheduling algorithm comprises three basic algorithms: the Minimum Turn Around Time (MTAT) algorithm, the Compression algorithm, and the Expansion algorithm. These basic algorithms are employed in handling all four scheduling events. The MTAT algorithm determines the minimum turn around time for the job, T_{min}, assuming that it is maximally distributed across the set of available processors Φ in a single time-slice. The Compression algorithm then attempts to achieve T_{min} using a smaller set of processors Φ_{min} in the same time slice. The result of the Compression algorithm is an allocation of VPs to processors that yields the minimum turn around time using the smallest number of processors. However, before the job is allocated to that set of processors in a new time slice, the Expansion algorithm is used to explore the use of existing free space in the allocation map. The result of the Expansion algorithm, specifically, the processor set containing suitable free space, is then passed back to the MTAT algorithm to calculate an alternative completion time for the job. If the job can be completed faster using existing free space, the Compression algorithm is called to attempt to compress the new set of processors, and the job is assigned to those processors during the appropriate time slice(s). Otherwise, the job is allocated a new time slice.

The advantage of this approach is that it is both efficient and fair in that it searches first for solutions that minimize the turn around time of the submitted job without impacting existing jobs, and

then falls back on solutions that continue to minimize the turn around time for the submitted job but impact all jobs equally. The relationship between these algorithms is illustrated in Figure 2.

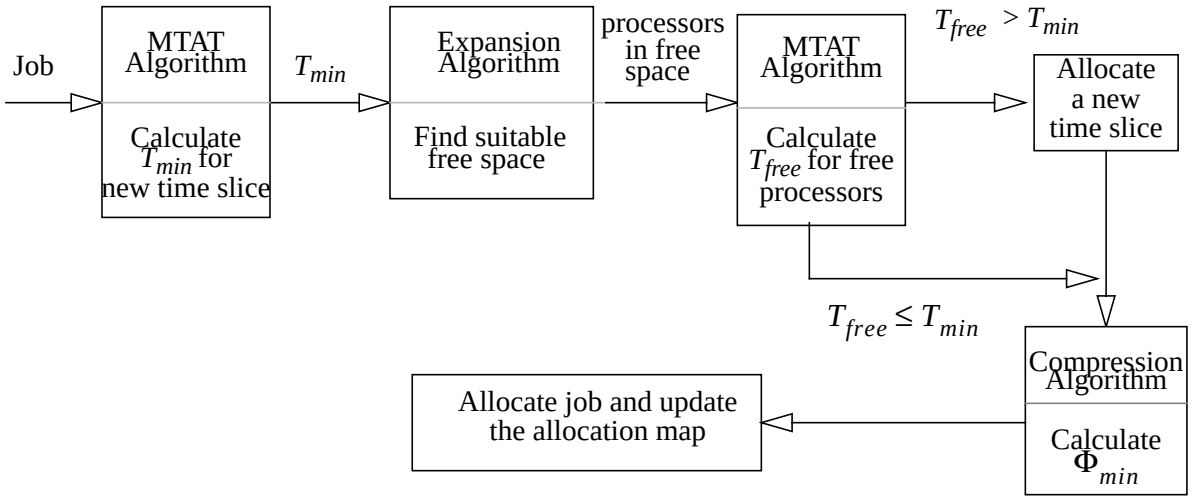

Figure 2. Relationships among the basic scheduling algorithms.

4.1. The Minimum Turn Around Time (MTAT) Algorithm

The MTAT algorithm has two stages. The first stage determines the ideal load distribution across the set of available processors. The second stage modifies this load distribution to take into account constraints imposed by VP granularity.

To distribute load fairly requires some knowledge of the relative processing capacity of each of the available processors. Let Φ_{total} be the set of all the processors in the system[4]. The relative processing capacity a_i of a processor P_i is defined with respect to the slowest processor in Φ_{total} as follows:

$$a_i = \frac{Capacity\ (P_i)}{Capacity\ (slowest\ processor\ in\ \Phi_{total})}$$

Let Φ be the set of currently available processors such that $\Phi \subseteq \Phi_{total}$. The fraction

4. Note that some of these processors may not be available to the scheduler at any particular time

$$\frac{a_i}{\sum_{\forall i \in \Phi} a_i}$$

represents the portion of the system's currently available processing capacity contributed by processor *i*. If *X* is the number of VPs in a job, then the real number x_i of the job's VPs that should be assigned to processor *i* is:

$$x_i = X\left(\frac{a_i}{\sum_{\forall i \in \Phi} a_i}\right) = a_i\alpha \tag{1}$$

where α is the job's minimum possible turn around time. Note that this turn around time assumes that load can be balanced exactly equally among the available processors in Φ. This assumption is generally not valid when the number of VPs is independent of the number of processors and their processing capacity. Therefore, the second stage of the MTAT algorithm must calculate a real-world distribution that takes into account VP granularity. To achieve this goal, x_i must be forced to an integer value. In other words, a real-world solution can not assign fractions of a VP to a processor.

The result of making x_i an integer will be an increase in the turn around time for the job since one or more of the processors will become a bottleneck. Let T_i be the completion time for the portion of the job assigned to processor P_i. The real-world minimum turn around time for the job is:

$$T_{min} = \underset{\forall i \in \Phi}{Max} \{T_i\}$$

The second stage of the MTAT algorithm determines the allocation of VPs to processors that minimizes T_{min}. This algorithm has the following five steps:

1. For each processor, determine a lower bound $\tilde{x}$ on the number of VPs that can be allocated to it in an optimal solution:

$$\tilde{x}_i = \lfloor x_i \rfloor, \forall i \in \Phi. \tag{2}$$

2. Calculate the number of VPs, *Diff*, left unallocated following step 1:

$$Diff = X - \sum_{\forall i \in \Phi} \tilde{x}_i$$

3. For each processor, determine the effect on overall turn around time of the job, called *Drag*, that would result from allocating an additional VP to that processor:

$$Drag_i = \frac{(\lceil x_i \rceil - x_i)}{a_i}, \forall i \in \Phi. \quad (3)$$

4. Order the processors in Φ based on *Drag* and select the *Diff* lowest processors. Allocate one of the remaining VPs to each of those processors. Let $\tilde{x}_{if}$ be the final allocation of VPs to processor i and T_{if} be the completion time for those VPs on processor i. Then

$$T_{if} = \alpha + \frac{(\tilde{x}_{if} - x_i)}{a_i} \quad \text{and} \quad T_{min} = \underset{\forall i \in \Phi}{Max} \{T_{if}\}$$

5. For any $i, j \in \Phi$, if $Drag_i = Drag_j$ and $(\tilde{x}_i = 0) \neq \tilde{x}_j$ with $Diff \neq 0$, then assign VPs to j before assigning them to i. This step biases the selection towards the minimum number of processors.

Example 1: Consider a job with $X = 20$. Let the relative processing capacity of the available processors $\{a_i\} = \{10, 1, 4, 3\}$. Then:

$$\alpha = \frac{X}{\sum a_i} = \frac{20}{18} = 1.111$$

$$\{x_i\} = \{a_i\ \alpha\} = \{11.11, 1.111, 4.444, 3.333\}$$

$$\{\tilde{x}_i\} = \{11, 1, 4, 3\}$$

and

$$Diff = X - \sum \tilde{x}_i = 1$$

Step 4 of the MTAT algorithm yields $\{Drag_i\} = \{0.089, 0.889, 0.139, 0.222\}$. Thus the extra VP is allocated to processor 1 and $\{\tilde{x}_{if}\} = \{12, 1, 4, 3\}$. Consequently, the completion times on

each processor $\{T_{if}\}$ are {1.2, 1, 1, 1} and the overall turn around time T_{min} for the job is 1.2.

We now show that the MTAT algorithm yields an assignment of VPs to processors that results in the minimum turn around time for a job.

Lemma 1: Equation 1 represents the lower bound turn around time for a job on Φ.

Proof: Equation 1 distributes VPs evenly to the processors in Φ based on their relative processing power. Hence all processors complete their work in exactly $x_i/a_i = \alpha$ time units. Moving work from one processor to another can only increase the completion time for the receiving processor and hence for the complete job. ❑

Since we assume that a VP can not execute on more than one processor at the same time, it is necessary to run the remainder of the MTAT algorithm if Equation 1 results in the assignment of a non-integer number of VPs to any processor.

Theorem 1: The MTAT algorithm finds the minimum turn around time for integer VP assignments.

Proof: From Lemma 1, the lower bound on T_{min} is α. The first step of the MTAT algorithm (Equation 2) starts with the VP assignments needed to achieve α and rounds them down to the nearest integer. Since this first step either reduces or leaves unchanged the amount of work at each processor, all processors will complete in α or less time units. Since Step 1 reduces the load at each processor by at most one VP, the number of VPs remaining to be assigned will be smaller than the number of processors. Since the MTAT algorithm computes the slow-down that would be caused by adding a VP to each processor in turn, and then selects the processors that will contribute least to the slow-down, it finds the assignment resulting in the minimum value for T_{min}. ❑

The above description of the MTAT algorithm assumes that all processors are architecturally equivalent. That is, it assumes that any of the VPs of a job can be executed on any processor. In reality, system heterogeneity will impose restrictions on where VPs can execute. These restrictions arise for two reasons: (a) when a new VP is spawned an executable image may not exist for all processor architectures, and (b) dynamic VP migration can usually only take place between processors with equivalent architectures because of the difficulty in translating state information relating to the VP's current context from one processor architecture to another. Consequently, the

MTAT algorithm must be extended to deal with disjoint pools of processors with equivalent architecture.

The basis for extending the MTAT algorithm is simple: call it once of each architecture pool and return the largest turn around time as T_{min}. The complete algorithm is as follows: let Z be the set of distinct architecture types represented in the set of available processors Φ, let Φ_z be the set of available processors of architecture type *z*, let X_z be the number of VPs of a job that are restricted to architecture type *z* and let T_z be the minimum turn around time of a job on Φ_z. Then T_{min} = max (T_z, $\forall z \in Z$), where T_z = MTAT (X_z, Φ_z).

Theorem 2: The heterogeneous MTAT algorithm returns the minimum turn around time for a job on a heterogeneous set of processors.

Proof: Since the largest turn around time in each architecture pool defines a lower bound on T_{min}, and since, by Theorem 1, the MTAT algorithm finds the smallest turn around time for each architecture pool, the heterogeneous MTAT algorithm finds the minimum turn around time for the job on the heterogeneous processor set.❑

The worst case complexity of the MTAT algorithm is $\Theta(Zn_z)$, where n_z is the number of processors of a given architecture available to the scheduler.

4.2. The Compression Algorithm

Although the MTAT algorithm yields the minimum possible turn around time, further work is required to determine the minimum set of processors $\Phi_{min} \subseteq \Phi$ that are necessary in order to achieve T_{min} (see Example 2).

Example 2: Consider an environment with 3 processors where $X = 4$ and $a_1 = a_2 = a_3 = 1$ (i.e., the processors are homogenous). In this case, $\alpha = 4/3$; $\{x_i\} = \{a_i\alpha\} = \{1.33, 1.33, 1.33\}$, $\tilde{x} = \{1, 1, 1\}$ and *Diff* = 4 - 3 = 1. Since *Drag* ={0.67, 0.67, 0.67}, the remaining VP can be granted to any of the three processors, yielding $T_{min} = 2$. Note, however, that the same value for T_{min} can be obtained using only two processors, each with two VPs (see Figure 3). Although the MTAT algorithm yields the minimum possible turn around time, it does not necessarily yield the minimum possible Φ

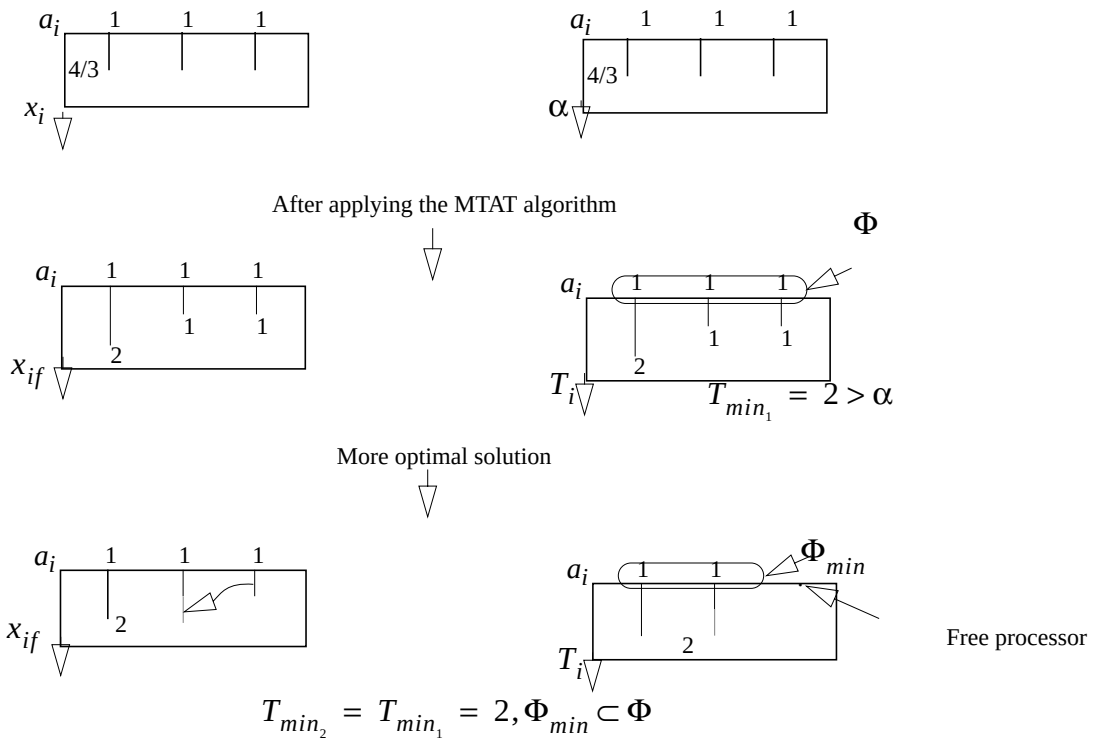

Figure 3. Optimizing the solution of Example 2.

The Compression algorithm compresses the set Φ , while maintaining T_{min}, in the following way:

1. Calculate an upper bound on the VP assignment $x_{i_{max}}$ at each processor in a solution that continues to maintain T_{min}:

   $$x_{i_{max}} = T_{min} a_i, \forall i \in \Phi$$

   If the VP assignment at any processor exceeds $x_{i_{max}}$ the completion time at that processor will exceed T_{min} and the solution will not be valid.

2. Identify the processors that can not have an integer number of VPs assigned in a solution that meets T_{min} and do not consider them further. In other words, find the set of processors $\Phi_{min} \subseteq \Phi$ such that $\Phi_{min} = \Phi - \{ i | \lfloor x_{i_{max}} \rfloor = 0 \}$.

3. If the total processing capacity (available at complete VP granularity) on all the processors in Φ_{min} exceeds the number of VPs in the job then attempt to reallocate VPs to free up as many processors as possible. To achieve this goal, first construct the set of processors Φ_T that have enough excess capacity to accommodate one or more additional VPs. Then visit

all processors in order of increasing VP allocation and attempt to redistribute their VPs to the processors in Φ_T until the free capacity is insufficient to free up a processor. The details of this step are as follows:

if $\sum_{\forall i \in \Phi_{min}} \lfloor x_{i_{max}} \rfloor > X$ **then**

$free = \sum \lfloor x_{i_{max}} \rfloor - X$

for all processors P_i in Φ_{min}

if $x_{i_{max}} - x_{if} \geq 1$ **then** add P_i to Φ_T

construct Φ_{Sorted} by sorting Φ_{min} on the number of VPs assigned to P_i

for all processors P_i in Φ_{Sorted}

if $x_{if} \leq free$ **then**

$free = free - x_{if}$

for all processors P_j in Φ_T

reallocate VPs from P_i to P_j

if P_i has no more VPs **then** remove P_i from Φ_{min}

until P_i has no more VPs or P_j has no more capacity

reinsert P_j in Φ_{Sorted}

We now show that the Compression algorithm yields the set Φ_{min} with the smallest number of processors.

Lemma 2: The Compression algorithm is necessary for yielding the set Φ_{min} with the smallest number of processors that can achieve T_{min}.

Proof: Example 2, illustrates a counter example in which the MTAT algorithm produces a solution with more processors than necessary.

Lemma 3: The Compression algorithm is sufficient for yielding the set Φ_{min} with the smallest number of processors that can achieve T_{min}.

Proof: The Compression algorithm assigns an upper bound $x_{i_{max}}$ on the allocation of VPs to processors such that all processors complete in less than or equal to T_{min} time units. This step guarantees that the job's completion time following the Compression algorithm will not exceed that of the MTAT algorithm. The Compression algorithm then identifies all processors that can accommodate additional whole VPs without exceeding $x_{i_{max}}$ (i.e., those for which $x_{i_{max}} - x_{if} \geq 1$) and inserts them in the set Φ_T. All processors are then added to the list Φ_{sorted} which is sorted according to the number of VPs allocated to that processor. Since the processors with the fewest VPs are visited first, and if it is not possible to redistribute all the VPs x_{if} of the current processor to the set of processors Φ_T without exceeding $x_{i_{max}}$, then it is not possible to redistribute all the VPs of any subsequent processor to Φ_T without exceeding $x_{i_{max}}$. The algorithm terminates at this point and hence the minimum number of processors needed to achieve T_{min} has been selected.❑

Theorem 3: The Compression algorithm yields the set Φ_{min} with the smallest number of processors that can achieve T_{min}.

Proof: Lemmas 2 and 3 show that the Compression algorithm is necessary and sufficient.❑

Example 3: Consider a job with $X = 9$ and $a = \{4, 2, 1\}$. The minimum ideal turn around time α is (9/7) = 1.2857. The MTAT algorithm yields $\Phi = \{\text{all}\}$, and $T_{min} = 3/2$. Figure 4-a illustrates the effects of the MTAT algorithm. Figure 4-b illustrates the effects of the Compression algorithm.:

a_i's ⟶	4	2	1
$x_i = \alpha a_i$	5.1428	2.5714	1.2857
$\tilde{x}_i$	5	2	1
Drag_i	0.21428	0.21428	0.7143
$\tilde{x}_{if}$	5	2+1= 3	1
$\tilde{\alpha}_i$	5/4	3/2	1

$$Diff = 1 \qquad T_{min_{MTAT}} = 3/2$$

Figure 4-a. The MTAT algorithm applied to Example 3.

The description of the Compression algorithm assumes that all processors are architecturally equivalent. The algorithm can be extended to deal with disjoint pools of processors with equiva-

a_i's ⟶	*4* (i=1)	*2* (i=2)	*1* (i=3)
$x_{i_{max}}$	6	3	1.5
$\lfloor x_{i_{max}} \rfloor$	6	3	1
$x_{3_{max}} = free$ = 1, thus			
x_{if}	6	3	

Figure 4-b. The Compression algorithm applied to Example 3.

lent architecture by calling it once for each architecture pool and returning the union of the minimum processor sets. The algorithm is as follows: let Z be the set of distinct architecture types represented in the set of available processors Φ, let Φ_z be the set of available processors of architecture type *z*, let X_z be the number of VPs of a job that are restricted to architecture type *z* and let Φ_{min} be the smallest set of processors on which the job can achieve T_{min} in the heterogeneous environment. Then $\Phi_{min} = \cup (\Phi_{zmin}, \forall z \in Z)$, where Φ_{zmin} = Compress (X_z, Φ_z, T_z) and T_z = MTAT (X_z, Φ_z).

Theorem 4: The heterogeneous Compression algorithm returns the minimum set of processors needed to achieve T_{min} in the heterogeneous environment.

Proof: The minimum set of processors needed to achieve T_{min} in the heterogeneous environment must be no greater than the set of processors needed to achieve T_{min} on each architecture pool. The Compression algorithm returns the minimum set of processors needed to achieve T_z on each architecture pool *z*, where $T_z \leq T_{min}$. Therefore, the processor set Φ_{zmin} is no larger than necessary for any architecture. Since the heterogeneous Compression algorithm returns the union of Φ_{zmin} for only the required architecture types, it returns the minimum set of processors needed to achieve T_{min}. ❑

The worst case complexity of the Compression algorithm is $\Theta (Zn_z \log n_z)$ where n_z is the number of processors of a given architecture available to the scheduler.

4.3. The Expansion Algorithm

The goal of the Expansion algorithm is to incorporate the job into an existing schedule contain-

ing other jobs. Using the model described earlier, it can achieve this goal either by allocating empty entries in the allocation map or by extending the allocation map with an additional time slice. The criterion for deciding when to allocate a new time slice is as follows. The MTAT algorithm indicates the minimum turn around time T_{min} for the job using all available processors. If the job were scheduled in a new time slice its turn around time would be T_{min}/τ where τ is the number of time slices. A new time slice will only be allocated if this turn around time can not be equalled or beaten using existing free space in the allocation map.

The purpose of the Expansion algorithm is to search for patterns of usable free space in the allocation map that satisfy the gang scheduling constraints discussed earlier. There are many different algorithms that could be used to discover such space, each of which makes a different trade-off of complexity for accuracy. An accurate algorithm would guarantee to find all possible combinations of usable space in the allocation map at the expense of high complexity. Other algorithms reduce complexity and hence run faster at the cost of missing some potentially good solutions. The overall design of our scheduling algorithm is such that the Expansion algorithm can be easily replaced to match the needs of a particular environment[5]. The Expansion algorithm outlined below is a compromise between complexity and accuracy and searches for *regular patterns* of free space.

Definition: A pattern is a collection of empty slots in the allocation map. A pattern is a *regular pattern* if the following condition is met: let the set Φ_p denote the processors where the pattern of empty slots reside. Also, let this pattern span the set of time slices τ. Then, the pattern of vacant slots is a regular pattern if all the slots resulting from the cross product $\Phi_p \times \tau$ are in the pattern.

Any pattern can be decomposed into regular patterns. For example, see Figure 5, any collection of vacant slots forming a row or a column is a regular pattern. Some of the regular patterns that can be derived from the pattern shown in Figure 5 are: {S0, S1, S2, S3, S4, S5}, {S1, S2, S4, S5, S9, S10}, {S1, S2, S4, S5, S9, S10, S15, S16},....etc.

The Expansion algorithm only considers regular patterns of free space. Specifically, it searches the allocation map for the largest regular pattern, uses the MTAT algorithm to compute the turn around time for the job using the processors in that regular pattern, and then compares the result

5. For example, a small system may favor accuracy over complexity by using an algorithm that finds all possible combinations of free space.

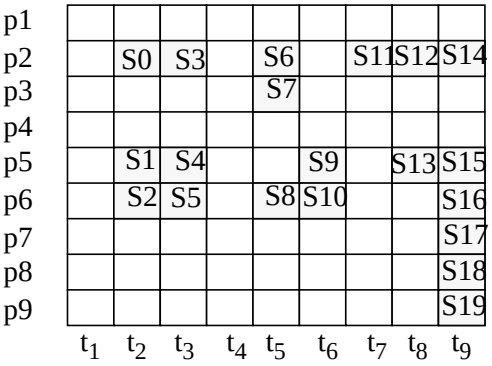

Figure 5. Regular patterns of empty space in the allocation map

with the original result from the MTAT algorithm that specifies the turn around time for the job using a new time slice. If the regular pattern results in a faster turn around time then the processors that it contains are passed as input to the Compression algorithm and the resulting minimal processor set is assigned to the job during the time slices contained in the pattern. Otherwise, a new time slice is added and the Compression algorithm is used on the full processor set to determine the final assignment.

The details of the Expansion algorithm are as follows.

Let E be the set of all columns (time slices) in the allocation map that contain empty entries and let E_i be the set of empty entries in column i.

For (all columns $Col_i \in E$)

$width_i = 1$

For (all columns $Col_j \in E$ where $j \neq i$)

if ($E_i \subseteq E_j$) **then**

$width_i = width_i + 1$

Size ($Pattern_i$) = $\left(\sum_{\forall k \in E_i} a_k\right) width_i$

Return max (size ($Pattern_l$), $\forall Col_l \in E$).

The worst case complexity of the Expansion algorithm is $\Theta(\Phi\bar{\tau}^2)$ where $\bar{\tau}$ represents the number of time slices and Φ is the number of processors. Note that the algorithm presented above does not necessarily find all regular patterns. We chose to avoid a more exhaustive approach due

to its complexity. Instead, our algorithm searches for column-oriented solutions, i.e., those that contain the maximum available parallelism in at least one of the time slices. The penalty for taking this approach is that we run the risk of missing some usable patterns that contain only subsets of the empty entries from all columns.

Since we do not yet have experience with running the Expansion algorithm in real-world systems, and since insufficient trace data on VP and processor behavior exists, we designed the scheduling algorithm with the Expansion algorithm as a repluggable component. This approach should facilitate future real-world and simulation-based comparisons of different algorithms.

4.4. Handling Scheduling Events

The three algorithms outlined above constitute the heart of the scheduling algorithm. This section illustrates their use in handling initial job submissions and each of the dynamic scheduling events (new_processor, new_VP, processor_exit and VP_exit).

4.4.1. Job submission

The submission of a new job invokes the MTAT, Expansion, and Compression algorithms as illustrated in Figure 2. First, the MTAT algorithm is used to estimate the job's turn around time T_{min} that would result from the use of a new time slice. The Expansion algorithm is then used to find the largest regular pattern of free entries in the allocation map, and the MTAT algorithm is used again to estimate the job's turn around time T_{free} using that pattern. If T_{free} is smaller than or equal to T_{min} it is not necessary to allocate a new time slice since the job will run at least as fast using existing free capacity. In either case, the job's minimum turn around time may be attainable using a subset of the processors originally considered. Therefore, it is necessary to run the Compression algorithm to determine the final allocation. Finally, the allocation map is updated to include the new job.

4.4.2. Handling new_processor and VP_exit events

The algorithms for handling new_processor and VP_exit events are very closely related because both events have a similar effect on the allocation map: they both lead to an increase in the number of free entries. The scheduler's response to both types of event is to attempt to use the free entries to speed up existing jobs. It runs the Expansion algorithm to determine the largest regular pattern of free entries. Then it selects a job and uses the MTAT algorithm to determine whether

the job can utilize the new entries. If so, the Compression algorithm is used to attempt to compress the job's processor allocation, the job is reallocated and the allocation map is updated to reflect the newly allocated and freed entries. The scheduler repeats the procedure for other jobs until either no free entries remain or all jobs have been visited. In order to maintain fairness among jobs, the scheduler manages its jobs in a queue and starts with a new job for each event. Figure 6 summarizes the scheduling algorithm.

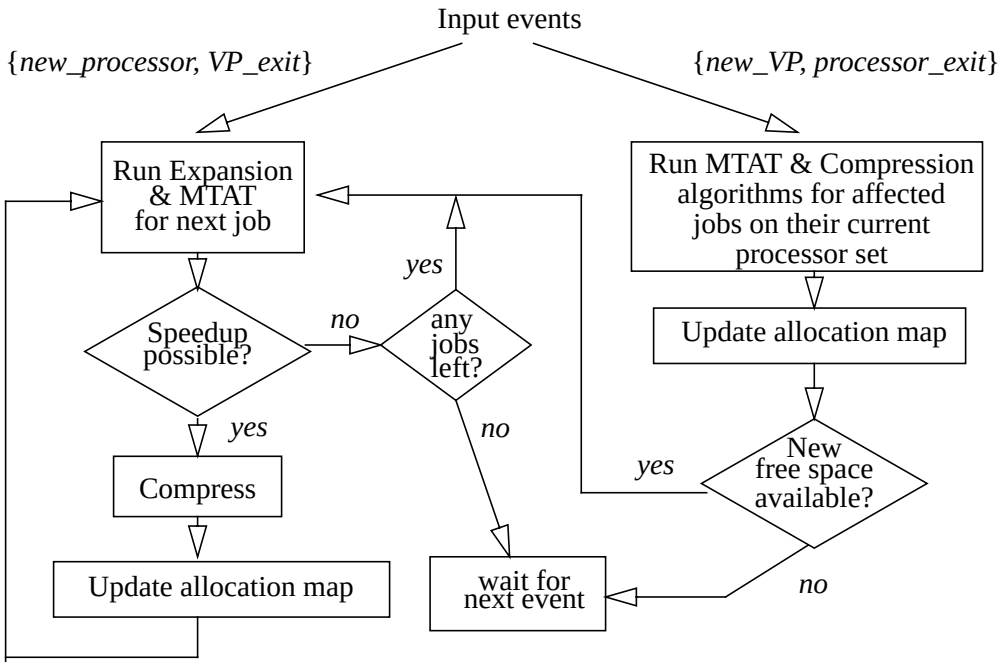

Figure 6. Summary of the scheduling algorithm

From the scheduler's point of view, the only distinction between the VP_exit and new_processor events is that VP_exit may cause a complete column of the allocation map to become empty, in which case the scheduler will remove it.

4.4.3. Handling new_VP and processor_exit events

The relationship between the new_VP and processor_exit events is similar to that between the new_processor and VP_exit events discussed in the previous section. Both events have the effect of consuming space in the allocation map. The scheduler's response to both events is to (a) call the MTAT algorithm to recompute the minimum turn around time for the affected jobs on the available processors and (b) call the Compression algorithm to recalculate the minimum proces-

sor allocation. If after running the Compression algorithm the number of free entries in the allocation map increases, the scheduler attempts to expand other jobs using the approach described in the previous section.

The main distinction between the processor_exit and new_VP events, from the scheduler's point of view, is that the processor_exit event causes a complete row to be removed from the allocation map.

5. CONCLUSION

We have presented an algorithm for dynamically scheduling parallel jobs in heterogeneous distributed systems. The algorithm, which is based on gang scheduling, supports environments in which processors can have unequal processing capacities and incompatible architecture types, and is dynamic in the sense that it handles the creation and deletion of both processors and VPs during the execution of a job. These characteristics make the algorithm applicable to systems ranging from massively parallel processors to multi-user networks of heterogeneous workstations.

The algorithm is modular in design, allowing the use of a variety of expansion policies. This approach allows the behavior of the scheduler to be tailored for different environments. For example, a scheduler for a small system might be required to reorganize its assignment of VPs to processors on every scheduling event in order to maintain the optimal assignment at all times. This approach quickly becomes infeasible in larger systems because (a) migration overhead increases with the frequency and scope of reorganization, which increase with system size, and (b) the complexity of calculating the optimal assignment increases rapidly with increases in system size. Consequently, schedulers with heuristic expansion policies are more suitable for larger systems, whereas schedulers with optimal expansion policies are suitable for small systems. The algorithm proposed here provides the flexibility to support a wide range of different systems by implementing its expansion policy as a replacable module.

A number of issues remain to be solved. The first, and most important, task for future research is to distribute the allocation map such that scheduling decisions can be made asynchronously at different sites. This extension will greatly improve the scalability of the algorithm. Second, we would like to explore the behavior of different allocation policies, particularly those that utilize information about past behavior for processors and VPs. We believe that such information would

be relatively easy to gather in real-world environments and would significantly improve the scheduler's allocation decisions. Finally, we are implementing a real-world scheduler based on this algorithm. We plan to release this scheduler as part the migratable PVM (MPVM) system which is currently under development at OGI [10].

6. REFERENCES

[1] T. L. Casavant and J. G. Kuhl, "A Taxonomy of Scheduling in General-purpose Distributed Computing systems," *IEEE Trans. Software Engineering*, Vol. 14, no. 2, pp. 141-154, Feb. 1988.

[2] M. W. Mutka, "Estimating Capacity for Sharing in a Privately Owned Workstation Environment," *IEEE Trans. Software Engineering*, Vol. 18, no. 4, pp. 319-328, April 1992.

[3] V. S. Sunderam, "PVM: A Framework for Parallel Distributed Computing," *Concurrency: Practice & Experience*, Vol. 2(4), pp. 315-339, December 1990.

[4] G. A. Geist, and V. S. Sunderam, "Network Based Concurrent Computing on the PVM system," *Concurrency: Practice & Experience*, Vol. 4(4), pp. 293-311, June 1992.

[5] C. C. Douglas, T. G. Mattson, and M. H. Schultz, "Parallel Programming Systems for Workstation Clusters," Tech. Rep. 975, Dep. Comput. Sci., Yale University, August 1993.

[6] R. Konuru, J. Casas, R. Prouty, S. Otto, and J. Walpole, "A User Level Process Package for PVM," *Proceedings of the Scalable High Performance Computing Conference*, pp 48-55, May 1994.

[7] M. K. Litzkow, M. Livney, M. W. Mutka, "Condor- A Hunter of Idle Workstations," *Proceedings the Eighth International Conference on Distributed Computing Systems*, pp. 104-111, San Jose, CA, June 1988.

[8] F. Douglas, and J. Ousterhout, "Process Migration in the Sprite Operating System," *Proceedings the Seventh International Conference on Distributed Computing Systems*, pp. 18-25, Berlin, West Germany, September 1987.

[9] J. M. Smith, "A survey of Process Migration Mechanisms," *ACM Oper. Syst. Review*, Vol. 22, No. 3, pp. 28-40, July 1988.

[10] J. Casas, R. Konuru, S. Otto, R. Prouty, J. Walpole, "Adaptive Load Distribution Systems for PVM," to appear as a contributed paper in *Supercomputing '94 Proceedings*.

[11] C. Maccann, R. Vaswani, and J. Zahorjan, "A Dynamic Processor Allocation Policy for Multiprogramming Shared Memory Multiprocessors," *ACM Transactions on Computer Systems*, Vol. 11, No. 2, pp.146-178, May 1993.

[12] A. Tucker, and A. Gupta, "Process Control and Scheduling Issues in Multiprogrammed Shared Memory Multiprocessors," *Proceedings of the 12th Symposium on Operating System Principles*, pp. 159- 166, Dec. 1989.

[13] D. L. Eager, E. D. Lazowaska, and J. Zahorajan, “Adaptive Load Sharing in Homogenous Distributed Systems,” *IEEE Trans. Software Engineering*, Vol. 12, no. 5, pp. 662-675, May 1986.

[14] Y. Belhamissi, and M. Jegado, “Scheduling in Distributed Systems: Survey and Questions,” Tech. Report 1478, IRISA/INRIA, Rennes cedex, France, June 1991.

[15] O. Kremien, and J. Kramer, “Methodical Analysis of Adaptive Load Sharing Algorithms,” *IEEE Transaction. on Parallel and Distributed Systems*, Vol. 3, No. 6, pp. 747-760, November 1992.

[16] S. Zhou, “A Trace Driven Simulation Study of Dynamic Load Balancing,” *IEEE Trans. Software Engineering*, Vol. 14, No. 9, pp. 1327-1341, September 1988.

[17] J. Xu, and K. Hwang, “Dynamic Load Balancing for Parallel Program Execution on a message Passing Multicomputer,” *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pp. 402-406, 1990.

[18] O. Kremien, J. Kramer, and J. Magee, “Scalable, Adaptive Load Sharing for Distributed Systems,” *IEEE Parallel and Distributed Technology*, Vol. 1, No. 3, pp. 62-70, August 1993.

[19] I. Ahmad, and A. Ghafoor, “A Semi Distributed Load Balancing Scheme for Massively Parallel Multicomputer Systems,” *IEEE Trans. Software Engineering*, Vol. 7, No. 10, pp. 987-1004, October 1991.

[20] T. T. Suen, and J. S. Wong, “Efficient Task Migration Algorithm for Distributed Systems,” *IEEE Tran. on Parallel and Distributed Systems*, Vol. 3, No. 4, pp. 488-499, July 1992.

[21] K. Baumgartner, and B. Wah, “Gammon: A load Balancing Strategy for Local Computer System with Multiaccess Networks,” *IEEE Trans. Computers*, Vol. 38, No. 8, pp.1098-1109, Aug. 1989.

[22] M. H. Willebeek-LeMair, and A. P. Reeves, “Strategies for Dynamic Load Balancing on Highly Parallel Computers,” *IEEE Tran. on Parallel and Distributed System*s, Vol. 4, No. 9, pp. 979-993, September 1993.

[23] S. Zhou, J. Wang, X. Zheng, and P. Delisle, “Utopia: A load Sharing Facility for Large, Heterogeneous Distributed Computer Systems,” Tech. Report 257, Computer Systems Research Institute, University of Toronto, Canada, April 1992.

[24] T. Green, and J. Snyder, “DQS, A distributed Queuing System,” Florida State University, March 1993.

[25] B. C. Neumann, and S. Rao, “Resource Management for Distributed Parallel Systems,” *Proceedings of the Second International Symposium on High Performance Distributed Computing*, Spokane, July 1993.

[26] D. L. Black, “Scheduling Support for Concurrency and Parallelism in the Mach Operating System,” *IEEE Computer*, Vol. 23, No. 5, pp. 35-43, May1990.