7-1997

Dynamic Load Distribution in MIST

K. Al-Saqabi

R. M. Prouty

Dylan McNamee

Steve Otto

Jonathan Walpole
Portland State University

Citation Details

"Dynamic Load Distribution in MIST," K. Al-Saqabi, R. Prouty, D. McNamee, S. Otto and J. Walpole, In Proceedings of the 1997 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 97), Las Vegas, Nevada USA, June 30 - July 3, 1997.

Dynamic Load Distribution in MIST

K. Al-Saqabi
Dept. of Electrical and Computer Engineering
College of Engineering and Petroleum
Kuwait University
Safat, Kuwait

R. M. Prouty, D. McNamee,
S. W. Otto, J. Walpole
Dept. of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology
Portland, Oregon U.S.A.

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 turnaround 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.*

Keywords: Scheduling, distributed systems, heterogeneity, process migration.

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]. The Parallel Virtual Machine (PVM) [2], P4, Linda and others [3] are examples of software systems that support such functionality.

While the problem of scheduling parallel applications on distributed computing systems is already well-explored [4], 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 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 [1]. 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 [5]. Consequently, scheduling decisions must be made quickly and it must be possible to migrate VPs dynamically.[1] 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 turnaround 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 main motivation for this work is the development of a real-world global scheduler for use in our parallel programming environment, MIST [9]. MIST combines Migratable PVM (MPVM) [10] with global scheduling and load monitoring. MIST is designed to use idle cycles scavenged from shared networks of workstations to run existing PVM programs efficiently and effectively. MPVM allows VPs to be asynchronously migrated between homogeneous processors. Migration is transparent to the application. MPVM

1. We assume the existence of a migration mechanism that is capable of migrating VPs among equivalent processor architectures at any stage during their execution [6,7,8]. We also assume that such a mechanism is heavy-weight and should not be invoked frequently.

also allows entire parallel jobs to be suspended and resumed while maintaining the ability to migrate VPs. VP migration and job suspension/resumption are all required for dynamic gang scheduling.

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 approaches [11,12] or distributed systems approaches [4,13,14]. 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 prior 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. Though this algorithm was developed primarily for use in a distributed scheduler for MIST, 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.

A number of techniques have been developed to reduce the overhead of Zhou's algorithm. Xu presents four heuristics [17] that reduce overhead 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., Ahmad, and Suen [18,19,20] propose various techniques to reduce overhead by subdividing the system and using combinations of local and global information to schedule tasks locally or remotely. Willebeek-LeMair and Reeve [21] presented four scheduling policies that dynamically balance load without using global information, instead considering load only on neighboring processors. Other researchers [22] 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.

More centralized, static approaches to scheduling in heterogeneous environments are supported in the Load Sharing Facility (LSF), Utopia [23], Distributed Queuing System (DQS) [24], Portable Batch System (PBS) [25] and Prospero Resource Manager (PRM) [26]. 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 concept of dynamic migration. Consequently, they are more appropriate for dedicated, or otherwise idle, environments than for continual use in general purpose multiuser networks.

The Adaptive Load Distribution SYstem (ALDY) is a library for dynamic load balancing of distributed objects [27]. ALDY manages the load balancing of application defined VPs and migrates VPs using an application defined mechanism. ALDY uses a repluggable strategy to make redistribution decisions. Parameters for the strategy are input from a start-up file.

Condor [5] is the only system we are aware of that supports dynamic, heterogeneous scheduling. Condor originated as a system for running sequential jobs using cycles scavenged from distributed systems. It has evolved to handle parallel jobs as well. Condor can checkpoint a single process in either of two ways: (a) to a file that is then saved to stable storage, which is useful for fault tolerance, or (b) to a socket and on into a waiting process, which is useful for process migra-

tion. A Condor job's environment is preserved across migrations through a facility called *remote system calls*.

Condor interacts with parallel jobs via three additional facilities: CARMI and WoDi [28], and CoCheck [29]. CARMI is an interface between Condor and parallel jobs that use message passing environments such as PVM. CARMI allows a parallel job to request resources from Condor, then create VPs on those resources. Jobs using CARMI have access to all resource information possessed by Condor, so resource requests can be as general or specific as necessary. WoDi, which stands for *work distributor*, is a framework for writing dynamic parallel programs using the *master-workers* model. WoDi uses CARMI to access Condor. CoCheck provides consistent checkpoints of PVM jobs. Condor uses CoCheck to checkpoint entire parallel jobs or migrate individual VPs.

To our knowledge, Condor does not schedule parallel programs in any special way. If gang scheduling of parallel jobs is desired, Condor could potentially use the algorithm described in this paper. CARMI allows the parallel jobs to take advantage of dynamic resources, and Condor can use CoCheck to dynamically migrate the VPs between processors.

Static gang scheduling of parallel programs has been shown to be viable on both dedicated [30] and non-dedicated [31], high-performance workstation clusters. In this paper, we present an algorithm for dynamically scheduling parallel applications in non-dedicated, 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 events: *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 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.

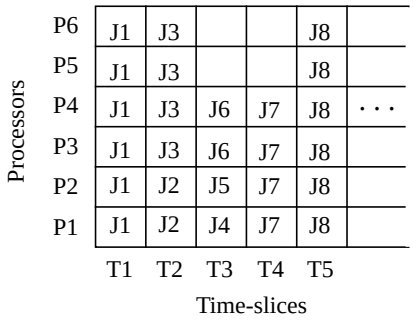

Figure 1: Allocation map over time for gang scheduling multiple jobs on a set of processors.

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. We assume that the VPs of a single job communicate frequently and hence will benefit from gang scheduling [32]. 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 any 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 turnaround 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 turnaround time for job J1[2]. Taking job J1's VPs from processors P5 and P6 and placing them on P3 and P4 will not further increase the turnaround 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 to new processors 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 prior 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 greedy approach: If a job can be speeded up through expansion, we assume that it will run for long enough to offset the migration cost. If a job can not be speeded up, we do not change its current allocation.

Compression and expansion are relatively straightforward 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 turnaround time for the job, T_{min}, assuming that it is maximally distributed across a given set of processors, Φ. The Compression algorithm attempts to achieve T_{min} using a smaller set of processors Φ_{min}. The result of the Compression algorithm is an allocation of VPs to processors that yields the minimum turnaround time using the smallest number of processors. The Expansion algorithm is used to explore the use of existing free space in the allocation map. The result of the Expansion algorithm is the most suitable processor set and time slice(s) available from free space.

The relationship between these algorithms is illustrated in Figure 2. The advantage of the approach shown in Figure 2 is that it is both efficient and fair. It prefers solutions that minimize the turnaround time of the submitted job without impacting existing jobs, and falls back on solutions that continue to minimize the turnaround time for the submitted job while impacting all jobs equally.

4.1 The MTAT Algorithm

The Minimum Turn-Around Time (MTAT) algorithm has two stages. The first stage determines the ideal load distribution across the set of available processors.

2. This example assumes an initial assignment of one VP per processor for job J1.

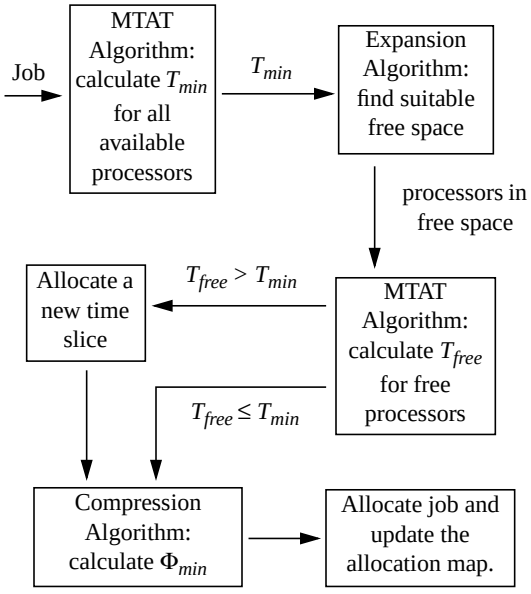

Figure 2: Relationships among the basic scheduling algorithms.

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[3]. 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

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

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:

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

where α is the job's minimum possible turnaround time. Note that this turnaround 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.

The result of making x_i an integer will be an increase in the turnaround 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 turnaround 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 an upper bound, $\tilde{x}$, on the number of complete VPs that can be allocated to it in an optimal solution:
$$\tilde{x}_i = \lfloor x_i \rfloor, \forall\ i \in \Phi.$$
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 turnaround time of the job, called *Drag*, that would result from allocating an additional VP to that processor:
$$Drag_i = \frac{(\lceil x_i \rceil - x_i)}{a_i}\ , \forall\ i \in \Phi.$$
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 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
$$T_{if} = \alpha + \frac{(\tilde{x}_{if} - x_i)}{a_i} \quad \text{and} \quad T_{min} = \underset{\forall i \in \Phi}{Max}\ \{T_{if}\}$$

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

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

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

$$\{\tilde{x}_i\} = \{11, 1, 4, 3\} \text{ and } Diff = X - \sum \tilde{x}_i = 1$$

Step 4 of the MTAT algorithm yields $\{Drag_i\}$ =

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

{0.089, 0.889, 0.139, 0.222}. Thus the extra VP is allocated to processor 1 and $\{x_{if}\} = \{12, 1, 4, 3\}$. Consequently, the completion times on each processor $\{T_{if}\}$ are {1.2, 1, 1, 1} and the overall turnaround time T_{min} for the job is 1.2. ❑

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 turnaround 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 turnaround time of a job on Φ_z. Then $T_{min} = \max(T_z, \forall z \in Z)$ where $T_z = \text{MTAT}(X_z, \Phi_z)$.

The complexity of the MTAT algorithm is $O(Z\ n_z \log n_z)$, 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 turnaround 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}.

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 scheduled on 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 turnaround time, it does not necessarily yield the minimum possible Φ. ❑

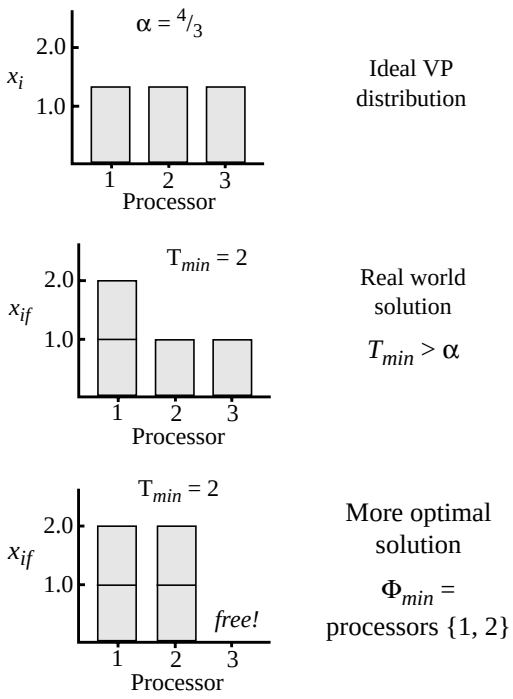

Figure 3: Optimizing the solution of Example 2

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 $\Phi_{min} = \Phi - \{i | x_{i_{max}} < 1\}$.
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} in increasing order on the number of VPs assigned to P_i.
  - for all processors P_i in Φ_{sorted}
    - if $x_{if} \leq free$ then
      - Remove P_i from Φ_{min}.
      - $free = free - x_{if}$
      - Re-allocate VPs from P_i to members of Φ_T, removing processors from Φ_T as their free capacity is exhausted.

Example 3: Consider a job with $X = 9$ and $a = \{4, 2, 1\}$. The minimum ideal turnaround time α is (9/7) = 1.29. The MTAT algorithm yields $\Phi = \{all\}$, and $T_{min} = 3/2$. Table 1 illustrates the effects of the MTAT algorithm. Table 2 illustrates the effects of the Compression algorithm. ❑

Table 1: The MTAT algorithm applied to Example 3. *Diff = 1* $T_{min} = 3/2$

a_i	$x_i = \alpha\, a_i$	$\tilde{x}_i$	$Drag_i$	x_{if}	T_{if}
4	*5.14*	*5*	*0.214*	*5*	*5/4*
2	*2.57*	*2*	*0.214*	*2+1=3*	*3/2*
1	*1.29*	*1*	*0.714*	*1*	*1*

Table 2: The Compression algorithm applied to Example 3.

a_i	x_{if}	$x_{i_{max}}$	$\lfloor x_{i_{max}} \rfloor$	*New* x_{if}
4	*5*	*6*	*6*	*5 + 1 = 6*
2	*2*	*3*	*3*	*3*
1	*1*	*1.5*	*1*	*0*

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 equivalent 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).

The complexity of the Compression algorithm is $O(Z\, n_z \log n_z)$ 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 containing 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 turnaround time T_{min} for the job using all available processors. If the job were scheduled in a new time slice its turnaround time would be $T_{min}\tau$ where τ is the number of time slices. A new time slice will only be allocated if this turnaround 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[4]. 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_p. Then, the pattern of vacant slots is a regular pattern if all the slots resulting from the cross product $\Phi_p \times \tau_p$ are in the pattern.

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

Any pattern can be decomposed into regular patterns. For example, in Figure 4, 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 4 are {S0, S1, S2, S3, S4, S5, S14, S15, S16}, {S0, S2, S3, S5, S6, S8, S14, S16}, and {S1, S2, S4, S5, S9, S10, S15, S16}.

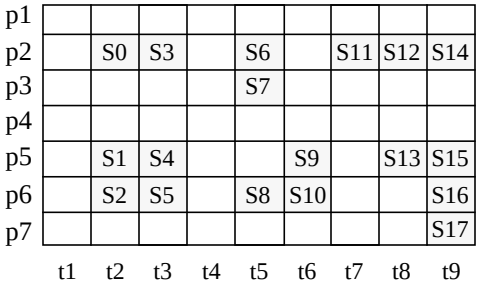

Figure 4: Patterns of empty space in the allocation map

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 turnaround time for the job using the processors in that regular pattern, and then compares the result with the original result from the MTAT algorithm that specifies the turnaround time for the job using a new time slice. If the regular pattern results in a faster turnaround 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 complexity of the Expansion algorithm is $O(\Phi\tau^2)$ where τ 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 such as the packing schemes discussed by Feitelson [33].

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 turnaround 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 turnaround time T_{free} using that pattern. Note that T_{min} must reflect that the job will be given a single, new time slice, and T_{free} must reflect use of however many time slices are part of the free space returned from the Expansion algorithm. 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 turnaround 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 5 summarizes the scheduling algorithm.

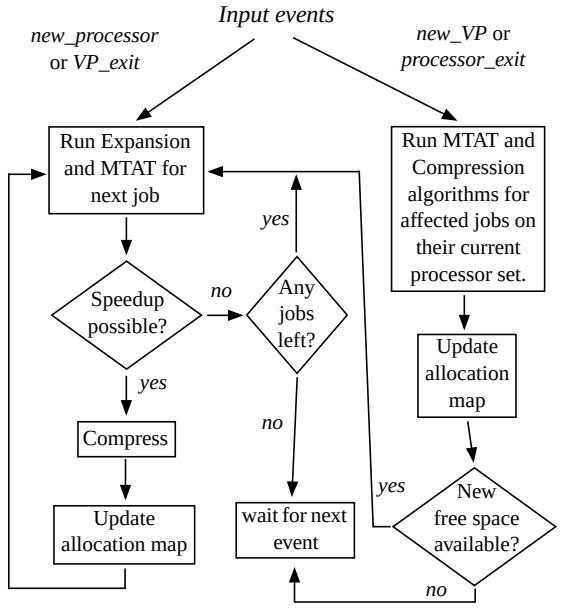

Figure 5: Summary of the scheduling algorithm

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 turnaround time for the affected jobs on the available processors and (b) call the Compression algorithm to recalculate the minimum processor 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 multiuser networks of heterogeneous workstations.

The algorithm's modular design can accommodate 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 could reorganize its assignment of VPs to processors on every scheduling event. This approach maintains the optimal assignment at all times, but 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 system growth. The algorithm proposed here provides the flexibility to support a wide range of different systems by implementing its expansion policy as a replaceable 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 MIST system which is currently under development at OGI [9].

6. References

[1] M. W. Mutka. Estimating capacity for sharing in a privately owned workstation environment. *IEEE Trans. Software Engineering* 18(4):319–328, Apr. 1992.

[2] G. A. Geist and V. S. Sunderam. Network-based concurrent computing on the PVM system. *Concurrency: Practice & Experience* 4(4):293–311, Jun. 1992.

[3] C. C. Douglas, T. G. Mattson and M. H. Schultz. Parallel programming systems for workstation clusters. Tech. Rep. 975, Dept. Comp. Sci., Yale University, Aug. 1993.

[4] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Software Engineering* 14(2):141–154, Feb. 1988.

[5] M. K. Litzkow, M. Livney and M. W. Mutka. Condor: A hunter of idle workstations. *Proceedings the Eighth International Conference on Distributed Computing Systems* pp. 104–111, San Jose, CA, Jun. 1988.

[6] 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, Sep. 1987.

[7] J. M. Smith. A survey of process migration mechanisms. *ACM Operating Systems Review* 22(3):28–40, Jul. 1988.

[8] J. Casas, R. Konuru, S. Otto, R. Prouty and J. Walpole. Adaptive load distribution systems for PVM. *Proceedings of Supercomputing '94* pp. 390–399, Washington, D.C., Nov. 14–18, 1994.

[9] J. Casas, D. Clark, P Galbiati, R. Konuru, S. Otto, R. Prouty and J. Walpole. MIST: PVM with transparent migration and checkpointing. presented at the 3rd annual PVM Users' Group Meeting, Pittsburg, PA, May 7–9, 1995.

[10] J. Casas, D. Clark, R. Konuru, S. Otto, R. Prouty and J. Walpole. MPVM: A migration transparent version of PVM. *Computing Systems* 8(2): 171–216, Spring 1995.

[11] C. Maccann, R. Vaswani and J. Zahorjan. A dynamic processor allocation policy for multiprogramming shared memory multiprocessors. *ACM Transactions on Computer Systems* 11(2):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* 12(5):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, Jun. 1991.

[15] O. Kremien and J. Kramer. Methodical analysis of adaptive load sharing algorithms. *IEEE Transaction. on Parallel and Distributed Systems* 3(6):747–760, Nov. 1992.

[16] S. Zhou. A trace driven simulation study of dynamic load balancing. *IEEE Trans. Software Engineering* 14(9):1327–1341, Sep. 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* 1(3):62–70, Aug. 1993.

[19] I. Ahmad and A. Ghafoor. A semi-distributed load balancing scheme for massively parallel multicomputer systems. *IEEE Trans. Software Engineering* 7(10):987–1004, Oct. 1991.

[20] T. T. Suen and J. S. Wong. Efficient task migration algorithm for distributed systems. *IEEE Tran. on Parallel and Distributed Systems* 3(4):488–499, Jul. 1992.

[21] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Tran. on Parallel and Distributed Systems* 4(9):979–993, Sep. 1993.

[22] K. Baumgartner and B. Wah. Gammon: A load balancing strategy for local computer system with multiaccess networks. *IEEE Trans. Computers* 38(8):1098–1109, Aug. 1989.

[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, Apr. 1992.

[24] T. Green and J. Snyder. DQS, A distributed queuing system. Florida State University, Mar. 1993.

[25] R. L. Henderson. Job scheduling under the Portable Batch System. In *Job Scheduling Strategies for Parallel Processing*, pp. 279–294. Lecture notes in computer science, Vol. 949, D. G. Feitelson and L. Rudolph, editors. Springer-Verlag, 1995.

[26] B. C. Neumann and S. Rao. Resource management for distributed parallel systems. *Proceedings of the Second International Symposium on High Performance Distributed Computing* Spokane, Jul. 1993.

[27] C. Gold and T. Schnekenburger. Using the ALDY load distribution system for PVM applications. In *Parallel Virtual Machine—EuroPVM '96*, pp 278–287. Lecture notes in computer science, Vol. 1156, A. Bode, et al., editors, Springer-Verlag, 1996.

[28] J. Pruyne and M. Livny. Parallel processing on dynamic resources with CARMI. In *Job Scheduling Strategies for Parallel Processing*, pp. 259–278. Lecture notes in computer science, Vol. 949, D. G. Feitelson and L. Rudolph, editors. Springer-Verlag, 1995.

[29] J. Menden and G. Stellner. Proving Properties of PVM applications—a case study with CoCheck. In *Parallel Virtual Machine—EuroPVM '96*, pp 134–141. Lecture notes in computer science, Vol. 1156, A. Bode, et al., editors, Springer-Verlag, 1996.

[30] A. Hori, H. Tezuka, Y. Ishikawa, N. Soda, H. Konaka and M. Maeda. Implementation of gang-scheduling on workstation cluster. In *Job Scheduling Strategies for Parallel Processing*, pp. 126–139. Lecture notes in computer science, Vol. 1162, D. G. Feitelson and L. Rudolph, editors. Springer-Verlag, 1996.

[31] F. Wang, H. Franke, M. Papaefthymiou, P. Pattnaik, L. Rudolph and M. S. Squillante. A gang scheduling design for multiprogrammed parallel computing environments. In *Job Scheduling Strategies for Parallel Processing*, pp. 111–125. Lecture notes in computer science, Vol. 1162, D. G. Feitelson and L. Rudolph, editors. Springer-Verlag, 1996.

[32] D. L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer* 23(5):35–43, May1990.

[33] D. G. Feitelson. Packing schemes for gang scheduling. In *Job Scheduling Strategies for Parallel Processing*, pp. 89–110. Lecture notes in computer science, Vol. 1162, D. G. Feitelson and L. Rudolph, editors. Springer-Verlag, 1996.