LOAD BALANCING FOR PARALLEL LOOPS IN WORKSTATION CLUSTERS

Tae-Hyung Kim and James M. Purtilo

Institute for Advanced Computer Studies
and Computer Science Department
University of Maryland
College Park, Maryland 20742
{thkim,purtilo}@cs.umd.edu

Abstract

Load imbalance is a serious impediment to achieving good performance in parallel processing. Global load balancing schemes are not adequately manage to balance parallel tasks generated from a single application. Dynamic loop scheduling methods are known to be useful in balancing parallel loops on shared-memory multiprocessor machines. However, their centralized nature causes a bottleneck for the relatively small number of processors in workstation clusters because of order-of-magnitude differences in communications overheads. Moreover, improvements of basic loop scheduling methods have not dealt effectively with irregularly distributed workloads in parallel loops, which commonly occur in applications for workstation clusters. In this paper, we present a new decentralized balancing method for parallel loops on workstation clusters.

Key words: Load balancing, Parallel loop scheduling, Workstation clusters, Distributed programs, Programming environments.

This research has been supported by Office of Naval Research and a grant from the Siemens Corporation.

1 INTRODUCTION

In a distributed parallel program, tasks are generated and distributed to multiple processors to be processed simultaneously. Load imbalance is a serious impediment to achieving good performance as it leaves some processors idle, when they could be working to make progress. While global load balancing should still be an issue in the whole operating system's concern, our focus is on balancing parallel tasks within an application. Since minimizing the execution time of an application is more important than average response time, each processor needs to keep making progress rather than merely to have a balanced load. Although the latter state may finally lead to the former, this is not a primary goal to shorten the finish time. From a program's viewpoint, loops are the largest source of task parallelism in a parallel application. A loop is called a *parallel loop* (DOALL-loop) if there are no data dependences among all iterations. The question of how to allocate an iteration to a particular processor for minimizing the total execution time is known as a loop scheduling problem [19, 12, 15, 20, 5].

If there are I uniformly distributed iterations, and P identical processors, load can be balanced simply by assigning I/P iterations to each processor. Since both factors may not be known in advance or may vary substantially, such a static method is often difficult or inefficient. *Self-scheduling* (SS) [19] is the simplest dynamic solution. It assigns a new iteration to a processor only when the processor becomes available. However, this method requires tremendous synchronization overhead; to be practical, hardware support to fast barrier synchronization primitives is desirable. *Uniform-sized chunking* (CSS) reduces such synchronization overhead by sending K iterations instead of one [12]. In this method, the overhead is amortized to $1/K$, but the possibility of load imbalance increases when K is increased. In *guided self-scheduling* (GSS), the fixed chunk function (K) is replaced with a non-linearly decreasing chunk function in order to reduce the overhead at the beginning of a loop by allocating larger chunks, and also to reduce the chance of load imbalancing at the end of the loop by allocating smaller chunks [15]. *Trapezoid self-scheduling* (TSS) uses a linearly decreasing chunk function, which helps to reduce scheduling overhead while still maintaining a reasonable balance [20].

Recently, networks of workstations have emerged as viable candidates for running parallel applications. To our knowledge, the first work on parallel loop scheduling in a network of heterogeneous workstations was done by Cierniak *et al.* [5]. They considered three aspects of heterogeneity — loop, processor, and network — and developed algorithms for generating optimal and sub-optimal schedules of loops. Two major limitations are that it is static and that the loop heterogeneity model is linear. In this paper, we present a dynamic load balancing method for parallel loops of more general patterns, since many non-scientific applications such as the DNA sequence search problem [4] or the Mandelbrot set computation [8], which are good candidate applications for workstation clusters, often do not carry conventional regular loop patterns. The unpredictable patterns can even be detrimental to those improvements [12, 15, 20], although the pure SS scheme is orthogonal to the loop patterns.

1.1 Programming Environment

Networks of workstations have by nature easy-to-change configurations; programs must be adapted accordingly whenever the hardware configuration has been changed. Without having to manually rewrite module programs, diverse performance-related configurations can be incorporated according to the given characteristics of an application program and its working platform with the aid of an automatic adaptation tool. We assume such a distributed parallel application is written to the RPC (Remote Procedure Call) paradigm. Normally, RPC does not consider the case of multiple servers for the same function — except for some variations like PARPC [14] and MultiRPC [16]. Our work, called CORD (Configuration-level Optimization of RPC-based Distributed programs), is a framework for automatically generating all necessary executables from RPC-based distributed programs according to a configuration-level description intended for high performance [10]. From the concern of software engineering, module interconnection activity is understood to be an essentially distinct and different intellectual activity from that of implementing individual modules; that is "programming-in-the-large" is distinct from "programming-in-the-small" [6]. We apply this separating principle in order to isolate *performance factors* from the module programming level with the aid of the CORD that eventually intergrates all information to prepare executables. The load balancing method presented in this paper represents a feature of the CORD system when an augmented configuration program specifies server replication to deal with parallel loops.

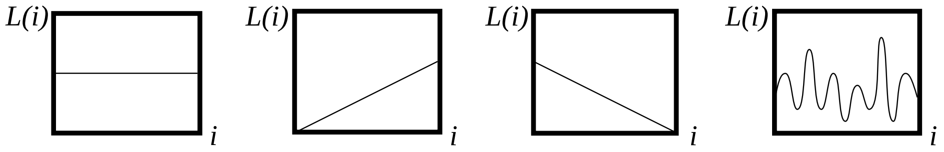
1.2 Motivations

Under a heterogeneous network of workstations, a simple policy like equally distributing workloads to multiple processors may lead to a parallelization anomaly. That is, the execution time of the given workload may take longer even if the number of workstations is increased. Suppose there are n processors $\{P_1, \ldots, P_n\}$, and T identical tasks. Let τ_i be the number of tasks per unit time that the processor i can process. In equal distribution, each processor has T/n numbers of tasks. The execution time of the program is determined by the critical processor that has the smallest τ_i value; let's say it is τ_{min}. Then the execution time is $\frac{T/n}{\tau_{min}} = \frac{T}{n\tau_{min}}$. Now, let's add a new processor of τ_{new} to the cluster for the application. Each processor will have $T/(n+1)$. Therefore, if $\tau_{new} < \frac{n}{n+1}\tau_{min}$, the execution time of $(n+1)$-processors cluster is $\frac{T}{(n+1)\tau_{new}}$, which is longer than that of n processors!

One may want to get around this problem by allocating tasks according to the known computing power of each processor [9, 3]. However, their methods were static, thus of limited usefulness. Dynamic loop scheduling methods can deal with more general cases, but the centralized nature of the methods — the central processor that generates sub-tasks has to manage all other processors — may cause a bottleneck in a network of many workstations. For example, if there are 100 servers, and if a master needs 10^{-2} second to prepare and send a task, the master would create a bottleneck unless the average time for each server to finish a task is greater than one second. In our experimentation with the Mandelbrot set computation on $[0.5, -1.8]$ to $[1.2, -1.2]$ using a 400×400 pixel window, the program reached its saturation point at 25 workstations under the *self-scheduling* scheme. To avoid such a situation, sub-tasks should be sufficiently large grained compared to communication overheads, but it is not likely considering relatively high communication costs in workstation clusters. Since there are many "embarrassingly parallel" applications, a decentralized load balancing scheme is called for. We present such a method that can reduce the overheads by means of establishing proper migration topology based on the known computing powers of the processors involved.

1.3 Our Approach

Parallel tasks ("objects") and their working platforms ("bases") are two ingredients in parallel processing. Nonetheless, only the "objects" part has been the focus of load balancing. There has been nothing wrong in this because the "bases" part has been mostly fixed. Meanwhile, workstation clusters have become viable platforms for parallel processing. As mentioned before, the conventional global load balancing and dynamic loop scheduling methods become problematic when they are employed to applications on workstation clusters. One of the key issues in dynamic load balancing is how to reduce the accompanying overheads. The main idea of our approach is balancing the "bases" to facilitate balancing the "objects;" i.e. we construct a special migration



(a) Constant (b) Increasing (c) Decreasing (d) Irregular

Figure 1: Four typical parallel loops.

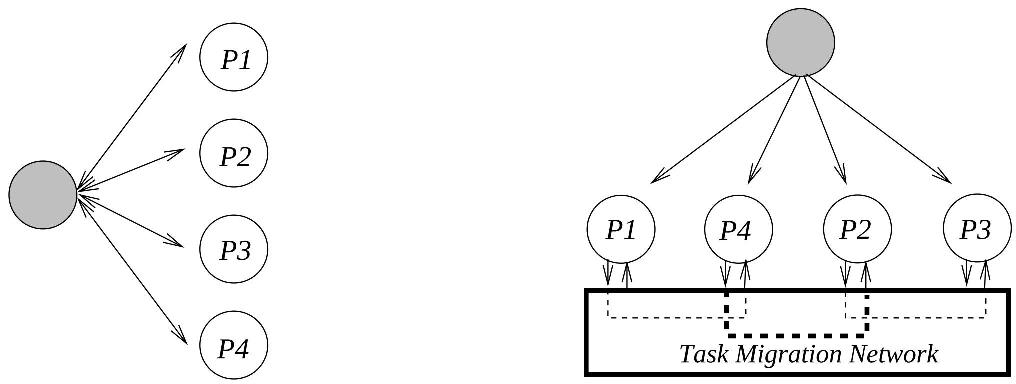
topology based on relative processor speeds in order to reduce migration overheads. Basically our method adopts demand-driven migrations the same way that dynamic loop scheduling methods do. Well-constructed topology reduces unnecessary migrations. The theme of this paper is how to construct such a topology that aims to do demand-driven migrations in a decentralized way for efficient load balancing.

1.4 Outline

Section 2 delineates models of parallel loops that need to be balanced and of workstation clusters that process those parallel tasks. Section 3 formally describes the cluster model and presents how to construct such a cluster and its corresponding task migration network based on the model. Section 4.1 provides preliminary characteristics that explain task migration behaviors under our method. Section 4.2 is devoted to the complexity issues incurred by migration using the results in Section 4.1. Section 4.3 addresses initial load distributions. In Section 5, we show experimental results using our implementation of the balancing scheme for an irregular and unpredictable loop.

2 LOOP AND WORKSTATION CLUSTER MODELS

In this section, we classify four typical parallel loop patterns that affect performance of load balancing schemes based on workload distribution in an iteration space. Next, we discuss our workstation cluster model to deal with those diverse patterns, especially if the workstations involved are heterogeneous.



(a) Loop Scheduling (b) Our Approach

Figure 2: Topologies in workstation cluster model for load balancing.

2.1 Loop Model

Fig. 1 shows four typical parallel loops where $L(i)$ represents the execution time of the i-th iteration. The workload may be uniformly distributed over an iteration space as shown in Fig. 1 (a). It may also be non-uniform but *linearly* distributed as in Figs. 1 (b) and (c); this kind of distribution is often contained in scientific programs. Finally, as in Fig. 1 (d), the workload may be quite irregular. Many non-scientific applications carry parallel loops of this type. The first three cases have been specially considered by conventional loop scheduling methods [15, 20, 5] in order to improve on the basic self-scheduling method.

Particularly for irregular loops, we can distinguish between the two cases: predictable vs. unpredictable. For example, the parallel tasks in the DNA sequence search problem [4] and the Mandelbrot set computation are all irregular, but the tasks in the first problem are predictable while the tasks in the second one are not. Of course, the above three loops are all predictable.

2.2 Workstation Cluster Model for Load Balancing

Fig. 2 shows two representative topologies in the workstation cluster model for parallel loops. Fig. 2 (a) represents the topology of traditional loop scheduling methods [19, 12, 15, 20], in which load migration is not performed. Instead, the main processor (shaded circle) prepares a set of tasks and allocates them to each server whenever the server demands them. Since the scheduling process is dedicated to the main processor (shaded circle), its chance of creating a bottleneck rises as the number of servers present on the network increases. Fig. 2 (b) illustrates the topology of our workstation cluster model. The main processor distributes workloads to all servers initially. Load balancing is attempted by task migration via pre-determined paths, deeming load state polling or exchange overhead unnecessary, unlike in global dynamic load balancing schemes. The migration is performed in a decentralized fashion between only the two processors involved. The workstation cluster model for load balancing is characterized by the following parameters:

- N: the number of workstations, $\{W_1, \ldots, W_N\}$.
- τ_i: the throughput of W_i, which is defined by the number of unit tasks per unit time.
- γ_{ij}: the amount of load to migrate from i to j.

3 LOAD BALANCING METHOD

Two important components of dynamic load balancing schemes are *transfer policy* and *location policy* [7, 11]. The transfer policy determines whether a task should be processed locally or remotely by transferring it at a particular load state. The location policy determines which process initiates the migration and its source or destination. These are for global load balancing from the OS's viewpoints. Multi-dimensional load vectors determine the load state of a processor. In our system, we aim to balance parallel loops in an application. A simple 'demand' message is enough to initiate load migration rather than load state exchange [11] or random polling of candidate processors [7] because the only load vector is the number of sub-tasks in a processor. The transfer policy then becomes simple: if a processor receives a request message for transfer from a processor that is running out of sub-tasks to work on, it migrates some of its sub-tasks to that processor.

Likewise, the location policy is now modified by the problem of establishing proper task migration paths. Workstation clusters have virtually no restrictions on topology for migration. It may be assumed that any two point-to-point communication overheads are equal, but identifying the optimal sender and receiver pair is essential. Considering all possible candidates for sender (or receiver) to migrate the excess load causes high overhead, but it is avoidable. The key is how

```
/* P_i:  sender */
for (i = 0; i < taskcnt; i++) {
    if (pvm_nrecv( P_j, MoreTaskReq )) {
        /* a request arrived */
        n = (taskcnt-i+1) * Ratio_ij;
        /* Migrate to P_j */
        if (n) {
            pvm_initsend( PvmDataDefault );
            pvm_pkint( &n, 1, 1 );
            pvm_pkint( &TaskQ[i], n, 1 );
            pvm_send( P_j, TaskMigrating );
            i += n;
            continue;
        }
    }
    /* loop body on TaskQ[i] */
}
```

```
/* P_j:  receiver */
LOOP:
   for (i = 0; i < taskcnt; i++) {
       /* loop body on TaskQ[i] */
   }
   /* Check the partner processor P_i */
   pvm_initsend( PvmDataDefault );
   pvm_pkint( &more, 1, 1 );
   pvm_send( P_i, MoreTaskReq );
   /* Wait until killed by parent */
   while(1)
       if (pvm_nrecv( P_i, TaskMigrating )) {
           /* migrated tasks arrived */
           pvm_upkint( &taskcnt, 1, 1 );
           pvm_upkint( TaskQ, taskcnt, 1 );
           goto LOOP;
       }
```

Figure 3: Programs generated for a migration path in Fig. 2 (b).

to identify the busy and the idle processors in the middle of computations. Since the relative processing speeds of workstations in a cluster are known in advance, the possible senders and receivers of migrations are not unknown — momentary overload by other activities is the reason for uncertainty.

In this section, we present how to construct such a task migration network as shown in Fig. 2 (b). Once the network is constructed, load balancing is pursued through task migration on it. For example, each pair connected in a dotted line in Fig. 2 (b) ($P_i \rightarrow P_j$) is a basic unit of migration; whenever the faster processor (P_j) depletes its workload, it demands that its pre-determined partner P_i share some of P_i's workload, and P_i migrates γ_{ij} of its current workload to P_j. Fig. 3 shows the generated source codes for such a connection. First, we will formally define the cluster model in Section 2.2. Then, we will describe how to construct such a cluster and its corresponding migration network based on the model.

A cluster is a bipartite form of (w_s, w_f), in which w_s is slower than w_f: i.e. $\tau_s < \tau_f$. Throughout the paper, we use the notation (τ_s, τ_f) interchangeably with the notation (w_s, w_f) when we focus on throughputs. An entire workstation cluster is defined as follows:

Definition 3.1 The *cluster tree* (CT) of N workstations $\{W_1, \ldots, W_N\}$ is a binary tree $CT = (V, E_{left} \cup E_{right})$, where

- The vertices V represent *clusters*. A distinguished vertex 'root' represents an entire cluster, and the right sub-cluster is faster than (or equal to) the left sub-cluster.
- E_{left} is a set of edges to the left sub-trees. E_{right} is a set of edges to the right sub-trees.
- If $(c, v) \in E_{left}$ and $(c, w) \in E_{right}$, a load migration path exists from v to w. When v and w are not terminal nodes, the path is established from the fastest node in cluster v, which is the rightmost terminal in the subtree of v, to the slowest node in cluster w, which is the leftmost terminal in the subtree of w.

Terminal vertices are individual workstations. Each terminal v is associated with its throughput τ_v. Throughput of non-terminal node $C = (v, w)$ is defined by $(\tau_v + \tau_w)$, which is explained by Theorem 4.3. $\square$

Definition 3.2 In a cluster $C_1 = (\tau_1, \tau_2)$, the *balance ratio* B_{C_1} is defined by $\frac{(\tau_2 - \tau_1)}{(\tau_2 + \tau_1)}$. A cluster $C_1 = (\tau_1, \tau_2)$ is said to be *more balanced* than another cluster $C_2 = (\tau_3, \tau_4)$, if the *balance ratio* of C_1 is less than that of C_2, i.e. $\frac{(\tau_2 - \tau_1)}{(\tau_2 + \tau_1)} < \frac{(\tau_4 - \tau_3)}{(\tau_4 + \tau_3)}$. $\square$

Definition 3.3 A cluster $C_1 = (\tau_1, \tau_2)$ is *faster* than another cluster $C_2 = (\tau_3, \tau_4)$ if τ_{C_1} is greater than τ_{C_2}, or if τ_{C_1} is equal to τ_{C_2} and C_1 is *more balanced* than C_2. $\square$

In the extreme case that τ_1 is equal to τ_2, the *balance ratio* is zero; thus load is perfectly balanced. Likewise, in the other extreme in which τ_2 is much greater than τ_1, the ratio is asymptotically 1. The *balance ratio* in a cluster can be related to the amount of load migration. When the components in a cluster are equally loaded initially, if the cluster is perfectly balanced, then no intra-cluster migration is necessary. In other words, the more balanced a cluster is, the less migration is needed.

The process of constructing a cluster tree from a set of workstations is done in recursive "bitonic" fashion. First, workstations in the set $\{w_1, \ldots, w_n\}$ become terminal nodes in the tree. They are sorted in ascending order by their throughputs. Let the sorted set be $\{w'_1, \ldots, w'_n\}$. The fastest one (w'_n) is coupled with the slowest one (w'_1), the second fastest one (w'_{n-1}) is coupled with the second slowest one (w'_2), and so forth. The couples come to have parents in the tree, i.e. $\{c_1 = (w'_1, w'_n), \ldots, c_{n/2} = (w'_{n/2}, w'_{n/2+1})\}$, which are likewise sorted by their throughputs.

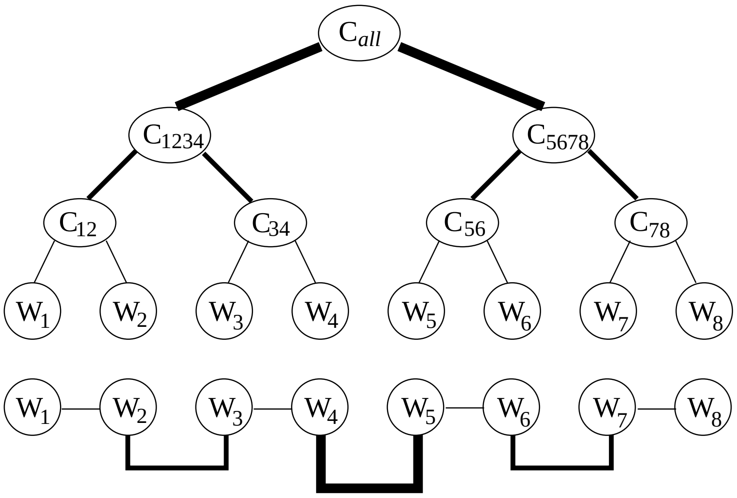


Figure 4: A cluster tree and its corresponding task migration paths.

Again, they are coupled in bitonic fashion. This process continues until it reaches a single cluster. Notice that the cluster of the two identical components still needs an intra-cluster migration because an equal distribution is not always possible. Once such a tree is constructed, the task migration topology is determined as follows:

Algorithm 3.1: *Task migration network from CT*

Begin

  For all clusters (non-terminal nodes) c in CT

    For two children v and w such that $(c, v) \in E_{left}$ and $(c, w) \in E_{right}$

      if (v, w are terminals) **then** *CONNECT* v *TO* w

      else *CONNECT* RightmostTerminal(v) *TO* LeftmostTerminal(w)

End

Fig. 4 shows the relationship between the cluster tree and the migration topology. For example, the rightmost terminal of C_{1234} is W_4, and the leftmost terminal of C_{5678} is W_5, so the link for the root cluster C_{all} is constructed between W_4 and W_5. The thicker links denote higher level links; they will be used only if the load cannot be balanced through the lower links.

4 ANALYSIS OF MIGRATION BEHAVIORS

There are two important concerns in devising a load balancing scheme [7]. First, the overhead should not negate the benefits of an improved load distribution. Next, the potential migration instability[1], in which processors spend too much time transferring tasks, should be avoided. Our method is orthogonal to the stability issue because a demand is issued only when the processor is idle. In this section, we present an analytic result on the overheads incurred by our method. We start with an example case to explain our method qualitatively.

Example 4.1 Suppose there are four processors $P1, P2, P3$ and $P4$ that have N identical tasks initially and we know their relative throughputs, which are $\tau, 2\tau, 3\tau$ and 4τ. When a load state of a potential sender $P1$ is probed by other processors, migration to $P2$ or $P3$ would be wasteful because its resulting resolution of $P1$'s overloaded state may be merely temporal. Since $P4$ is the fastest, the then-migrated load may have to be migrated again to $P4$, while a single migration directly to $P4$ would have been more efficient. Thus we can say the $P1$ has the greatest affinity to $P4$ among all possible receiver candidates. □

The above example suggests that the slowest processor should be connected to the fastest processor, and the second slowest one is to the second fastest one, and so on, in bitonic fashion. The resulting pairs would tend to be *more balanced* in terms of the combined throughputs. We will elaborate on the effects of this kind of bitonic pairing in Section 4.1. This method calls for load migration to be done in as much bulk as possible. One ten-byte sized load migration is cheaper than ten one-byte sized load migrations. This is particularly important in workstation clusters where the communication overheads are still high.

Example 4.2 Let us consider the topology of $P1 \rightarrow P4$ and $P2 \rightarrow P3$ as shown in Fig. 2 (b). Throughputs are the same as in Example 4.1. In this case the combined throughputs of the two sub-clusters turn out to be equal. That is, no further load migration is necessary through the link between the two clusters $(P1, P4)$ and $(P2, P3)$! □

[1]For example, in a two-processor system where both are overloaded, they may continuously migrate each part of loads to the other processor, which does not improve the situation at all.

However, now that cluster $(P2, P3)$ is *more balanced* than cluster $(P1, P4)$, the resulting decrease in the intra-cluster migration makes cluster $(P2, P3)$ process more tasks. That is why this cluster is defined as the faster one in Def. 3.3. In general, such an ideal case may not be common in real situations; throughputs may fluctuate in the middle of computing and initial distributions are not always equal. For the case that the load is not balanced in the first cluster for some reason, we continue to balance the load through inter-cluster migrations. In the following analysis, we use $\gamma_{ij} = 1/2$, for all i, j, which guarantees uni-directional migration is enough for load balancing (notice P_j is faster), although more aggressive choice like $\gamma_{ij} = \tau_i/\tau_j$ may reduce overheads.

4.1 Preliminaries

To examine migration overhead, we need a communication time model. The conventional approach to modeling communication time for transferring a message of m bytes is a simple linear function, i.e. $T_{comm} = \alpha + \beta m$, where α is startup time and β is transfer time per byte [2]. The empirical values for α and β under the PVM system [18] at LAN-based clustered workstations are 4.527 $msec$, 0.0024 $msec$ and 1.661 $msec$, 0.00157 $msec$ for datagram and stream transmission cases, respectively, which imply $\alpha \gg \beta$ [17].

In Theorems 4.1 and 4.2, we compute the total number of migrated tasks (β's multiplier) and the frequencies of migrations (α's multiplier) in a cluster. Furthermore, we also illustrate an important characteristics of our method, which is that balance ratio gets improved as clustering happens at higher levels.

Theorem 4.1 In a cluster $C = (v, w)$ where v and w are terminal nodes in CT, and they have initially loaded N identical tasks respectively, the total number of tasks to be migrated from v to w to meet the finish times at both processors is $\frac{\tau_w - \tau_v}{\tau_w + \tau_v} N$, i.e. the balance ratio of C times N.

Proof: Let us determine the general terms of the number of tasks to be migrated from v to w at the time w becomes idle. Since w is faster than v, w's first incidence of task depletion occurs after $\frac{N}{\tau_w}$; thus the number of tasks in the first migration is half of what remains in v at that time, which is $\frac{1}{2}(N - \frac{N}{\tau_w} \cdot \tau_v) = \frac{N}{2}(1 - \frac{\tau_v}{\tau_w})$. Notice that τ_v/τ_w is less than 1. T_w, the total number of tasks that are eventually processed by w, is a summation of the following series:

$$\begin{aligned} T_w &= N + \frac{N}{2}(1 - \frac{\tau_v}{\tau_w}) + \frac{N}{4}(1 - \frac{\tau_v}{\tau_w})^2 + \cdots = N \sum_{i=0}^{\infty} \frac{1}{2^i}(1 - \frac{\tau_v}{\tau_w})^i \\ &= N \lim_{k \to \infty} \frac{1 - (\frac{1}{2}(1 - \frac{\tau_v}{\tau_w}))^{k+1}}{1 - \frac{1}{2}(1 - \frac{\tau_v}{\tau_w})} = \frac{2N\tau_w}{\tau_v + \tau_w} \end{aligned}$$

Therefore, $\text{Migrated}_{v \to w} = T_w - N$, which yields $\frac{\tau_w - \tau_v}{\tau_w + \tau_v} N$. □

Theorem 4.2 In a cluster $C = (v, w)$ where v and w are terminal nodes in CT, and they have initially loaded N identical tasks respectively, the frequency of migration from v to w to meet the finish times at both processors is $\log_{\frac{1}{2}(1 - \frac{\tau_v}{\tau_w})} \frac{1}{N}$.

Proof: The general term in the series is $\frac{N}{2^k}(1 - \frac{\tau_v}{\tau_w})^k$. Thus, $k = \log_{\frac{1}{2}(1 - \frac{\tau_v}{\tau_w})} \frac{1}{N}$. □

Theorem 4.3 In a cluster $C = (v, w)$ where v, w are arbitrary nodes in CT, and they have initially loaded N identical tasks, the combined throughput of a cluster $C = (v, w)$ is $\tau_v + \tau_w$, assuming no migration overhead.

Proof: Suppose v and w are terminal nodes in CT. In Theorem 4.1, the total number of tasks processed by v and w is given by $\frac{2N\tau_v}{\tau_v + \tau_w}$ and $\frac{2N\tau_w}{\tau_v + \tau_w}$, respectively, and the finish time is $\frac{N}{(\tau_v + \tau_w)/2}$ at either processor. As cluster C have loaded $2N$ tasks in total, this may be interpreted to mean that the *de facto* throughputs of the cluster is $\tau_v + \tau_w$. Now let us assume this holds for two clusters $C_1 = (\tau_1, \tau_2)$ and $C_2 = (\tau_3, \tau_4)$; i.e. τ_{C_1} and τ_{C_2} are $\tau_1 + \tau_2$ and $\tau_3 + \tau_4$, respectively. For a cluster $C = (C_1, C_2)$ (we can assume C_1 is slower without loss of generality), we can calculate the number of tasks processed by C_2 as follows:

$$T_{C_2} = N \sum_{i=0}^{\infty} \frac{1}{2^i}(1 - \frac{\tau_{C_1}}{\tau_{C_2}})^i = N \cdot \frac{2\tau_{C_2}}{\tau_{C_1} + \tau_{C_2}} = N \cdot \frac{\tau_{C_2}}{(\tau_{C_1} + \tau_{C_2})/2}$$

By induction, this completes our proof. □

Theorem 4.3 implies that the sum of the two throughputs in a cluster may represent the combined throughput of the cluster so that we can cluster recursively in bitonic fashion. The real combined throughput can be yielded by subtracting the throughput loss incurred by migration overheads (see Section 4.2) from that amount.

Theorem 4.4 If there are two clusters $C_1 = (\tau_1, \tau_4)$ and $C_2 = (\tau_2, \tau_3)$, and C_1 is slower than C_2 (i.e. τ_{C_1} is less than τ_{C_2}), then another cluster $C = (C_1, C_2)$ is always *more balanced* than the *less balanced* cluster between C_1 and C_2.

Proof: Consider the case when B_{C_1} is greater than B_{C_2} (i.e. C_1 is *less balanced* than C_2). Due to the property of bitonic coupling, $\tau_1 \leq \tau_2 \leq \tau_3 \leq \tau_4$ must hold. Let us write $\tau_2 = a\tau_1$, $\tau_3 = ab\tau_1$ and $\tau_4 = abc\tau_1$, where $a, b, c \geq 1$. By Theorem 4.3, B_C is yielded by $\frac{\tau_2+\tau_3-\tau_1-\tau_4}{\tau_1+\tau_2+\tau_3+\tau_4}$. That is, $B_{C_1} = \frac{abc-1}{abc+1}$ and $B_C = \frac{a+ab-(abc+1)}{abc+ab+a+1}$. Since $(abc + ab + a + 1) \cdot (abc - 1) - (a + ab - (abc + 1)) \cdot (abc + 1) = 2abc(abc+1) - 2a(b+1) \geq 0$, B_C is less than or equal to B_{C_1}. But if $2abc(abc+1) - 2a(b+1) = 0$, all a, b, c must be 1, which implies $\tau_1 = \tau_2 = \tau_3 = \tau_4$ that contradicts the given assumption ($\tau_{C_1} < \tau_{C_2}$ or $B_{C_1} > B_{C_2}$). Hence B_C is strictly less than B_{C_1}. Likewise, when B_{C_1} is less than B_{C_2} (i.e. C_2 is *less balanced* than C_1), we also can show that B_C is less than B_{C_2} — now $\tau_2 \leq \tau_1 \leq \tau_4 \leq \tau_3$ holds. Finally, consider the case when B_{C_1} is equal to B_{C_2}. Again, due to the property of bitonic coupling, this condition implies $\tau_1 = \tau_2 = \tau_3 = \tau_4$, which is a contradiction. This completes the proof. $\square$

Theorem 4.4 contains an important subtlety. It implies the amount of inter-cluster migration is always less than that of intra-cluster migration in a critical sub-cluster. Since migrations through a higher-level link may need multi-hop communications, they result in higher overheads. Theorem 4.4 assures that the amount of migrations of such higher overheads get smaller. Consequently, the complexity of migration overheads is bounded.

4.2 Complexities of Task Migration Overhead

Consider the topologies in Fig. 2 (a) and (b) extended to p processors and the total number of tasks are pN. Self-scheduling requires $pN(\alpha + \beta)$, where N is the total number of tasks between a master and its servers. Putting aside the fact that the master can easily create a bottleneck in that topology, we investigate the complexity of our method and compare it with that of self-scheduling.

The worst case happens when the fastest processor (the rightmost one in a cluster tree) is far faster than the remaining ones: i.e. $\tau_3 \gg \tau_1, \tau_4, \tau_2$ in Fig 2 (b). Let us calculate the overhead for a one-hop migration in this scenario. For example, in a link between $P2$ and $P3$, the total number of tasks to migrate is, by Theorem 4.1, $\frac{\tau_3-\tau_2}{\tau_3+\tau_2}N$. As $\tau_3 \gg \tau_2$, the number becomes N.

In other words, all of the task in a slower processor must be migrated to the infinitely faster one. Likewise, by Theorem 4.2, the frequency of migrations is given by $\log_{\frac{1}{2}} \frac{1}{N} = \log_2 N$. Thus, the one-hop overhead (OH_1) is $\alpha \log_2 N + \beta N$. Since the farthermost tasks need $p-1$ hops, we obtain the worst case complexity of migration overhead as follows:

$$OH_{worst} = \sum_{k=1}^{p-1} k \cdot OH_1 = \frac{1}{2} p(p-1)(\alpha \log_2 N + \beta N)$$

Recalling the facts that $\alpha \gg \beta$ and $N \gg p$, OH_{worst} can hardly be worse than $pN(\alpha + \beta)$. Now let us consider an average case where each processor contains the average number of tasks (N) at any moment during computation.[2] Consider a lowest-level cluster (v, w); i.e. v and w are terminal nodes in CT. By Theorem 4.2 and 4.1, the one-hop migration overhead is obtained as follows:

$$OH_1 = \frac{1}{1 - \log_2 \frac{\tau_v}{\tau_w}} \log_2 N \cdot \alpha + \frac{\tau_w - \tau_v}{\tau_w + \tau_v} N \cdot \beta$$

By Theorem 4.4, the balance ratio of a higher-level cluster is always less than the maximum of those of the two sub-clusters. That is, the maximum balance ratio among all clusters (v, w) at the lowest level is the maximum balance ratio of all clusters in an entire cluster tree. Let it be B_{max}. Then, no $(p-1)$ links in the topology can migrate more than $B_{max} \cdot N$ tasks. Therefore, the average case complexity of migration overhead is a lower bound of the following formula, where r_{max} is the maximum of $\frac{\tau_v}{\tau_w}$ for all clusters (v, w) at the lowest level in CT:

$$OH_{average} = \sum_{k=1}^{p-1} OH_1 = \frac{p-1}{1 - \log_2 r_{max}} \log_2 N \cdot \alpha + B_{max}(p-1)N\beta$$

Notice that $0 < r_{max} < 1$ and $0 < B_{max} < 1$. $OH_{average}$ is always better than $pN(\alpha + \beta)$. Furthermore, since $\alpha \gg \beta$ and $N \gg p$, it is significantly better in general.

Example 4.3 Let us consider Fig. 2 (b) again. Each processor initially has N identical sub-tasks. Throughputs are the same as in Example 4.1: i.e. $\tau, 2\tau, 3\tau, 4\tau$ for $P1, P2, P3$ and $P4$, respectively. For brevity, suppose all processors have constant throughputs, and we assume no migration overhead for the time being. Then the following table shows each snapshot of load

[2] Obviously this is a harsher condition than what a real average case needs to be, since the number of remaining tasks gets decreased as time goes by. Therefore, our obtained complexity is an upper-bound of the average complexity.

distribution under our load balancing method in case we chose $\gamma_{14} = \frac{4}{5}$ and $\gamma_{23} = \frac{3}{5}$ particularly.

	$P1$	$P4$	$P2$	$P3$
Initial Load	N	N	N	N
After $N/4\tau$	$3N/4$	0	$N/2$	$N/4$
After Load Migration	$3N/20$	$3N/5$	$N/2$	$N/4$
After $N/12\tau$	$N/15$	$4N/15$	$N/3$	0
After Load Migration	$N/15$	$4N/15$	$2N/15$	$3N/15$
After $N/15\tau$	0	0	0	0

Table 1: Snapshots of load distribution.

The table shows that total execution time is $\frac{N}{4\tau} + \frac{N}{12\tau} + \frac{N}{15\tau} = \frac{2N}{5\tau}$; in other words, the average throughput of this 4-processor cluster with $4N$ sub-tasks is 10τ. However, the real behavior deviates from this ideal behavior because of migration overheads. We calculate the overhead for two different choices of γ: when γ is taken proportionally based on throughput (**Case 1**) and when all $\gamma = \frac{1}{2}$ (**Case 2**).

Case 1: As shown in Table 1, migrations occur twice of amount $3N/5$ and $3N/15$, respectively. Thus, the overhead is yielded by $\alpha + \frac{3}{5}N\beta + \alpha + \frac{3}{15}N\beta = 2\alpha + \frac{4}{5}N\beta$.

Case 2: By Theorem 4.1, the number of tasks to migrate for $P1 \to P4$ and $P2 \to P3$ links is calculated as follows:

$$M_{P1\to P4} = \frac{4\tau - \tau}{4\tau + \tau}N = \frac{3}{5}N, \qquad M_{P2\to P3} = \frac{3\tau - 2\tau}{3\tau + 2\tau}N = \frac{1}{5}N$$

Similarly, by Theorem 4.2, the number of migrations that occur for the two links is as follows:

$$k_{P1\to P4} = \log_{\frac{1}{2}(1-\frac{1}{4})}\frac{1}{N} = \log_{\frac{3}{8}}\frac{1}{N}, \qquad k_{P2\to P3} = \log_{\frac{1}{2}(1-\frac{2}{3})}\frac{1}{N} = \log_{\frac{1}{6}}\frac{1}{N}$$

Thus, the overhead is yielded by

$$\begin{aligned} OH &= \alpha(k_{P1\to P4} + k_{P2\to P3}) + \beta(M_{P1\to P4} + M_{P2\to P3}) \\ &= \alpha(\log_{\frac{3}{8}}\frac{1}{N} + \log_{\frac{1}{6}}\frac{1}{N}) + \frac{4}{5}N\beta \approx 3.63\log N\alpha + \frac{4}{5}N\beta \end{aligned}$$

In either case, the overhead is much less than that of *self-scheduling*, which is $4N(\alpha + \beta)$. □

4.3 Initial Load Distribution

While any initially distributed load should be balanced through a dynamic load balancing method, the resulting overhead is associated. We discuss now the initial load distribution issue that can lower overhead, compared with the equal distribution that was assumed for analysis in the previous sections.

When loops are predictable (see Section 2.1), there are two cases: one is when we know the amount of the required computation exactly, as in Fig. 1 (a), (b), (c) and sometimes (d), and the other is when we can determine just the orderings, like in the DNA sequence search problem [4]. For the former case, as $L(i)$ is known in advance, if we distribute proportionately according to each processor's throughput, we can reduce the likelihood of migration. In other words, the processor P_i with τ_i will get $\tau_i \sum_i L(i)/\sum_k \tau_k$. Dynamic adjustments to this approximation are made by our load balancing method. In a lowest-level cluster (v, w) in CT, if we allocate $\lfloor \tau_i \sum_i L(i)/\sum_k \tau_k \rfloor$ to v, and $\lceil \tau_i \sum_i L(i)/\sum_k \tau_k \rceil$, Since v is slower than w, uni-directional migration is enough. If we cannot guarantee the faster processor finishes earlier, the migration paths must be bi-directional as in the following cases. For the latter case, we cannot initialize in the above way as the value of $L(i)$ is unknown. The LPT (Largest Processing Time first) algorithm [1] is for this class of loop models. The tasks are sorted in descending order based on execution time $L(i)$. Each processor should process the largest task first. Otherwise, an unfortunate processor may happen to take a large task (say, about 100 times larger than the small ones) as a last one at the near end of all computations, which results in a load imbalance — other processors are idle because few tasks left to migrate at this moment.

When tasks are not "orderable" and quite irregular like in the Mandelbrot set computation problem, we can neither quantify the loads to proportionately distribute to processors of diverse throughputs nor sort in decreasing order and apply the LPT algorithm. No general heuristics can be used — random distribution does not need to be worse.

5 EXPERIMENTS

To demonstrate the performance of our method, we conducted our experiment on 16 workstation clusters using PVM message passing systems. The example program was Mandelbrot set

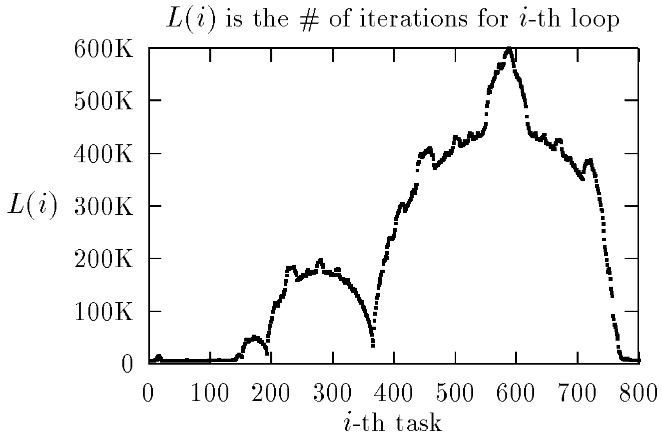


Figure 5: The load distribution pattern of a loop in the Mandelbrot set computation.

computation on $[0.5, -1.8]$ to $[1.2, -1.2]$ using a 800×800 pixel window. This program contains unpredictably irregular loops as shown in Fig. 5, which cannot be analyzable as in Section 4. The x-value indicates the x-th row in an outer loop. The y-value is the number of inner iterations $(L(x))$ to compute the corresponding x-th row. The total number of sub-tasks are 800, and the result size of a sub-task is 800 in integers: one integer per pixel.

We have initially distributed those tasks in a round-robin style. A variety of heterogeneous workstations have been used as shown in Fig. 6 (a) which shows the execution time for each of 16 workstations[3] to compute the given Mandelbrot set; the range is from 250 seconds to 2000 seconds. The results by 16-workstations cluster are given by Fig. 6 (b). The dotted boxes represent the finish times of each workstation under the pure self-scheduling method, which substantiate the expected good load balance. The result by our method is seemingly imbalanced but the actual finish time is much improved. Perfect balance may be good but the evaluation should be based on how much its overheads negate its resulting benefits.

taskcnt	1	2	3	4	5	6	7	10	11	12	17	19	20	30
freq	13	6	3	3	3	1	4	1	1	1	1	1	1	1

Table 2: The sizes of migration units and the frequencies of migrations

Table 2 summarizes the size of each migration and its frequency that are counted in our experi-

[3] 1 SPARCstation 20, 3 SPARCstation 5's, 2 SPARCstation 10's, 2 DECstation 5000/25's, 4 SPARCstation IPX's, 2 DECstation 23/100's, 2 SPARCstation IPC's are used.

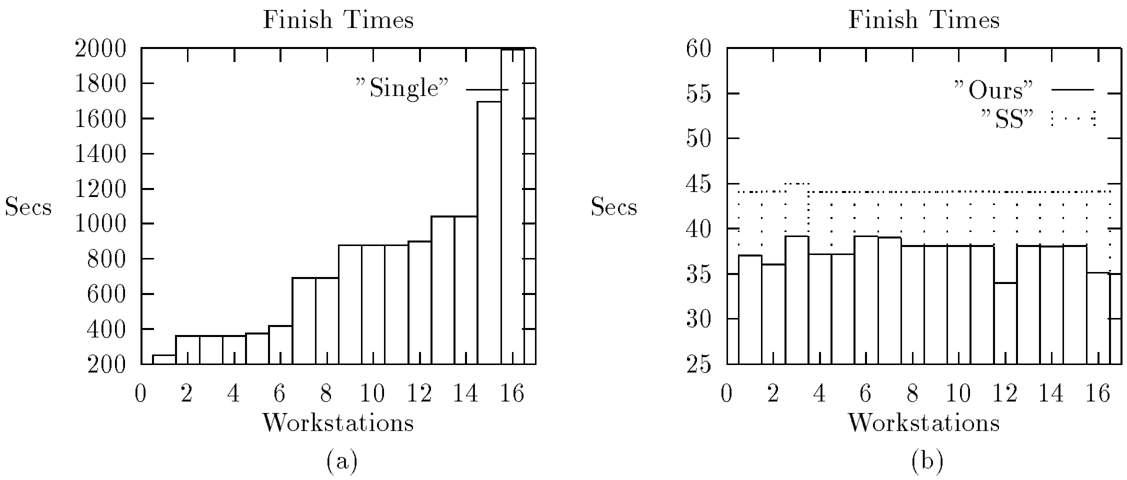


Figure 6: Execution times: Mandelbrot set computation on [0.5,-1.8] to [1.2,-1.2]

mentation. For example, the single-task migration occurred 13 times, and the 30-tasks migration occurred once, etc, during the entire task migration attempts. In the table, we can compute the total occurrences of migrations by summing all frequencies up, that is 40. If we calculate this figure from our formula on $OH_{average}$, that is $\frac{p-1}{1-\log_2 r_{max}} \log_2 N$, where $p = 16$, $N = 800/16 = 50$, $r_{max} = 692/693 \approx 1$. This formula gives $15 \log_2 50 \approx 84.7$. Considering this formula is obtained as an upper bound, the experimental value is said to conform to the theoretically obtained value. Although the theoretical model does not exactly with our experimental environments, the model gives us a reasonable implication about the migration behaviors in general cases.

6 CONCLUSION

We have presented a new decentralized load balancing method for parallel tasks in heterogeneous workstation clusters to deal with various patterns of parallel loops. We discussed why the conventional global dynamic load balancing methods are not adequate to our application area. Loop scheduling schemes that have been useful under shared-memory multiprocessor machines cause a bottleneck in workstation clusters because the communication overheads are so high. To our knowledge, migration topology for load balancing is considered for the first time. The topology has not been considered important heretofore because sometimes it is given in a hard-wired form [13] or it is meaningless where distributed load patterns cannot be assumed to be known in advance [7, 11]. More interesting topologies can be studied in the future. We have shown analytically that the overhead of our method is lower than that of the self-scheduling scheme when an "predictability" condition is given. We have also provided some experimental data for cases when the loop pattern is unpredictably irregular.

References

[1] K. P. Belkhale and P. Banerjee. An approximate algorithm for the partitionable independent task scheduling problem. In *Proceedings of '90 International Conference on Parallel Processing*, August 1990.

[2] L. Bomans and D. Roose. Benchmarking the iPSC/2 hypercube multiprocessor. *Concurrency: Practice and Experience*, Vol. 1(1):3–18, September 1989.

[3] Clemens H. Cap and Volker Strumpen. Efficient parallel computing in distributed workstation environments. *Parallel Computing*, Vol. 19:1221–1234, 1993.

[4] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, Vol. 21(6):322–356, September 1989.

[5] M. Cierniak, W. Li, and M. J. Zaki. Loop scheduling for heterogeneity. In *Proceedings of the 4th International Symposium on High-Performance Distributed Computing*, August 1995.

[6] F. DeRemer and H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, Vol. 2(2), June 1976.

[7] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, Vol. 12(5):662–675, May 1986.

[8] J. D. Foley, A. van. Dam, S. K. Feiner, J. F. Hughes, and R. L. Phillips. *Introduction to Computer Graphics*. Addison-Wesley Publishing Company, 1993.

[9] A. S. Grimshaw, J. B. Weissman, E. A. West, and Jr. E. C. Loyot. Metasystems: An approach combining parallel processing and heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, Vol. 21:257–270, 1994.

[10] T.-H. Kim and J. M. Purtilo. Configuration-level optimization of RPC-based distributed programs. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, May 1995.

[11] Philip Krueger and Niranjan G. Shivaratri. Adaptive location policies for global scheduling. *IEEE Transactions on Software Engineering*, Vol. 20(6):432–444, June 1994.

[12] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, Vol. 11(10):1001–1016, October 1985.

[13] Frank C. H. Lin and Robert M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, Vol. 13(1):32–38, January 1987.

[14] Bruce Martin, Charles Bergan, and Brian Russ. PARPC: A system for parallel remote procedure calls. In *Proceedings of the International Conferences on Parallel Processing*, pages 449–452, 1987.

[15] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computer*, Vol. C-36(12):1425–1439, December 1987.

[16] M. Satyanarayanan and E. H. Siegel. MultiRPC: A parallel remote procedure call mechanism. Technical Report CMU-CS-86-139, Carnegie-Mellon University, 1986.

[17] B. K. Schmidt and V. S. Sunderam. Empirical analysis of overheads in cluster environments. *Concurrency: Practice and Experience*, Vol. 6(1):1–32, February 1994.

[18] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, Vol. 2(4):315–339, December 1990.

[19] P. Tang and P. C. Yew. Processor self-scheduling for multiple nested parallel loops. In *Proceedings of '86 International Conference on Parallel Processing*, pages 528–535, August 1986.

[20] T. H. Tzen and L. M. Ni. Dynamic loop scheduling for shared-memory multiprocessors. In *Proceedings of '91 International Conference on Parallel Processing*, pages II:247–250, August 1991.