Juggle: Proactive Load Balancing on Multicore Computers

Steven Hofmeyr
LBNL
shofmeyr@lbl.gov

Juan A. Colmenares
Par Lab, UC Berkeley
juancol@eecs.berkeley.edu

Costin Iancu
LBNL
ciancu@lbl.gov

John Kubiatowicz
Par Lab, UC Berkeley
kubitron@eecs.berkeley.edu

ABSTRACT

We investigate *proactive* dynamic load balancing on multicore systems, in which threads are continually migrated to reduce the impact of processor/thread mismatches to enhance the flexibility of the SPMD-style programming model, and enable SPMD applications to run efficiently in multiprogrammed environments. We present Juggle, a practical decentralized, user-space implementation of a proactive load balancer that emphasizes portability and usability. Juggle shows performance improvements of up to 80% over static balancing for UPC, OpenMP, and pthreads benchmarks. We analyze the impact of Juggle on parallel applications and derive lower bounds and approximations for thread completion times. We show that results from Juggle closely match theoretical predictions across a variety of architectures, including NUMA and hyper-threaded systems. We also show that Juggle is effective in multiprogrammed environments with unpredictable interference from unrelated external applications.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming – Parallel Programming; D.4.1 [**Operating Systems**]: Process Management – Scheduling

General Terms

Experimentation, Theory, Performance, Measurements

Keywords

Proactive Load Balancing, Parallel Programming, Operating System, Muticore, Load balancing.

1. INTRODUCTION

The primary goal of this research is to improve the flexibility of thread-level scheduling for parallel applications. Our focus is on single-program, multiple-data (SPMD) parallelism, one of the most widespread approaches in HPC. Traditionally in HPC systems, SPMD programs are scheduled with one thread per dedicated processor.[1] With the rise of multicore systems, we have an opportunity to experiment with novel scheduling approaches that use thread migration within a shared-memory node to enable both oversubscription and multiprogramming. Not only can this result in more flexible usage of HPC systems, but it could also make SPMD-style parallelism more suitable for the consumer market, where we expect environments to be multiprogrammed and unpredictable.

In SPMD programs each thread executes the same code on a different partition of a data set; this usually means that applications have a fixed number of threads. If the partitioning of the data is not uniform, *intrinsic* load imbalances will result. Most solutions to this problem involve extensions to the programming model, e.g., work stealing [1]. Often, the only (or easiest) way to achieve intrinsic load balance is to constrain the thread counts (e.g., to powers of two). However, if these thread counts do not match the available processor counts, or if running in multiprogrammed environments, *extrinsic* load imbalances can result.

Our goal is to address the problem of extrinsic imbalance through runtime tools that improve parallel-programming productivity without requiring changes to the programming model. We wish to reduce the effect of constraints so that SPMD applications can be more easily run in non-dedicated (multiprogrammed) environments, and in the presence of varying and unpredictable resource availability, such as changing processor counts. In general, we are interested in dynamic load-balancing techniques that allow us to run n threads on m processors, where $n \geq m$ and m is not a factor of n. In particular, we are concerned with the problem of *off-by-one* imbalances, where the number of threads on each processor is within one of each other, since we assume that we can always achieve an off-by-one imbalance for SPMD applications with a simple cyclic distribution of threads to processors.

Our investigations focus on the *proactive* approach to load balancing. In this approach application threads are continually, periodically migrated with the goal of minimizing the completion time of the thread set by getting all the threads to progress (more or less) at the same rate; i.e., we want each thread to ideally receive m/n processor time over the course of a computation phase.[2] If the load balancing period, λ, is small compared to the computation-phase length of the parallel application, then over time the *progress rate* of every

[1]We refer to a single processing element, whether it be a core or hyper-thread, as a *processor*.

[2]Most SPMD applications have a pattern of computation phases and communication, with barrier synchronization.

Copyright 2011 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
HPDC'11, June 8–11, 2011, San Jose, California, USA.
Copyright 2011 ACM 978-1-4503-0552-5/11/06 ...$10.00.

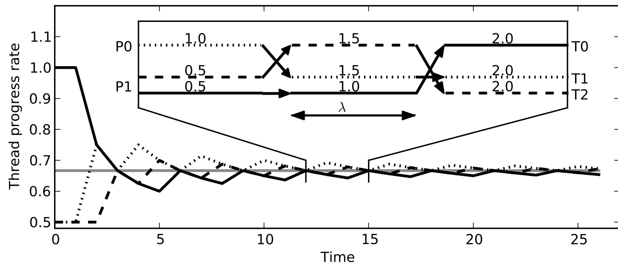

Figure 1: Load balancing three threads (T0, T1, T2) on two processors (P0, P1), using a load-balancing period of $\lambda = 1s$. The gray line indicates a progress rate of $m/n = 2/3$. In the top inset, migrations are indicated by the arrows between processors; the number is the total progress of the thread after that many balancing periods.

thread will tend to m/n, as illustrated in Figure 1. We contrast proactive balancing to *reactive* load balancing, where threads are rebalanced only when the load on a processor changes (i.e., some thread enters the barrier).

The concept of proactive load balancing is not new [17, 12, 7]. Our contributions are two-fold. Firstly, we present a practical decentralized, user-space implementation of a novel proactive load-balancing algorithm, called Juggle. In experiments, Juggle shows performance improvements over static balancing of up to 80% for UPC, OpenMP and pthreads benchmarks, on a variety of architectures, including NUMA and hyper-threaded systems. We also show that Juggle is effective in unpredictable, multiprogrammed environments. Secondly, we analyze Juggle and derive theoretical bounds and approximations that closely predict its performance. Our analysis is the first step towards a more comprehensive theoretical understanding of the fundamentals of proactive load balancing.

2. THE JUGGLE ALGORITHM

Juggle executes periodically, every λ milliseconds (the load-balancing period) and attempts to balance an application that has n threads running on m processors, where $n \geq m$. The objective is to assign threads to processors such that the threads that have made the least progress now run on the processors with the lightest loads (the "fastest" processors). In practice, we classify threads as either *ahead* (above-average progress) or *behind* (below-average progress), and we classify processors as either *fast* or *slow*, according to whether they are less or more heavily loaded than average, respectively. Juggle attempts to assign ahead threads to slow processors, and behind threads to fast processors.

For ease of use and portability, Juggle runs on Linux, in user-space without needing root privileges or any kernel modifications. Furthermore, it can balance applications from multiple SPMD runtimes (e.g., UPC, OpenMP, and pthreads) without requiring any modifications to the parallel application or runtime. The parallel application is launched using Juggle as a wrapper, which enables Juggle to identify the application threads as soon as possible and begin balancing with minimum delay. Juggle identifies the application threads by polling the `proc` file system; to keep this polling period to a minimum, the number of threads to be expected is a configuration parameter to Juggle. In addition, Juggle can be configured to regard a particular thread

```
1 Determine progress of threads (all balancers)
2 Determine fast and slow processors (all balancers)
3 [Barrier]
4 Classify threads as ahead and behind (single balancer)
5 Redistribute threads (single balancer)
6 [Barrier]
7 Migrate threads (all balancers)
8 [Barrier]
```

Figure 2: Pseudo code for Juggle.

as *idle* to accommodate applications that use one thread to launch the others (e.g., OpenMP applications).

Once Juggle has identified the expected number of threads it distributes them uniformly across all available processors, ensuring that the imbalance is never more than one. Threads are distributed using the `sched_setaffinity` system call, which enables a user-space application to force a thread to run on a particular processor. In Linux, threads that are moved from one processor to another do not lose out on scheduling time. Moreover, a thread that is pinned to a single processor will not be subsequently moved by the Linux load balancer, ensuring that the only balancing of the parallel application's threads is done by Juggle.

The implementation of Juggle is distributed across m *balancer* threads, with one balancer running on each processor. Every balancing period, all the balancer threads are woken, and execute the load-balancing code, as shown in Figure 2. All balancer threads execute lines 1, 2 and 7 in parallel, and a single balancer executes lines 4 and 5 while the other balancers sleep on a barrier. This serialization simplifies the implementation and ensures that all balancer threads operate on the same set of information. It is also worth noting that while the single balancer thread is involved in computation, the other processors are doing useful work running application threads. We discuss the scalablity of our approach in Section 2.5. The steps shown in Figure 2 are discussed in more detail below.

2.1 Gathering information

Because we do not want to modify the kernel, application, or runtime, Juggle infers thread progress and processor speeds indirectly using elapsed time. Each balancer thread independently gathers information about the threads on its own processor, using the `taskstats netlink` interface. For each thread τ_i that is running on a processor ρ_j (or more formally, $\forall \tau_i \in \mathcal{T}_{\rho_j}$), the balancer for ρ_j determines the elapsed user time, $t_{user}(\tau_i)$, system time, $t_{sys}(\tau_i)$, and real time, $t_{real}(\tau_i)$, over the most recent (the k-th) load-balancing period. From these, the balancer estimates the change in progress of τ_i as $\Delta P_{\tau_i}(k\lambda) = t_{user}(\tau_i) + t_{sys}(\tau_i)$, i.e., we assume that progress is directly proportional to computation time. The *total* progress made by τ_i after $k\lambda$ time is then $P_{\tau_i}(k\lambda) = P_{\tau_i}((k-1)\lambda) + \Delta P_{\tau_i}(k\lambda)$.

Using elapsed user and system times enables Juggle to easily estimate the impact of external processes, regardless of their priorities and durations. An alternative is to determine progress from the length of the run queue (e.g., two threads on a processor would each make $\lambda/2$ progress during a balancing period). In this case, Juggle would have to gather information on all other running processes, recreate

kernel scheduling decisions, and model the effect of the duration of processes. This alternative is complicated and error prone; moreover, changes from one version of the kernel to the next would likely result in inaccurate modeling of kernel decisions. Juggle avoids these issues by using elapsed time.

Once the progress of every thread on the processor ρ_j has been updated, the balancer uses this information to determine the speed of ρ_j as $\Delta\overline{P}_{\rho_j} = (1/|\mathcal{T}_{\rho_j}|)\sum_{\tau_i \in \mathcal{T}_{\rho_j}} \Delta P_{\tau_i}(k\lambda)$. That is, the speed of the processor is the average of the change in the progress of all the threads on ρ_j during the most recent load balancing period. The speed is later used to determine whether ρ_j will run threads fast or slow.

For applications that block when synchronizing (instead of yielding or spinning), processor idle time is discounted, otherwise the inferred speed of the processor will be wrong. For example, if a processor, ρ_1, has only one thread τ_1, and τ_1 finishes at $\lambda/4$ then the speed ($\lambda/4$) of ρ_1 will appear to be less than that of a more heavily loaded processor, ρ_2, that has two threads finishing at λ and effective speed of $\lambda/2$. To correct for this, the speed given by $\Delta\overline{P}_{\rho_j}$ is multiplied by a factor of $t_{real}(\rho_j)/(t_{real}(\rho_j) - t_{idle}(\rho_j))$

2.2 Classifying threads as ahead and behind

A single balancer classifies all application threads as either ahead or behind, an operation which is $O(n)$: one iteration through the thread list is required to determine the average total progress of *all* threads, denoted as $\overline{P}_{\mathcal{T}}(k\lambda)$, in the k-th balancing period, and another iteration to label the threads as ahead (above average) and behind (below average). Although external processes can cause threads to progress in varying degrees within a load-balancing period, simply splitting threads into above and below average progress works well in practice, provided that we add a small error margin ξ. Hence, a thread τ_i is classified as behind after the k-th balancing period only if $P_{\tau_i}(k\lambda) < \overline{P}_{\mathcal{T}}(k\lambda) + \xi$. Otherwise, it is classified as ahead.

2.3 Redistributing threads

The goal of redistribution is to place as many behind threads as possible on processors that can run those threads at fast speed; we say that those processors $\in \mathcal{P}_{fast}$ and have *fast slots*. If a processor $\rho_j \notin \mathcal{P}_{fast}$, then $\rho_j \in \mathcal{P}_{slow}$ and has *slow slots*. In practice, the presence of fast slots in processors can change depending on the external processes that happen to be running. For this reason, Juggle identifies the fast slots by first computing the average change in progress of all the processors as $\Delta\overline{P}_{\mathcal{P}} = (1/m)\sum_{j=1}^{m} \Delta\overline{P}_{\rho_j}$, and then counting one fast slot per thread on each processor with $\Delta\overline{P}_{\rho_j} > \Delta\overline{P}_{\mathcal{P}}$. This requires two passes across all processors (i.e., $O(m)$). The behind threads are then redistributed cyclically until either there are no more behind threads or no more fast slots, as illustrated by the pseudo code in Figure 3.

Although the cyclical redistribution of behind threads can help to spread them across the fast slots, the order of the choice of the next processor is important. The selection of the next slow processor (line 2 in Figure 3) starts at the processor which has threads with the least average progress, whereas the selection of the next fast processor (line 5) starts at the processor which has threads with the most average progress. This helps distribute the behind threads more uniformly across the fast slots. For example, consider two slow processors, ρ_1 with one ahead and one behind thread, and ρ_2 with two behind threads, and assume there is only one available fast slot. Here it is better to move one of the threads from ρ_2 (not ρ_1) to the fast slot so that both ρ_1 and ρ_2 may start the next load-balancing period with one ahead and one behind thread. This will only make a difference if the ahead threads reach the barrier and block partway through the next balancing period, because then both the behind threads will run at full speed.

1 While there are fast slots and behind threads
2 Get next slow processor $\rho_s \in \mathcal{P}_{slow}$
3 Get next behind thread τ_{bh} on ρ_s
4 If no behind threads on ρ_s go to Line 2
5 Get next fast processor $\rho_f \in \mathcal{P}_{fast}$
6 Get next fast slot (occupied by the ahead thread τ_{ah}) on ρ_f
7 If no more fast slots on ρ_f go to Line 2
8 Set τ_{bh} to be migrated to ρ_f
9 Set τ_{ah} to be migrated to ρ_s

Figure 3: Pseudo code for redistribution of threads, executed by a single balancer.

Lines 8 and 9 in Figure 3 effectively swap the ahead and behind threads, requiring two migrations per fast slot (or per behind thread if there are fewer behind threads than fast slots). Although this may result in more than the minimum number of migrations, Juggle uses swaps because that guarantees that the imbalance can never exceed one (i.e., a processor will have either $\lceil n/m \rceil$ or $\lfloor n/m \rfloor$ threads). Consequently, errors in measurement cannot lead to imbalances greater than one, or any imbalance in the case of a perfect balance (e.g., 16 threads on 8 processors).

An off-by-one thread distribution may not be the best on multiprogrammed systems, but the best could be very hard to determine. For instance, if a high-priority external process is running on a processor, it may make sense to run fewer than $\lfloor n/m \rfloor$ threads on that processor, but what if the external process stops running partway through the balancing period? Swapping is a simple approach that works well in practice, even in multiprogrammed environments (see Section 4.4).

2.4 Modifications for NUMA

Using continual migrations to attain dynamic balance is reasonable only if the impact of migrations on locality is transient, as is the case with caches. However, on NUMA systems, accessing memory on a different NUMA domain is more expensive than accessing memory on the local domain, e.g., experiments with stream benchmarks on Intel Nehalem processors show that non-local memory bandwidth is about 2/3 of local access and latency is about 50% higher.

To address this issue, Juggle can be run with inter-domain migrations disabled. In this configuration each NUMA domain is balanced independently, i.e., all statistics, such as average thread progress, are computed per NUMA domain and a different balancer thread, one per domain, carries out classification and redistribution of the application threads within that domain. Furthermore, the initial distribution of application threads is carried out so that there is never more than one thread difference between domains. Our approach to load balancing on NUMA systems is similar to the way Linux partitions load balancing into domains defined by the

memory hierarchy. Juggle, however, does not implement domains based on cache levels; these often follow the NUMA domains anyway.

2.5 Scalability considerations

The complexity of the algorithm underlying Juggle is dominated by thread classification, and is $O(n)$. With one balancer per NUMA domain, the complexity is $O(zn/m)$, where z is the *size* of a NUMA domain (defined as the number of processors in the domain). Proactive load balancing is only useful when n/m is relatively small (less than 10 – see Section 4.1), so the scalability is limited by the size of the NUMA domains. In general, we expect that as systems grow to many processors, the number of NUMA domains will also increase, limiting the size of individual domains. If NUMA domains are large in future architectures, or if it is better to balance applications across all processors, then the complexity could be reduced to $O(n/m)$ by using all m balancers in a fully decentralized algorithm to classify the threads.

Although it would be possible to implement Juggle in a fully decentralized manner, as it stands it requires global synchronization (i.e., barriers), which is potentially a scalability bottleneck. Once again, synchronization is limited to each individual NUMA domain, so synchronization should not be an issue if the domains remain small. Even if synchronization is required across many processors, we expect the latency of the barrier operations to be on the order of microseconds; e.g., Nishtala et al. [13] implemented a barrier that takes 2μs to complete on a 32-core AMD Barcelona. The only other synchronization operation, locking, should not have a significant impact on scalability, since the only lock currently used is to protect per-processor data structures when threads are migrated.

3. ANALYSIS OF JUGGLE

We analyze an idealized version of the Juggle algorithm, making a number of simplifying assumptions that in practice do not significantly affect the predictive power of the theory. First, we assume that the required information about the execution statistics and the state of the application threads (e.g., if a thread is blocked) is available and precise. Second, we assume that any overheads are negligible, i.e., we ignore the cost of collecting and processing thread information, the overhead of the algorithm execution, and the cost of migrating a thread from one processor to another. Finally, we assume that the OS scheduling on a single processor is perfectly fair, i.e., if h identical threads run on a processor for Δt time, then each of those threads will make progress equal to $\Delta t/h$, even if Δt is very small (infinitesimal).

Consider a parallel application with n identical threads, $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$, running on m homogeneous processors, $\mathcal{P} = \{\rho_1, \ldots, \rho_m\}$, where $n > m$ and $n \bmod m \neq 0$ (i.e., there is an off-by-one imbalance). Initially, all the threads are distributed uniformly among the processors, which can consequently be divided into a set of slow processors, $\mathcal{P}_{slow}$, of size $|\mathcal{P}_{slow}| = n \bmod m$, and a set of fast processors, $\mathcal{P}_{fast}$, of size $|\mathcal{P}_{fast}| = m - (n \bmod m)$. Each processor in $\mathcal{P}_{slow}$ will run $\lceil n/m \rceil$ threads and each processor in $\mathcal{P}_{fast}$, will run $\lfloor n/m \rfloor$ threads. The set of slow processors provides $n_{slow} = (n \bmod m) \times \lceil n/m \rceil$ slow slots and the set of fast processors provides $n_{fast} = n - n_{slow}$ fast slots.

We assume that the threads in $\mathcal{T}$ are all initiated simultaneously at time t_0, which marks the beginning of a computation phase Φ. Once a thread τ_i completes its computation phase, it blocks on a barrier until the rest of the threads complete. We assume that it takes e units of time for a thread to complete its computation phase when running on a dedicated processor, and that when it blocks it consumes no processor cycles. Hence we can ignore all blocked threads. We say that the thread set $\mathcal{T}$ completes (phase Φ finishes) when the last thread in $\mathcal{T}$ completes and hits the barrier at t_f. Then, $CT_{\mathcal{T}} = t_f - t_0$ is the completion time of the thread set $\mathcal{T}$.

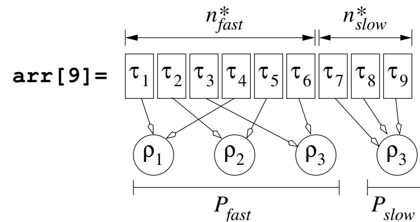

Figure 4: Example of *cyclic* distribution of $n^* = 9$ non-blocked threads among $m = 4$ processors. The first $n^*_{fast} = 6$ threads in the array are distributed cyclically among $|\mathcal{P}_{fast}| = 3$ processors and the last $n^*_{slow} = 3$ threads end up assigned to the single processor in $\mathcal{P}_{slow}$.

The load-balancing algorithm executes periodically every λ time units (the load-balancing period). To simplfy the analysis, we assume that the algorithm sorts the $n^* \leq n$ non-blocked threads in $\mathcal{T}$ in increasing order of progress, and then assigns the first n^*_{fast} threads to processors in $\mathcal{P}_{fast}$ in a cyclic manner, and the remaining n^*_{slow} threads to processors in $\mathcal{P}_{slow}$, also in a cyclic manner (see Figure 4).

Our analysis focuses on deriving lower bounds and approximations for the completion time $CT_{\mathcal{T}}$ of a thread set $\mathcal{T}$, when balanced by an ideal proactive load balancer. We split our analysis into two parts: the execution of a *single* computation phase Φ (Section 3.1), and the execution of a *sequence* of Φ (Section 3.2). Furthermore, for the purposes of comparison, we also provide analysis of $CT_{\mathcal{T}}$ for ideal reactive load balancing. Our theory helps users determine if proactive load balancing is likely to be beneficial for their applications compared to static or reactive load balancing. In our experience, SPMD parallel programs often exhibit completion times that are close to the theoretical predictions (see Section 4.

In the worst case, proactive load balancing is theoretically equivalent to static load balancing, because we assume neglible overheads. The completion time for static load balancing can be derived by noting that threads are distributed evenly among the processors before starting and never rebalanced. Consequently, each thread runs on its initially assigned processor until completion and the completion time of the thread set $\mathcal{T}$ is determined by the progress of the slowest threads; thus

$$CT^{static}_{\mathcal{T}} = e \times \lceil n/m \rceil \tag{1}$$

3.1 Single computation phase

We consider only the case where $\lambda < e \times \lceil \frac{n}{m} \rceil$. Above this limit load balancing will never execute before the thread set completes because even the slowest threads will take no more than $e \times \lceil \frac{n}{m} \rceil$ time to complete.

To determine a lower bound for $CT_{\mathcal{T}}$, the completion

time of the thread set $\mathcal{T}$, we compare the rate of progress of threads in $\mathcal{T}$ to an *imaginary thread* τ_{imag} that makes progress at a rate equal to $1/\lfloor n/m \rfloor$ and completes in $e\lfloor n/m \rfloor$ time. At the next load balancing point after $t = t_0 + e\lfloor n/m \rfloor$ (i.e., when τ_{imag} would have finished), one or more threads in $\mathcal{T}$ will lag τ_{imag} in progress. In Theorem 1 we show that at any load-balancing point that progress lag $\Delta P_{imag} \geq \lambda/(\lceil n/m \rceil \lfloor n/m \rfloor)$. If we assume that every thread that lags the imaginary thread τ_{imag} completes its execution on a dedicated processor, then a lower bound for the completion time of the thread set $\mathcal{T}$ is:

$$CT_{\mathcal{T}} \geq e\lfloor n/m \rfloor + \lambda/(\lceil n/m \rceil \lfloor n/m \rfloor) \tag{2}$$

In order to prove Theorem 1, we first focus on the progress lag between each pair of threads at any load-balancing point.

Without loss of generality, in the following lemma and theorem we assume that the threads in $\mathcal{T}$ are all initiated simultaneously at time $t = 0$ (i.e., Φ starts at $t = 0$). In addition, we assume that no thread in $\mathcal{T}$ completes before the load-balancing points under consideration (i.e., both n and m remain constant).

LEMMA 1. *Let $\Delta P(k\lambda)$ be the difference in progress between any pair of threads in $\mathcal{T}$. At any load-balancing point at time $t = k\lambda$ where $k \in \mathbb{N} \cup \{0\}$, $\Delta P(k\lambda)$ is either 0 or $\lambda/(\lceil n/m \rceil \lfloor n/m \rfloor)$.*

PROOF. The proof is by induction. Initially, at $t = 0$ (when $k = 0$) each thread $\tau_i \in \mathcal{T}$ has zero progress; thus, $\Delta P(0) = 0$.

Next we consider the difference in progress among the threads in $\mathcal{T}$ at the end of the first load-balancing period. At $t = \lambda$, the n_{slow} threads in slow slots will have made $\lambda/\lceil n/m \rceil$ progress and the n_{fast} threads in fast slots will have made $\lambda/\lfloor n/m \rfloor$ progress, which means that the progress lag at that point in time is:

$$\Delta P(\lambda) = \frac{\lambda}{\lfloor n/m \rfloor} - \frac{\lambda}{\lceil n/m \rceil} = \frac{\lambda}{\lceil n/m \rceil \lfloor n/m \rfloor} \tag{3}$$

This result holds for any $n_{slow} > 0$ and $n_{fast} > 0$, provided $n > m$ (which is one of our fundamental assumptions).

Equation (3) suggests that threads can be grouped according to their progress into i) a set $\mathcal{T}_{ah}$ of *ahead threads* of size n_{ah} and ii) a set $\mathcal{T}_{bh}$ of *behind threads* of size n_{bh}. Although n_{slow} and n_{fast} remain fixed because n and m are assumed to be constant, n_{ah} and n_{bh} can vary at each load-balancing point.

We now assume that at the kth balancing point $\Delta P(k\lambda) = \lambda/(\lceil n/m \rceil \lfloor n/m \rfloor)$ and show that at the next balancing point $\Delta P((k+1)\lambda)$ is either $\lambda/(\lceil n/m \rceil \lfloor n/m \rfloor)$ or 0. We consider all the possible scenarios in terms of the relations among n_{fast}, n_{slow}, n_{ah}, and n_{bh}. These scenarios can be grouped into three general cases that share identical analyzes.

Case A: $n_{bh} < n_{fast}$. All the behind threads can run on fast slots, and so form a group $G_{bh \to fast}$, which progresses at $1/\lfloor n/m \rfloor$. The left-over fast slots are filled by a fraction of the ahead threads, $G_{ah \to fast}$, which also progress at $1/\lfloor n/m \rfloor$. The remaining ahead threads must run on slow slots, forming a group $G_{ah \to slow}$ that progresses at $1/\lceil n/m \rceil$.

The progress that a thread in each of these groups achieves by the next balancing point is then:

$$P_{G_{bh \to fast}}((k+1)\lambda) = P_{\mathcal{T}_{bh}}(k\lambda) + \lambda/\lfloor n/m \rfloor \tag{4}$$
$$P_{G_{ah \to slow}}((k+1)\lambda) = P_{\mathcal{T}_{ah}}(k\lambda) + \lambda/\lceil n/m \rceil \tag{5}$$
$$P_{G_{ah \to fast}}((k+1)\lambda) = P_{\mathcal{T}_{ah}}(k\lambda) + \lambda/\lfloor n/m \rfloor \tag{6}$$

By substituting $P_{\mathcal{T}_{ah}}(k\lambda)$ for $P_{\mathcal{T}_{bh}}(k\lambda) + \lambda/(\lceil n/m \rceil \lfloor n/m \rfloor)$ in Equation (5), we can show that $P_{G_{bh \to fast}}((k+1)\lambda) = P_{G_{ah \to slow}}((k+1)\lambda)$. At $t = (k+1)\lambda$, $\mathcal{T}_{ah} = G_{ah \to fast}$ and $\mathcal{T}_{bh} = G_{bh \to fast} \cup G_{ah \to slow}$. Moreover, by subtracting Equation (6) from Equation (5) we obtain:

$$\Delta P((k+1)\lambda) = \lambda/(\lceil n/m \rceil \lfloor n/m \rfloor) \tag{7}$$

Case B: $n_{bh} > n_{fast}$. All the ahead threads run on slow slots, forming a group $G_{ah \to slow}$, which progresses at $1/\lceil n/m \rceil$. A fraction of the behind threads, $G_{bh \to slow}$, will also run on slow slots and progress at $1/\lceil n/m \rceil$. The remainder of the behind threads, $G_{bh \to fast}$, run on fast slots and progress at $1/\lfloor n/m \rfloor$.

The progress that a thread in each of these groups achieves by the next balancing point is then:

$$P_{G_{ah \to slow}}((k+1)\lambda) = P_{\mathcal{T}_{ah}}(k\lambda) + \lambda/\lceil n/m \rceil \tag{8}$$
$$P_{G_{bh \to fast}}((k+1)\lambda) = P_{\mathcal{T}_{bh}}(k\lambda) + \lambda/\lfloor n/m \rfloor \tag{9}$$
$$P_{G_{bh \to slow}}((k+1)\lambda) = P_{\mathcal{T}_{bh}}(k\lambda) + \lambda/\lceil n/m \rceil \tag{10}$$

By substituting $P_{\mathcal{T}_{ah}}(k\lambda)$ for $P_{\mathcal{T}_{bh}}(k\lambda) + \lambda/(\lceil n/m \rceil \lfloor n/m \rfloor)$ in Equation (8), we can show that $P_{G_{ah \to slow}}((k+1)\lambda) = P_{G_{bh \to fast}}((k+1)\lambda)$. At $t = (k+1)\lambda$, $\mathcal{T}_{ah} = G_{ah \to slow} \cup G_{bh \to fast}$ and $\mathcal{T}_{bh} = G_{bh \to slow}$. Moreover, by subtracting Equation (10) from Equation (9) we obtain:

$$\Delta P((k+1)\lambda) = \lambda/(\lceil n/m \rceil \lfloor n/m \rfloor) \tag{11}$$

Case C: $n_{bh} = n_{fast}$. All the ahead threads run on slow slots, forming a group, $G_{ah \to slow}$, that progresses at $1/\lceil n/m \rceil$, and all the behind threads run on fast slots, forming a group, $G_{bh \to fast}$, that progresses at $1/\lfloor n/m \rfloor$.

The progress that a thread in each of these groups achieves by the next balancing point is then:

$$P_{G_{ah \to slow}}((k+1)\lambda) = P_{\mathcal{T}_{ah}}(k\lambda) + \lambda/\lceil n/m \rceil \tag{12}$$
$$P_{G_{bh \to fast}}((k+1)\lambda) = P_{\mathcal{T}_{bh}}(k\lambda) + \lambda/\lfloor n/m \rfloor \tag{13}$$

By substituting $P_{\mathcal{T}_{ah}}(k\lambda)$ for $P_{\mathcal{T}_{bh}}(k\lambda) + \lambda/(\lceil n/m \rceil \lfloor n/m \rfloor)$ in Equation (12), we can show that $P_{G_{ah \to slow}}((k+1)\lambda) = P_{G_{bh \to fast}}((k+1)\lambda)$. Consequently, $\Delta P((k+1)\lambda) = 0$. This means that at $t = (k+1)\lambda$ the threads will be in a situation similar to that we first analyzed when $t = 0$. □

THEOREM 1. *Let $\Delta P_{imag}(t)$ be the progress lag at time t between some thread in $\mathcal{T}$ and an imaginary thread τ_{imag} that always progresses at a rate of $1/\lfloor n/m \rfloor$. At any load-balancing point at time $t = k\lambda$ where $k \in \mathbb{N}^+$*

$$\Delta P_{imag}(k\lambda) \geq \Delta P = \lambda/(\lceil n/m \rceil \lfloor n/m \rfloor)$$

PROOF. At the end of the first load-balancing period, when $t = \lambda$, the ahead threads will have made the same progress as τ_{imag}, and hence the behind threads will lag τ_{imag} by ΔP. As proved in Lemma 1, if $n_{bh} < n_{fast}$ or $n_{bh} > n_{fast}$ at this or any subsequent balancing point at $t = k\lambda$ with $k = 1, 2, 3, \ldots$, then by the next balancing point at $t = (k+1)\lambda$ the difference in progress between the ahead threads ($\in \mathcal{T}_{ah}$) and the behind threads ($\in \mathcal{T}_{bh}$) is

ΔP. Hence, at $t = (k+1)\lambda$ the threads in $\mathcal{T}_{bh}$ will lag τ_{imag} by *at least* ΔP. Note that the ahead threads may have made progress at a rate of $1/\lceil n/m \rceil$ (i.e., slow speed) in some previous load-balancing period.

Now consider the case in which $n_{bh} = n_{fast}$ at a load-balancing point at $t = k\lambda$ with $k \in \mathbb{N}^+$. Here the threads in $\mathcal{T}_{ah}$ progress at a rate of $1/\lceil n/m \rceil$ (i.e., slow speed) during the next λ time units, while the threads in $\mathcal{T}_{bh}$ progress at a rate of $1/\lfloor n/m \rfloor$ (i.e., fast speed). In Lemma 1 we proved that in this case all the threads in $\mathcal{T}$ have made the same progress by the next load-balancing point at $t = (k+1)\lambda$. If we assume that by $t = k\lambda$ the threads in $\mathcal{T}_{ah}$ and τ_{imag} have the same progress, it is easy to see that at $t = (k+1)\lambda$ the threads in $\mathcal{T}_{ah}$ will fall behind τ_{imag} by at least ΔP. Again note that the difference in progress between τ_{imag} and the threads in $\mathcal{T}_{ah}$ can be greater because the ahead threads may have run at slow speed in some previous load-balancing period. □

In practice, Equation (2) is a good predictor of performance when $e \approx \lambda$. However, when $\lambda \ll e$, Equation (2) degenerates to a bound where every thread makes progress at a rate of $1/\lfloor n/m \rfloor$ (i.e., like τ_{imag}). To refine this bound, we consider what happens when λ is infinitisemal, i.e., load balancing is continuous and perfect. Given our assumption that on a single processor each thread gets exactly the same share of processor time, this scenario is equivalent to that in which all threads execute on a single processor ρ^*, which is m times faster than each processor in $\mathcal{P}$. Hence each thread makes progress on ρ^* at a rate of $(n/m)^{-1}$ and the completion time of the thread set $\mathcal{T}$ is given by

$$CT_{\mathcal{T}}^{\lambda \rightsquigarrow 0} = e \times n/m \tag{14}$$

This expression also yields a *lower bound* for $CT_{\mathcal{T}}$. When λ is *not* infinitesimal, some threads in $\mathcal{T}$ make progress between balancing points at a rate equal to $\lceil n/m \rceil^{-1}$, where $\lceil n/m \rceil^{-1} < (n/m)^{-1}$, given our assumptions that $n > m$ and $n \bmod m \neq 0$. Therefore, some threads fall behind when compared with the threads running on ρ^* and delay the completion of the entire thread set. Because of the progress lag, a thread set $\mathcal{T}$ that is periodically balanced among processors in $\mathcal{P}$ could have a completion time significantly more (and never less) than the completion time of $\mathcal{T}$ running on ρ^*.

3.2 Multiple computation phases

Generally, SPMD parallel applications have multiple computation phases. We analyze this case by assuming that we have a parallel application where every thread in the set $\mathcal{T}$ sequentially executes exactly the same computation phase Φ multiple times. We assume that all threads synchronize on a barrier when completing the phase and that the execution of Φ starts again immediately after the thread set has completed the previous phase (e.g., see Figure 5). The sequence of executions of Φ, denoted as $\mathcal{S}_\Phi$, is finite and $l_{\mathcal{S}_\Phi} \in \mathbb{N}^+$ denotes the number of executions of Φ in $\mathcal{S}_\Phi$. We are interested in lower bounds and approximations for the completion time, $CT_{\mathcal{S}_\Phi}$, of the entire sequence.

We can derive a simple lower bound for $CT_{\mathcal{S}_\Phi}$ by assuming that in each execution of Φ the threads in $\mathcal{T}$ are continuously balanced. Thus, using Equation (14), we get

$$CT_{\mathcal{S}_\Phi} \geq l_{\mathcal{S}_\Phi} \times e \times \frac{n}{m} \tag{15}$$

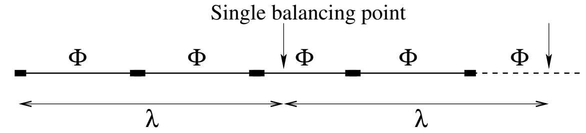

Figure 5: A sequence of executions of a single computation phase Φ with the load-balancing period λ extending over multiple executions.

This bound is reasonably tight when $\lambda \ll e$. Although we have derived bounds for $\lambda \approx e$ and $\lambda \geq e \times \lceil \frac{n}{m} \rceil$, they are somewhat loose and we omit their derivations due to space constraints. Instead, we present approximations to the completion times that work well in practice.

In the case where $\lambda < e \times \lceil \frac{n}{m} \rceil$, we use the lower bound derived in Equation (2) to approximate $CT_{\mathcal{S}_\Phi}$ as:

$$CT_{\mathcal{S}_\Phi} \approx l_{\mathcal{S}_\Phi} \times CT_{\mathcal{T}} \tag{16}$$

When $\lambda \geq e \times \lceil \frac{n}{m} \rceil$, some executions of Φ in $\mathcal{S}_\Phi$ will contain a single balancing point, while the others will not be balanced, provided that $\lambda < l_{\mathcal{S}_\Phi} \times e \times \lceil n/m \rceil$ and $(l_{\mathcal{S}_\Phi} \times e \times \lceil n/m \rceil) \bmod \lambda \neq 0$. In the worst case, if load balancing is completely ineffective, the completion time of the entire sequence will be $CT_{\mathcal{S}_\Phi} = l_{\mathcal{S}_\Phi} \times e \times \lceil n/m \rceil$. Consequently, the maximum number of executions of Φ in $\mathcal{S}_\Phi$ that can contain a single load-balancing point is $\eta = \lfloor l_{\mathcal{S}_\Phi} \times e \times \lceil n/m \rceil / \lambda \rfloor$. If we can compute the expected completion time, $\overline{CT}_{\mathcal{T}}$, for a single execution phase balanced only once during the computation, then we can approximate the completion time across all phases as:

$$CT_{\mathcal{S}_\Phi} \approx (l_{\mathcal{S}_\Phi} - \eta) \times e \times \lceil n/m \rceil + \eta \times \overline{CT}_{\mathcal{T}}^* \tag{17}$$

Consider a load balancing point that occurs after some fraction q/k of the computation phase Φ has elapsed (for some value of k). The completion time of threads that start slow and become fast after balancing is $e \times (q/k\lceil n/m \rceil + (1-q/k)\lfloor n/m \rfloor)$ whereas the completion time of threads that start fast and become slow is $e \times (q/k\lfloor n/m \rfloor + (1-q/k)\lceil n/m \rceil)$. The completion time of the thread set $\mathcal{T}$ is then the longest completion time of any thread, so

$$\begin{aligned} CT_{\mathcal{T}}^q = e \times max(&q/k\lceil n/m \rceil + (1-q/k)\lfloor n/m \rfloor, \\ &q/k\lfloor n/m \rfloor + (1-q/k)\lceil n/m \rceil) \end{aligned}$$

We assume that the load balancing point is equally likely to fall at any one of a discrete set of fractional points, $1/k$, $2/k, \ldots, (k-1)/k$ during the execution of Φ. We can then estimate the *expected* completion time of the thread set by calculating the average of the completion times at all these points as k becomes large. Because we are taking the maximum, the completion times are symmetric about $k/2$, i.e., $CT_{\mathcal{T}}^{q/k} = CT_{\mathcal{T}}^{1-q/k}$. Hence we compute the average over the first half of the interval

$$\overline{CT_{\mathcal{T}}} = \frac{2}{k} \sum_{q=1}^{k/2} CT_{\mathcal{T}}^q \approx (\lceil n/m \rceil - 1/4)$$

since $k/2 + 1 \approx k/2$ for large k.

Note that if $n_{fast} < n_{slow}$, a single load-balancing point in the execution of Φ cannot reduce the completion time of $\mathcal{T}$ when compared to static load-balancing (see Equation (1)). It is not possible to make *all* the slow threads run at fast

speed at the single balancing point, so some threads will run at slow speed during the *entire* execution of Φ. Therefore, when $n_{fast} < n_{slow}$

$$CT_{\mathcal{S}_\Phi} = l_{\mathcal{S}_\Phi} \times e \times \lceil n/m \rceil \quad (18)$$

3.3 Reactive load balancing

We analyze the case for ideal reactive load balancing, where threads are redistributed immediately some of them block and the load balancer incurs no overheads. We only provide results for $1 < n/m < 2$. To derive the completion time for reactive balancing, we note that as soon as fast threads block, the remaining threads are rebalanced and some of them become fast. Every time load balancing occurs, the slow threads have run for half as long as the fast threads, so the completion time for the thread set is

$$CT_{\mathcal{T}}^{react} = \sum_{i=0}^{k} \frac{1}{2^i} = 2 - 2^{-k} \quad (19)$$

where k is the number of times the load balancer is called. To determine k, we observe that the number of fast threads before the first balancing point is

$$n_f(0) = \lfloor n/m \rfloor (n \bmod m) = \lfloor n/m \rfloor (m\lceil n/m \rceil - n) = 2m - n$$

since $\lceil n/m \rceil = 2$. At every balance point, all the currently fast threads block which means the number of fast slots available doubles. Consequently, at any balance point, i, the number of fast threads is

$$n_f(i) = (2m - n) \times 2^i \quad (20)$$

All threads will complete at the k-th balance point when $n_f(k) \leq m$ because then all unblocked threads will be able to run fast. So we solve for k using Equation (20)

$$(2m - n) \times 2^k \geq m \implies k = -\lfloor \log_2(2 - n/m) \rfloor$$

Substituting k back into Equation (19) gives

$$CT_{\mathcal{T}}^{react} = 2 - 2^{\lfloor \log_2(2-n/m) \rfloor} \quad (21)$$

4. EMPIRICAL EVALUATION

In this section we investigate the performance of Juggle through a series of experiments with a variety of programming models (pthreads, UPC and OpenMP) using microbenchmarks and the NAS parallel benchmarks.[3] We show that for many cases, the lower bounds derived in Section 3 are useful predictors of performance, and demonstrate that actual performance of Juggle closely follows a simulation of the idealized algorithm. We also explore the effects of various architectural features, such as NUMA and hyper-threading. Finally, we show that the overhead of Juggle is negligible at concurrencies up to 16 processing elements.

All experiments were carried out on Linux systems, running 2.6.30 or newer kernels. The `sched_yield` system call was configured to be POSIX-compliant by writing 1 to the `proc` file `sched_compat_yield`. Hence, a thread that spins on `yield` will consume almost no processor time if it is sharing with a non-yielding thread.

Whenever we report a *speedup*, it is relative to the statically balanced case with the same thread and processor configuration, i.e., $RelativeSpeedup = CT_{static}/CT_{dynamic}$. Completion times for statically balanced cases are given by Equations (1) and (18). Moreover, we define the *upper bound* for the relative speedup as: $UB(RelativeSpeedup) = CT_{static}/LB(CT_{dynamic})$, where LB denotes the theoretical lower bound. For static balancing, we pin the threads as uniformly as possible over the available processing elements using the `sched_setaffinity` system call. Thus the imbalance is at most one. Every reported result is the average of ten runs; we do not report variations because they are small.

Some experiments compare the performance of Juggle with the default Linux Load Balancer (LLB). Although LLB does not migrate threads to correct off-by-one imbalances in active run queues, it dynamically rebalances applications that sleep when blocked, because a thread that sleeps is moved off the active run queue, increasing the apparent imbalance. We view LLB as an example of *reactive* load balancing: threads are only migrated when they block. To ensure a fair comparison between LLB and Juggle, when testing LLB we start with each application thread pinned to a core so that the imbalance is off-by-one. We then let the LLB begin balancing from this initial configuration by unpinning all the threads.

4.1 Ideal relative speedups

We tested the case for the ideal relative speedup using a compute-intensive pthreads microbenchmark, μbench, that uses no memory and does not perform I/O operations, and scales perfectly because each thread does exactly the same amount of work in each phase. As shown in Figure 6, the theory derived in Section 3.1 (Equation (14)) for the ideal case closely predicts the empirical performance when $\lambda \ll e$. Figure 6 presents the results of running μbench on an 8-core Intel *Nehalem*[4] with hyper-threading disabled. This is a two-domain NUMA system, but even with inter-domain migrations enabled, the relative speedup is close to the theoretical ideal for 8 processing elements, because μbench does not use memory. When we restrict inter-domain migrations, the performance is close to the theoretical ideal for two NUMA domains of four processing elements each.

Preventing inter-domain migration limits the effectiveness of load balancing. For example, 8 threads on 7 processors will have an ideal completion time equal to $(8/7) \times e \approx 1.143 \times e$, but split over two domains, one of them will have 4 threads on 3 processors, for an ideal completion time of $(4/3) \times e \approx 1.333 \times e$. This issue is explored more fully in Section 4.3.

Figure 6 also shows a theoretical upper bound for the relative speedup when using reactive load balancing (Equation (21)). The closeness of the results using LLB and the theory for reactive load balancing shows that LLB is a good implementation of reactive load balancing when $\lambda \ll e$. Note, however, that LLB is actually periodic; consequently balancing does not happen immediately when a thread blocks on a barrier, so the performance of LLB degrades as e decreases (data not shown).

Figure 7 shows that the advantage of proactive load balancing over static balancing decreases as n/m increases because the static imbalance decreases. If an application exhibits strong-scaling, then increasing the level of oversubscription (i.e., increasing n relative to m) should reduce

[3]UPC 2.9.3 with NAS 2.4, OMP Intel 11.0 Fortran with NAS 3.3, available at `http://www.nas.nasa.gov/Resources/Software/npb.html`

[4]Two sockets (NUMA domains) of Xeon E5530 2.4GHz Quad Core processors with two hyper-threads/core, 256K L2 cache/core, 8M L3 cache/socket and 3G memory/core.

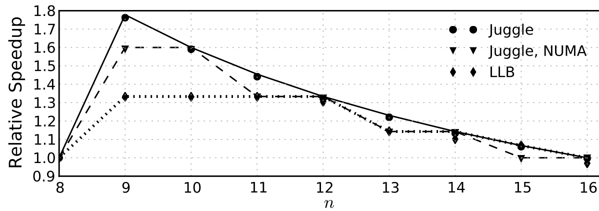

Figure 6: Relative speedup of μbench with Juggle and LLB. The configuration parameters are $n \in [8, 16]$, $m = 8$, $e = 30s$ and $\lambda = 100ms$. Markers are empirical results and lines are theoretical results.

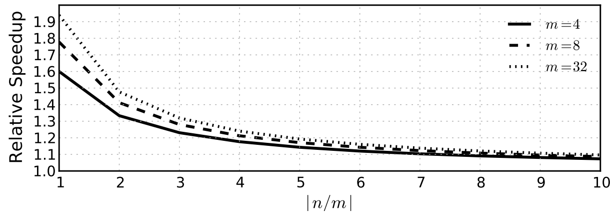

Figure 7: Ideal relative speedup of proactive load balancing for increasing $\lfloor n/m \rfloor$; n is chosen to give the most imbalanced static thread distribution (e.g., $n = 9$ when $m = 8$, $n = 33$ when $m = 32$, etc.).

the completion time even without proactive load balancing. The reason is that in this case oversubscription reduces the amount of work done by each thread and hence the off-by-one imbalance has less impact. However, high oversubscription levels can reduce the performance of strong-scaling applications [8]; consequently, oversubscription cannot be regarded as a general solution for load imbalances.

We illustrate this fact in Table 1, which shows the results of oversubscription (with static balancing) and proactive load balancing on the UPC NAS parallel benchmark, FT, class C, running on the 8-core Intel *Nehalem* with hyper-threading disabled. In this experiment, inter-domain migrations were disabled for proactive load balancing. When the number of threads increases from 16 to 32 on 8 processing elements, the completion time for the perfectly balanced case increases by 19%. Furthermore, when the oversubscription level is low enough not to have a negative impact, the use of proactive balancing improves the performance of the benchmark significantly (i.e., 43% for $n = 8$ and $m = 7$ and 21% for $n = 16$ and $m = 7$). In the rest of our experiments, we focus on cases where $\lfloor n/m \rfloor \leq 2$.

4.2 Effect of varying λ and e

We tested the effects of varying the load-balancing period, λ, using the UPC NAS benchmark EP, class C, on

n	m	Static LB	$e\lceil n/m \rceil$	Juggle
8	8	68	68	–
16	8	68	68	–
32	8	81	68	–
8	7	120	136	84
16	7	92	102	76
32	7	87	85	87

Table 1: Effects of oversubscription on the completion time, in seconds, of UPC FT, class C.

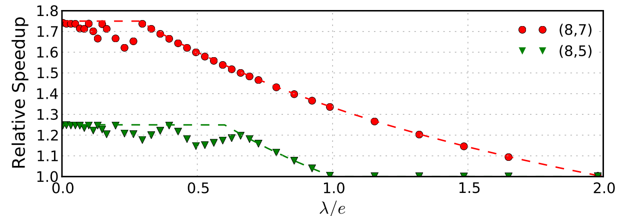

Figure 8: Relative speedup of UPC EP class C with Juggle; $e = 30s$, and λ varies. Threads and processing elements (i.e., cores) are denoted by (n, m). Dotted lines are the upper bounds for the relative speedup and markers are empirical results.

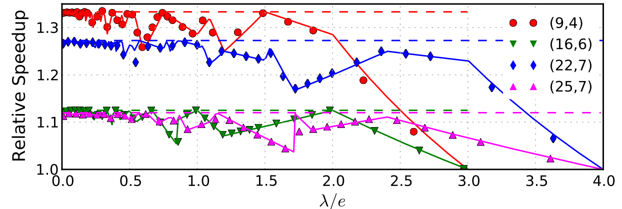

Figure 9: Relative speedup of UPC EP class C with Juggle; $e = 30s$ and λ varies. Threads and processing elements are denoted by (n, m). Dotted lines are the upper bounds, solid lines are the simulation of the ideal algorithm, and markers are empirical results.

the 8-core *Nehalem* with hyper-threading disabled. EP has a single phase, with $e = 30s$ on the *Nehalem*, and uses no memory, so we can ignore the impact of NUMA and balance fully across all 8 cores. Figure 8 shows that the empirical results obtained using Juggle closely follow the upper bounds for the relative speedup when $\lfloor n/m \rfloor = 1$. The figure also indicates how proactive load balancing becomes ineffective as λ increases relative to e.

In addition to the cases shown in Figure 8, we have tested Juggle on a variety of configurations up to $\lfloor n/m \rfloor = 4$. A selection of these results is shown in Figure 9. We can see that the upper bound for the relative speedup (the dotted line) is looser when $\lfloor n/m \rfloor > 1$. Figure 9 also presents the result of a simulation of the idealized algorithm (the solid line). We can see that this very closely follows the empirical results, which implies that our practical decentralized implementation of Juggle faithfully follows the principles of the idealized algorithm. The simulation also enables us to visualize a large variety of configurations, as shown in Figure 10.

To explore the effect of multiple computation phases, we modified EP to use a configurable number of barriers, with fixed phase sizes. Figure 11 shows that the empirical behavior closely matches the approximations (the dashed line) given by Equation (18) when $\lambda \geq e\lfloor n/m \rfloor$, and by Equation (16) when $\lambda < e\lfloor n/m \rfloor$

An interesting feature in Figure 11 is that when λ/e is a multiple of 2 (i.e., $\lceil n/m \rceil$), there is no relative speedup for the experimental runs, because the balancing point falls almost exactly at the end of a phase. If we add a small random variation ($\pm 10\%$) to λ, this feature disappears. In general, adding randomness should not be necessary, because it is unlikely that phases will be quite so regular and that there will

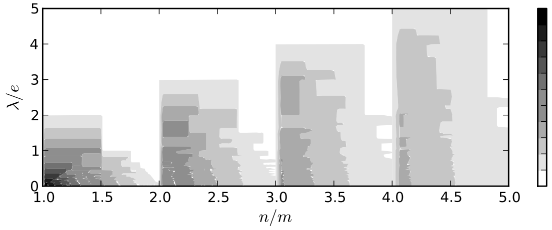

Figure 10: Relative speedup of the simulated idealized algorithm. Each point in the heat map is colored according to relative speedup over the statically balanced case. $n = 3, 4, \ldots, 64$, $m = 2, 3, \ldots, n-1$ **and** $\lambda/e = 0.01, 0.02, \ldots, 5$.

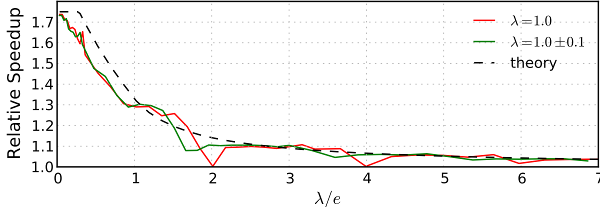

Figure 11: Relative speedup of UPC EP class C (multiple barriers) with Juggle; $n = 8$ **and** $m = 7$**. Continuous lines are the empirical results and the dashed line is the relative speedup approximation.**

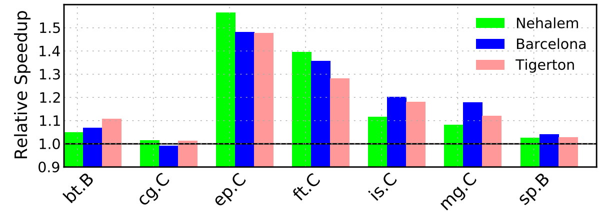

Figure 12: Relative speedup of UPC NAS benchmarks with Juggle; $n = 16$, $m = 12$ **and** $\lambda = 100ms$**.**

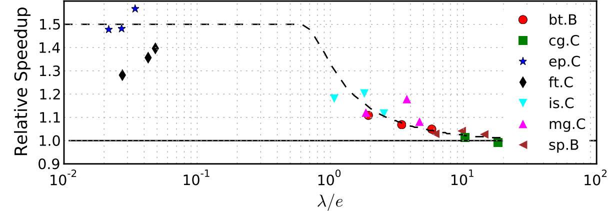

Figure 13: The effect of λ/e **on the relative speedup of UPC NAS benchmarks balanced by Juggle;** $n = 16$, $m = 12$ **and** $\lambda = 100ms$**. The dashed line is the relative speedup approximation and markers are empirical results from three systems, *Barcelona*, *Tigerton*, and *Nehalem*.**

be such perfect correspondence between the phase length e and the balancing period λ.

4.3 NAS benchmarks

We explored the effects of memory, caching, and various architectural features such as NUMA and hyper-threading, through a series of experiments with the NAS benchmarks on three different architectures: in addition to the *Nehalem* already described, we used a 16-core AMD *Barcelona*[5] and a 16-core Intel *Tigerton*.[6] These systems represent three important architectural paradigms: the *Nehalem* and *Barcelona* are NUMA systems, the *Tigerton* is UMA, and the *Nehalem* is hyper-threaded.

To give a reasonable running time, we chose class C for most benchmarks and class B for BT and SP, which take longer to complete. All the experiments were carried out with $n = 16$ and $m = 12$. We selected the 12 processing elements uniformly across the NUMA domains, so for the *Barcelona* we used three cores per domain, and for the *Nehalem* we used 6 hyper-threads per domain. Although we disabled inter-domain migrations on the *Barcelona* and *Nehalem*, we expect the same ideal relative speedup across all systems, $2/(4/3) = 2/(8/6) = 2/(16/12) = 1.5$. Furthermore, with $n = 16$ and $m = 12$ the relative speedup should be the same for reactive and proactive load-balancing (see Figure 6).

In Figure 12 we can see that EP gets close to the ideal relative speedup on the *Barcelona* and the *Tigerton*, but actually *better* (1.57) than the ideal relative speedup on the *Nehalem*. This is attributable to hyper-threading: when there is one application thread per hyper-thread, and it blocks (sleep or yield), any other threads on the paired hyper-thread will go 35% faster (for this benchmark), which breaks the assumption of homogeneous processing elements.

For the benchmarks which do not attain the ideal relative speedup (all except EP) we can determine how much of the slowdown is due to the value of λ/e. Recall that e denotes the running time of an execution phase for a thread running on a dedicated processing element. We approximate e by counting the number of `upc_barrier` calls in each benchmark and dividing that into the running time for 16 threads on 16 processing elements. Figure 13 shows that correlating the relative speedup with λ/e accounts for most of the deviation from the ideal relative speedup, because the empirical performance is close to that obtained from the approximations derived in Sections 3.1 and 3.2. FT deviates the most because it is the most memory intensive, and so we can expect migrations to have a larger impact.

In UPC, blocked threads do not sleep which means that the Linux Load Balancer (LLB) will not balance UPC applications. By contrast, in OpenMP blocked threads first yield for some time period, k, and then sleep. If k is small then OpenMP applications can to some extent be balanced by LLB. Figure 14 shows results of running the OpenMP NAS benchmarks on the *Barcelona* system[7] with $k = 200ms$ (the default) and $k = 0$, meaning that threads immediately sleep when they block. LLB has some beneficial effect on EP, giving a 35% relative speedup when $k = 0$ and a 30% relative speedup when $k = 200ms$. This is below the theoretical 50% maximum for reactive balancing that we expect with $n = 16$ and $m = 8$, which indicates that LLB is not balancing the threads immediately they block, or not balancing them correctly. The only other benchmark that benefits from LLB is FT, where we see a small (9%) relative speedup. By contrast, Juggle improves the performance of most benchmarks,

[5]Four sockets (NUMA domains) of Opteron 8350 2GHz Quad Core processors with 512K L2 cache/core, 2M L3 cache/socket and 4G memory/core.

[6]Four sockets of Xeon E7310 1.6GHz Quad Core processors with one 4M L2 cache and 2G memory per pair of cores.

[7]We observed similar results on *Tigerton* and *Nehalem*.

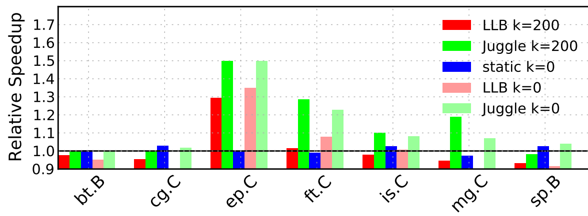

Figure 14: Relative speedup of OpenMP NAS benchmarks balanced by Juggle and LLB on the *Barcelona* system with $n = 16$, $m = 12$ and $\lambda = 100ms$. Relative speedup is measured against the statically balanced case with $k = 200ms$.

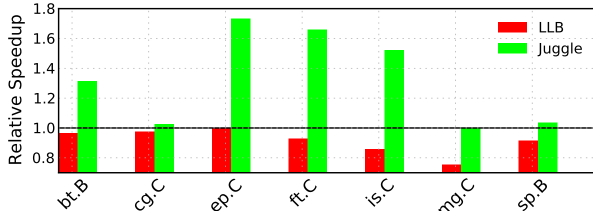

Figure 15: Relative speedup of UPC NAS benchmarks with Juggle and LLB in a highly multiprogrammed environment; $n = 8$ except for BT and SP where $n = 9$, $m = 8$, and $\lambda = 100ms$.

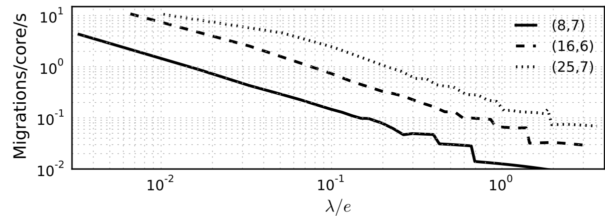

Figure 16: Migration rate for Juggle balancing UPC EP class C on the *Nehalem* system. Thread and core counts are given as (n, m). $e = 30s$ and λ varies.

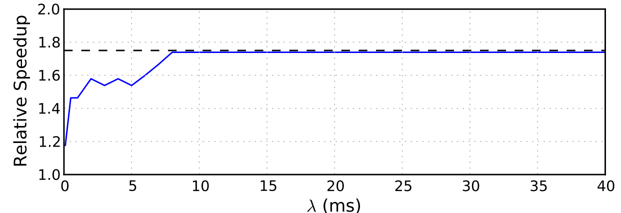

Figure 17: Effect of decreasing λ on EP class C, with $n = 8$, $m = 7$. The dashed line is the theoretical maximum relative speedup.

and the default synchronization with $k = 200ms$ performs slightly better than pure sleep. With Juggle, it makes no difference how blocking is implemented and yielding results in faster synchronization than sleep.

4.4 Multiprogrammed environments

Due to space limitations we cannot fully explore the issue of multiprogrammed environments. Instead we present the results of using Juggle in an unpredictable multiprogrammed environment that, although simple, is nevertheless challenging for load balancing in the SPMD model.

We ran the UPC NAS benchmarks on the *Nehalem* system with $m = 8$ and hyper-threading disabled. We used $n = 8$, except for BT and SP, where we used $n = 9$ because they require a square number of threads. While the benchmarks were running, we also ran two unrelated single-threaded processes that could represent system daemons or user-initiated processes. Each external process cycles continually between sleeping for some random time from 0 to $5s$, and computing for some random time from 5 to $10s$. We set one of the processes to have a higher priority (`nice` -3).

Figure 15 shows that Juggle enables the benchmarks to run efficiently even in this unpredictable environment. By contrast, LLB usually causes the benchmark to run *slower* than in the statically balanced case. Even though LLB cannot correct off-by-one imbalances in UPC applications, we expect that LLB should be at least as good as static balancing. The problem is that LLB schedules tasks without considering that some of them form a single, parallel application.

4.5 Evaluating the overhead

We measured the compute time taken by Juggle when balancing EP on the *Nehalem* system, with $n = 8$, $\lambda = 100ms$, and inter-domain migrations enabled. When $m = 8$ there are no migrations and the compute time for Juggle is about $20\mu s$ per load-balancing point, and when $m = 7$ there are on average 28 migrations per second and the compute time for Juggle is about $100\mu s$ per balancing point. Both of these translate into negligible effects on the running time of the benchmark, and since Juggle scales as $O(kn/m)$ (where k is the size of a NUMA domain), we expect the algorithm to have no scaling issues as long as NUMA domains do not get orders of magnitude larger than 8.

Figure 16 shows how the number of migrations scales as the ratio λ/e increases, and as n increases relative to m. The number of migrations is generally determined by the load-balancing period. For example, when $n = 8$ and $m = 7$, there is only one slow core, so there are at most two swaps (four migrations) per period, but sometimes there are no swaps, so the average is lower (3 per period on average). In addition, the cost of migrations is low. We measured the time taken by the `sched_setaffinity` system call as $8\mu s$ on the *Nehalem* system.

The theoretical analyzes of Section 3 indicate that the smaller the value of λ relative to e, the better. The analyzes assume that the cost of migrations is negligible, and that there are no other disadvantages to very small values of λ. In practice, however, when λ is on the order of the scale at which the OS scheduler operates, the assumption that each thread gets a fair share of a processing element breaks down. We can see this in Figure 17, which shows how performance degrades as λ falls below $10ms$, which is the scheduling interval on this particular system. As λ reduces even further, the $100\mu s$ overhead starts to impact performance, e.g., at $0.5ms$ we expect there to be a 20% decrease in performance due to the $100\mu s$ overhead. These limitations imply that our user-space implementation is not suitable for very fine-grained applications (very small e), because we cannot reduce λ sufficiently to balance effectively.

5. RELATED WORK

We have focused on approaches to overcoming extrinsic imbalances in SPMD applications that are a result of mismatches between processor and thread counts, or caused by the presence of external processes in multiprogrammed environments. We do not address the issue of intrinsic imbalance. For the latter many different approaches have been proposed [5, 16], from programming-model extensions (such as work stealing [1, 11, 14]) to hardware prioritization on hyper-threaded architectures [2].

Our interest is in load balancing for the off-by-one imbalance problem and we assume that we can always start with at most an off-by-one imbalance. Much research [18, 19], on the other hand, has focused on getting from larger imbalances to off-by-one, usually for distributed memory systems. Most often these load balancers are themselves distributed and avoid global state (e.g., nearest neighbor strategies [10]), because of the overheads associated with distributed memory. Moreover, correctness and efficiency of the distributed load-balancing algorithms are usually the research focus (e.g., proving that an algorithm converges to the off-by-one state in a reasonable amount of time [3]).

An approach that does address off-by-one imbalances for SPMD applications is *Speed Balancing* [7]. This approach implements a decentralized user-space balancer that continually migrates threads with the goal of ensuring that all threads run at the same "speed" (or make the same progress). Speed Balancing is thus a form of proactive load balancing. Although it uses some global state, Speed Balancing is asynchronous and hence there are no guarantees that it will achieve the best balance. By contrast, we have carefully constructed and analyzed an algorithm that guarantees the best dynamic load balance, which we have confirmed both theoretically and with a simulation for a wide variety of configurations. Although our actual implementation uses global synchronization, in practice the overhead is small and has no effect on the performance.

Some attention has been paid to the off-by-one problem in operating-system (OS) scheduler design. The FreeBSD ULE scheduler [17] was originally designed to migrate threads twice a second, even if the imbalance in run queues was only one. More recently, Li et al. [12] developed the Distributed Weighted Round Robin (DWRR) scheduler as an extension to the Linux kernel. DWRR attempts to ensure fairness across processors by continually migrating threads, even for off-by-one imbalances. Under DWRR, the lag experienced by any thread τ_i at time t is bounded by $-3Bw < lag_i(t) < 2Bw$, where B is the *round slice unit*, equivalent to our load-balancing period λ, and w is the maximum thread weight. In SPMD applications all threads have the same weight, so the upper bound for the difference in progress between ahead and behind threads under DWRR would be the equivalent of 5λ. This upper bound is considerably worse than $\lambda/(\lceil n/m \rceil \lfloor n/m \rfloor)$, which is the largest difference in progress among threads under proactive load balancing.

The looser upper bound for the thread lag under DWRR illustrates some fundamental issues with load balancing of parallel applications in a traditional OS: when the balancer is part of an OS scheduler, it often becomes very complex because of the need to support applications with different priorities, different responsiveness, etc. It is hard to make a general scheduler and load balancer that works well for a large variety of applications. OS schedulers are extremely performance sensitive and hence tend to avoid using global information or any form of synchronization. Furthermore, they typically do not take into account the fact that a group of threads constitute a parallel application. These aspects limit the efficacy of OS scheduler approaches when applied to the particular problem of balancing SPMD applications.

Gang Scheduling [15] is an approach to dealing with extrinsic imbalances for data parallel applications in multiprogrammed environments. It has been shown to improve the performance of fine grained applications by enabling them to use busy-wait synchronization instead of blocking and thus avoid context-switch overheads [6, 4]. Gang Scheduling is beneficial on large-scale distributed systems where OS-jitter is problematic [9]. However, Gang Scheduling does not address the problem of off-by-one imbalances. Consequently, it can be regarded as complementary to proactive scheduling, which cannot balance very fine grained applications.

6. DISCUSSION AND FINAL REMARKS

Proactive load balancing can be a powerful technique for increasing the flexibility of SPMD parallelism, and improving its usability in multiprogrammed environments with unpredictable resource constraints. Our results indicate that it is most effective when the load-balancing period, λ, is much smaller than the computation-phase length. Consequently, for fine-grained parallelism λ needs to be small, but there are practical limitations to how small λ can be. The cost of migration and the overhead of executing Juggle impose a fundamental constraint on the minimum grain size. Our experiments show that the overhead of Juggle is about $100\mu s$, so as λ approaches this value, Juggle becomes impractical.

Our investigations of reactive load balancing have been cursory, limited to using the Linux Load Balancer as an imperfect example of a reactive balancer. A topic of future research is to implement a fully reactive balancer (i.e., one that rebalances immediately threads yield or sleep, instead of doing so periodically). It is possible that reactive balancing will be better for fine-grained parallelism, because the balancing events will coincide with synchronization, and so it will not introduce any additional, unnecessary overhead. A hybrid approach of using a periodically-triggered proactive balancer with a reactive balancer might give the best of both worlds.

Our analysis is a step towards a deeper theoretical understanding of dynamic load balancing for SPMD parallelism. Much work remains to be done. The analysis needs to be extended to cover the case of multiple phases and phases of different lengths. Much tighter bounds are required in the general case, for $\lfloor n/m \rfloor > 1$. The effects of overheads, such as migration, need to be incorporated into the model. The impact of multiprogrammed environments and sharing also needs to be quantified more precisely.

An interesting question is whether proactive load balancing would be relevant to HPC systems that batch schedule jobs and pin threads to cores. For example, consider a large distributed HPC system[8] composed of computing nodes, each featuring 16 processing elements and shared memory. Also consider running two jobs, J_1 and J_2, with processor requirements of 80% and 30%, respectively. The best a batch scheduler can do is to run the jobs one after

[8]such as Ranger at the Texas Advanced Computing Center (`http://www.tacc.utexas.edu/resources/hpc/`).

the other. Assuming that both jobs have a completion time of t, it will take $2t$ for them both to run and only 55% of the system will be utilized. Proactive load balancing could improve the utilization by allowing both jobs to run simultaneously. If the jobs are uniformly distributed across *all* nodes, then J_1 could run with 13 (16×0.8) threads per node and J_2 with 5 (16×0.3) threads per node. Consequently, in this scenario two processing elements in each node would be shared, so the progress rate for J_1 would ideally be equal to $(13/(13-2/2))^{-1}$ and for J_2 it would be $(5/(5-2/2))^{-1}$ until one of jobs completes. Therefore, proactive load balancing within a node would yield in the best case completion times of $CT_{J_1} = 13t/12 = 1.083t$ and $CT_{J_2} = 1.083t + (1-1.083/(5/4))t = 1.217t$ because J_1 completes before J_2. The utilization would be 92.3%. Although this is a simplistic example, we believe that it illustrates the need to further explore these questions.

One of our basic assumptions is that all processors have the same processing power, which is invalid in many cases (e.g., Intel's Turbo Boost selectively overclocks cores that are not too hot). The analysis assumes homogeneity, as does the implementation of Juggle. None-the-less, Juggle is effective for the hyper-threaded *Nehalem* system, where the processor (hyper-thread) capacities vary dynamically depending on the state of the paired hyper-thread. To incorporate processor heterogeneity into Juggle, we would not only have to modify the algorithm, but also alter the way thread progress is measured: instead of elapsed time, performance counters could be used, which would reflect the processing power of different processing elements. Our current implementation does not use performance counters because these are not portable, and are often used by parallel applications for performance monitoring and tuning.

Of practical importance is how architectures are going to change as core counts increase. Our implementation exhibits low overheads at small scales (up to 16 cores) and the complexity is bounded by the size of the domains in a NUMA system. For future systems with large NUMA domains we may have to increase the parallelism within Juggle so that it is still usable at scale. Although future systems are likely to consist of tens or even hundreds of NUMA domains, restricting inter-domain migrations should not result in much loss of balance, because domains will almost certainly be large enough to get close to the best possible relative speedup. For example, balancing 13 threads on a 12-core domain gives a $2/(13/12) = 1.85$ relative speedup, and 25 threads on a 24-core domain gives a $2/(25/24) = 1.92$ relative speedup.

7. ACKNOWLEDGEMENTS

The authors acknowledge the support of DOE ASCR FastOS Grant #DE-FG02-08ER25849. Juan Colmenares and John Kubiatowicz acknowledge support of Microsoft (Award #024263), Intel (Award #024894), matching U.C. Discovery funding (Award #DIG07-102270), and additional support from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Samsung, and Sun Microsystems. No part of this paper represents the views and opinions of the sponsors.

8. REFERENCES

[1] R. D. Blumofe and D. Papadopoulos. The performance of work stealing in multiprogrammed environments (extended abstract). *ACM SIGMETRICS Perform. Eval. Rev.*, 26(1):266–267, 1998.

[2] C. Boneti et al. A Dynamic Scheduler for Balancing HPC Applications. In *Proc. 2008 ACM/IEEE Supercomputing Conference*, 2008.

[3] F. Cedo et al. The Convergence of Realistic Distributed Load-Balancing Algorithms. *Theory Computer Systems*, 41:609–618, 2007.

[4] D. G. Feitelson and L. Rudolph. Gang Scheduling Performance Benefits for Fine-Grain Synchronization. *J. Parallel and Distributed Computing*, 16:306–318, 1992.

[5] C. Fonlupt et al. Data-parallel load balancing strategies. *Parallel Computing*, 24:1665–1684, 1996.

[6] A. Gupta et al. The Impact Of Operating System Scheduling Policies And Synchronization Methods On Performance Of Parallel Applications. *SIGMETRICS Perform. Eval. Rev.*, 19(1), 1991.

[7] S. Hofmeyr et al. Load Balancing on Speed. In *Proc. 15th ACM Sym. on Principles & Practice of Parallel Programming*, 2010.

[8] C. Iancu et al. Oversubscription on multicore processors. In *Proc. 24rd Int'l Parallel and Distributed Processing Sym. (IPDPS)*, 2010.

[9] T. Jones et al. Improving the Scalability of Parallel Jobs by adding Parallel Awareness to the OS. *ACM Supercomputing*, 2003.

[10] Z. Khan et al. Performance analysis of Dynamic Load Balancing Techniques for Parallel and Distributed Systems. *(IJCNS) Int'l J. of Computer and Network Security*, 2, 2010.

[11] A. Kukanov et al. The Foundations for Scalable Multi-Core Software in IntelÂő Threading Building Blocks. *Intel Tech. Jour.*, 2007.

[12] T. Li et al. Efficient And Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round-Robin. In *Proc. 14th ACM SIGPLAN Sym. on Principles and Practice of Parallel Programming*, 2009.

[13] R. Nishtala et al. Optimizing Collective Communication on Multicores. In *Proc. 1st USENIX Ws. on Hot Topics in Parallelism*, 2009.

[14] S. Olivier and J. Prins. Scalable Dynamic Load Balancing Using UPC. In *Proc. 37th Int'l Conf. on Parallel Processing*, pages 123–131, 2008.

[15] J. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proc. 3rd Int'l Conf. on Distributed Computing Systems*, 1982.

[16] A. Plastino et al. Developing SPMD Applications with Load Balancing. *Parallel Computing*, pages 743–766, 2003.

[17] J. Roberson. ULE: A Modern Scheduler for FreeBSD. In *USENIX BSDCon*, pages 17–28, 2003.

[18] M. Willebeek-LeMair and A. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. on Parallel and Distributed Systems*, 4, 1993.

[19] C. Xu and F. C. Lau. *Load Balancing in Parallel Computers: Theory and Practice*. Kluwer Academic Publishers, 1997.