Cluster Comput
DOI 10.1007/s10586-012-0204-0

Juggle: addressing extrinsic load imbalances in SPMD applications on multicore computers

Steven Hofmeyr · Juan A. Colmenares · Costin Iancu · John Kubiatowicz

Received: 12 September 2011 / Accepted: 25 February 2012

Abstract We investigate *proactive* dynamic load balancing on multicore systems, in which threads are continually migrated to reduce the impact of processor/thread mismatches. Our goal is 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. In this paper we assume perfect intrinsic load balance and focus on *extrinsic* imbalances caused by OS noise, multiprogramming and mismatches of threads to hardware parallelism. Juggle shows performance improvements of up to 80 % over static load balancing for oversubscribed UPC, OpenMP, and pthreads benchmarks. We also show that Juggle is effective in unpredictable, multiprogrammed environments, with up to a 50 % performance improvement over the Linux load balancer and a 25 % reduction in performance variation. 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.

Keywords Proactive load balancing · Parallel programming · Single-program multiple-data parallelism · Operating system · Multicore

S. Hofmeyr (✉) · C. Iancu
Lawrence Berkeley National Laboratory, Berkeley, CA, USA
e-mail: shofmeyr@lbl.gov

J.A. Colmenares · J. Kubiatowicz
Parallel Computing Laboratory, UC Berkeley, Berkeley, CA, USA

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 High Performance Computing (HPC). In SPMD programs each thread executes the same code on a different partition of a data set, which usually means that applications have a fixed number of threads. Typically, in HPC systems each thread runs on a 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, such as running in-situ analytics or system services on compute nodes, but it could also make SPMD-style parallelism more suitable for non-HPC markets, where execution environments are multiprogrammed and unpredictable. Programmers in non-HPC domains will need to exploit increasing on-chip concurrency, and hence can benefit from tools that enable them to leverage HPC expertise in parallel programming.

A key to SPMD performance is load balance because it helps minimize the amount of idle time a thread spends waiting for other threads' results. Load imbalances can be classified as *intrinsic* or *extrinsic* [2]. Intrinsic imbalances are caused by characteristics inherent to the application, such as non-uniform partitioning of the data set. Extrinsic imbalances, on the other hand, result from external factors, such as interference from the operating system (OS) [22]

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

and other simultaneously running applications (in multiprogrammed environments), as well as oversubscription, where there are more threads than processors. Often there is a tension between extrinsic and intrinsic load balance, because frequently the simplest way to achieve intrinsic balance involves using constrained thread counts (e.g., powers of two) that do not match the available processor counts.

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 or application code. We wish to reduce the effect of constraints so that SPMD applications can be more easily run in non-dedicated (multiprogrammed) environments, and with varying processor counts.

Our intention is to offer another way of adapting to the ever-changing amount of available parallelism, one that works for legacy code and older programming models that are still in widespread use, such as MPI. Furthermore, as the concurrency increases on the node in future HPC systems, constraints in memory accesses and networking will prevent many applications from taking full advantage of the computational capacity of all the cores [21]. These spare cores are likely to be used for OS services (e.g., the REDFish project[2]) and in-situ analytics, resulting in noisy, multiprogrammed environments, even on HPC systems.

The simplest approach to load balancing is *static*: threads are initially distributed as uniformly as possible, and then not migrated subsequently. Static balancing can always achieve at least an *off-by-one* balance (where the number of threads on any processor is within one of every other processor) for any n threads on m processors (where $n \geq m$) by a simple cyclic distribution of threads to processors. However, to overcome that final, off-by-one imbalance requires dynamic load balancing, where threads are migrated while running.

Our investigations focus on the *proactive* approach to dynamic 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 at the same rate. Given n identical threads to run on m processors, where $n \geq m$, we want each thread to ideally receive m/n processor time over the course of a computation phase. Most SPMD applications have a pattern of computation phases and communication, with barrier synchronization. If all threads progress at the same rate, they will reach the barriers simultaneously so that no time is wasted waiting on the barrier. Provided the load balancing period, λ, is small compared to the computation-phase length, over time the *progress rate* of every thread will tend to m/n, as illustrated in Fig. 1.

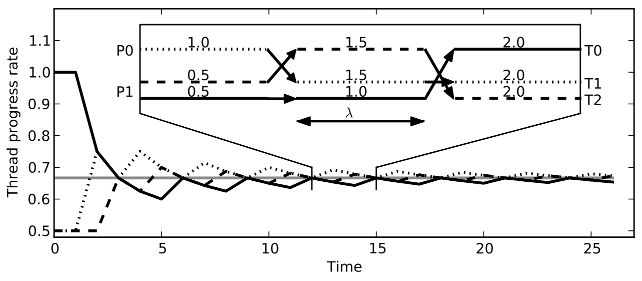


Fig. 1 Load balancing three threads (T0, T1, T2) on two processors (P0, P1), using a load-balancing period of $\lambda = 1s$. The *gray line* indicates a progress rate of $m/n = 2/3$. In the *top inset*, migrations are indicated by the *arrows* between processors; the *number* is the total progress of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

An alternative dynamic load balancing approach is *reactive*, where threads are rebalanced only when the load on a processor changes (i.e., some thread enters the barrier). Typically, OSs implement a form of reactive balancing. For example, Linux tries to achieve at most an off-by-one balance on active run queues, so if a thread sleeps when it reaches the barrier, it will be moved off the active run queue, enabling the OS to rebalance the remaining active threads. The Linux load balancer will not continually migrate threads between active run queues that are only one different in length.[3]

The concept of proactive load balancing is not new [9, 15, 20]. 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 oversubscribed 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, with up to a 50 % performance improvement over the Linux load balancer and a 25 % reduction in performance variation. 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. In this paper we investigate the simplest case where we have uniform task sizes (we assume perfect intrinsic balance) and the costs of migration are negligible (we ignore non-local memory access effects and migration costs). Our intention is to lay the foundations for future work that considers both irregular applications and migration costs.

This paper presents an extension of our previous work reported in [10]. Here we include additional theoretical results on lower bounds for Juggle in Sect. 3.2.1 and enhanced

[2]http://newmexicoconsortium.org/usrc/usrc-publication-pdfs/REDfish.pdf.

[3]This is a simplification. Load balancing in Linux is also dependent on the memory hierarchy, through scheduling domains.

derivations and explanations of other theoretical results. We also present more extensive results on multiprogramming in Sect. 5.4. In addition to a more in-depth exploration of Juggle in an unpredictable multiprogrammed environment, we evaluate the performance of multiple independent instances of Juggle running simultaneously.

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 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 through the use of 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 Fig. 2. All

```
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]
```

Fig. 2 Pseudo code for Juggle

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 scalability of our approach in Sect. 2.5. The steps shown in Fig. 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 calculate $\Delta\overline{P}_{\rho_j} = (1/|\mathcal{T}_{\rho_j}|)\sum_{\tau_i \in \mathcal{T}_{\rho_j}} \Delta P_{\tau_i}(k\lambda)$, which is the average of the change in the progress of all the threads on ρ_j during the most recent load-balancing period. $\Delta\overline{P}_{\rho_j}$ is proportional to the processor's speed collectively experienced by the threads running on ρ_j and it is later used to determine whether ρ_j will run threads fast or slow.

A difficulty of simply measuring elapsed time is that it may not accurately reflect actual progress, for example, when a thread is spinning on a barrier. The impact of this depends on the type of synchronization used. The runtimes tested in this paper use three different kinds of synchronization implementations: *busy-wait*, where threads spin in a tight loop, *spin-yield*, where threads spin in a loop calling yield each time, and *blocking*, where threads sleep until they receive a notification. In this paper, we focus on oversubscribed and multiprogrammed environments, where busy-wait is usually avoided because it results in idle threads consuming processor time. In the case of spin-yield, we have no way of differentiating spinning on a barrier from useful work. However, the impact is not great because inaccuracies only occur when all threads on a core are spinning on yield; otherwise any non-spinning threads will effectively consume all processor time. In the case of blocking, we can get better information, since a sleeping thread only accumulates idle time. Consequently, to discount idle time we multiply $\Delta\overline{P}_{\rho_j}$ 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 Fig. 3.

```
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 $\tau_{bh}$ on $\rho_s$
4         **if** no behind threads on $\rho_s$
5           **then** go to Line 2
6         Get next fast processor $\rho_f \in \mathcal{P}_{fast}$
7         Get next fast slot (occupied by
            the ahead thread $\tau_{ah}$) on $\rho_f$
8         **if** no more fast slots on $\rho_f$
9           **then** go to Line 2
10        Set $\tau_{bh}$ to be migrated to $\rho_f$
11        Set $\tau_{ah}$ to be migrated to $\rho_s$
```

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

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 Fig. 3) starts at the processor which has threads with the least average progress, whereas the selection of the next fast processor (line 6) 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.

Lines 10 and 11 in Fig. 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 Sect. 5.4).

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 Sect. 5.1), 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. [16] implemented a barrier that takes 2 µs 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

In this section 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, $\mathcal{T} = \{\tau_1, \dots, \tau_n\}$, running on m homogeneous processors, $\mathcal{P} = \{\rho_1, \dots, \rho_m\}$, 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 (or equivalently 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 $\mathcal{T}$. For a summary of notation used in the analysis, refer to Table 1.

An idealized version of our proactive load-balancing algorithm is shown in Fig. 4. The algorithm executes periodically every λ time units (the load-balancing period). On

Table 1 Notation used in the analysis of Juggle

Term	Meaning
n	Number of threads
m	Number of processors
$\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$	Set of n identical threads
$\mathcal{P} = \{\rho_1, \ldots, \rho_m\}$	Set of m homogeneous processors
e	Units of time a thread $\tau_i \in \mathcal{T}$ requires to complete its work when running on a dedicated processor $\rho_j \in \mathcal{P}$
λ	Length of the load-balancing period
Φ	A computation phase involving the thread set $\mathcal{T}$
t_0	Start time of the thread set $\mathcal{T}$
t_f	Time when the last thread in $\mathcal{T}$ completes and reaches the barrier
$CT_{\mathcal{T}} = t_f - t_0$	Completion time of the thread set $\mathcal{T}$
$\mathcal{S}_{\Phi}$	Finite sequence of executions of a computation phase Φ
$l_{\mathcal{S}_{\Phi}} \in \mathbb{N}^+$	Number of executions of Φ in $\mathcal{S}_{\Phi}$ (i.e., the sequence's length)
$CT_{\mathcal{S}_{\Phi}}$	Completion time of the entire sequence $\mathcal{S}_{\Phi}$
$\mathcal{P}_{slow} \in \mathcal{P}$	Subset of processors with threads running at slow speed
$\mathcal{P}_{fast} \in \mathcal{P}$	Subset of processors with threads running at fast speed
n_{slow}	Number of non-blocked threads in $\mathcal{T}$ assigned to processors in $\mathcal{P}_{slow}$ and that run at slow speed
n_{fast}	Number of non-blocked threads in $\mathcal{T}$ assigned to processors in $\mathcal{P}_{fast}$ and that run at fast speed
$\mathcal{T}_{ah} \in \mathcal{T}$	Subset of threads that are ahead
$\mathcal{T}_{bh} \in \mathcal{T}$	Subset of threads that are behind
n_{ah}	Number of ahead threads
n_{bh}	Number of behind threads
$P_{\tau_i}(t)$	Progress that the thread τ_i has made by time t
$\Delta P(t)$	Difference between progress of an ahead thread and a behind thread at time t

each execution, it updates the progress of the threads, sorts them in increasing order of progress (Line 7), and updates the count of non-blocked threads, denoted as n^* (Line 8). Then, it 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 Lines 18–30 in Fig. 4, and the illustration in Fig. 5).

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 (Sect. 3.1), and the execution of a *sequence* of Φ (Sect. 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 Sect. 5).

In the worst case, proactive load balancing is theoretically equivalent to static load balancing, because we assume negligible 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 by an amount ΔP_{imag}. In Theorem 1 we show that at any load-balancing point, $\Delta P_{imag} \geq \lambda/(\lceil n/m \rceil \lfloor n/m \rfloor)$. 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

```
    ▷ Initialization
1   $m \leftarrow$ Number the processors in the system.
2   $arr[1 \ldots n]$ ▷ Array that contains the threads in $\mathcal{T}$.
3   Set progress of each thread in $arr[1 \ldots n]$ equal to zero.

4   from $t = 0$, every $\lambda$ time units
5       do
6           Update progress of threads in $arr[1 \ldots n]$.
7           Sort the threads in $arr[1 \ldots n]$ in increasing
            order of progress.
8           $n^* \leftarrow$ Current number of non-blocked threads
                in $arr[1..n]$.
9           if $n^* > m$
10            then $|\mathcal{P}_{slow}| \leftarrow n^* \bmod m$
11                 $|\mathcal{P}_{fast}| \leftarrow m - |\mathcal{P}_{slow}|$
12                 $n^*_{slow} \leftarrow (n^* \bmod m) \times \lceil n^*/m \rceil$
13                 $n^*_{fast} \leftarrow n^* - n^*_{slow}$
14            else $|\mathcal{P}_{slow}| \leftarrow 0$
15                 $|\mathcal{P}_{fast}| \leftarrow m$
16                 $n^*_{slow} \leftarrow 0$
17                 $n^*_{fast} \leftarrow n^*$
18          $f \leftarrow 1$
19          $s \leftarrow m - |\mathcal{P}_{slow}| + 1$
20          for $j \leftarrow 1$ to $n^*$
21              do
22                 if $j \leq n^*_{fast}$
23                   then Assign thread $arr[j]$
                          to processor $\rho_f$
24                        $f \leftarrow f + 1$
25                        if $f > |\mathcal{P}_{fast}|$
26                          then $f \leftarrow 1$
27                   else Assign thread $arr[j]$
                          to processor $\rho_s$
28                        $s \leftarrow s + 1$
29                        if $s > m$
30                          then $s \leftarrow m - |\mathcal{P}_{slow}| + 1$
```

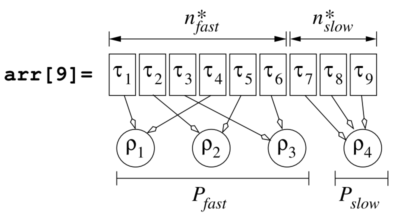
Fig. 4 Pseudo code for an idealized version of Juggle

completion time of the thread set $\mathcal{T}$ is:

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

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).



n^*_{fast}
n^*_{slow}
arr[9]=
τ_1 τ_2 τ_3 τ_4 τ_5 τ_6 τ_7 τ_8 τ_9
ρ_1 ρ_2 ρ_3 ρ_4
P_{fast}
P_{slow}

Fig. 5 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 $\mathcal{P}_{slow}$

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 that $n > m$ (which is one of our fundamental assumptions).

Equation (3) suggests that threads can be grouped according to their progress into a set $\mathcal{T}_{ah}$ of *ahead threads* of size n_{ah} and 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 k^{th} 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}. As shown in Table 2, these scenarios can be grouped into three general cases that share identical analyzes. Figure 6 presents examples of the general cases.

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 $1/\lceil n/m \rceil$.

Table 2 All possible scenarios in terms of the relations among n_{fast}, n_{slow}, n_{ah}, and n_{bh} at the load-balancing point at $t = k\lambda$ and the resulting difference in the progress of ahead threads and behind threads by the next balancing point at $t = (k+1)\lambda$

	$n_{slow} < n_{fast}$		
$n_{bh} < n_{ah}$	$n_{bh} < n_{slow} < n_{fast} < n_{ah}$ ∴ **[A]**	$n_{bh} = n_{slow} < n_{fast} = n_{ah}$ ∴ **[A]**	$n_{slow} < n_{bh} < n_{ah} < n_{fast}$ ∴ **[A]**
$n_{bh} > n_{ah}$	$n_{slow} < n_{ah} < n_{bh} < n_{fast}$ ∴ **[A]**	$n_{slow} = n_{ah} < n_{bh} = n_{fast}$ ∴ **[C]**	$n_{ah} < n_{slow} < n_{fast} < n_{bh}$ ∴ **[B]**
$n_{bh} = n_{ah}$	$n_{slow} < n_{bh} = n_{ah} < n_{fast}$ ∴ **[A]**		
	$n_{slow} > n_{fast}$		
$n_{bh} < n_{ah}$	$n_{bh} < n_{fast} < n_{slow} < n_{ah}$ ∴ **[A]**	$n_{bh} = n_{fast} < n_{slow} = n_{ah}$ ∴ **[C]**	$n_{fast} < n_{bh} < n_{ah} < n_{slow}$ ∴ **[B]**
$n_{bh} > n_{ah}$	$n_{fast} < n_{ah} < n_{bh} < n_{slow}$ ∴ **[B]**	$n_{fast} = n_{ah} < n_{slow} = n_{bh}$ ∴ **[B]**	$n_{ah} < n_{fast} < n_{slow} < n_{bh}$ ∴ **[B]**
$n_{bh} = n_{ah}$	$n_{fast} < n_{bh} = n_{ah} < n_{slow}$ ∴ **[B]**		
	$n_{slow} = n_{fast}$		
$n_{bh} < n_{ah}$	$n_{bh} < n_{slow} = n_{fast} < n_{ah}$ ∴ **[A]**		
$n_{bh} > n_{ah}$	$n_{ah} < n_{slow} = n_{fast} < n_{bh}$ ∴ **[B]**		
$n_{bh} = n_{ah}$	$n_{bh} = n_{slow} = n_{fast} = n_{ah}$ ∴ **[C]**		

Case [A]: $n_{bh} < n_{fast}$ at $t = k\lambda$ and $\Delta P((k+1)\lambda) = \lambda/(\lceil n/m \rceil \lfloor n/m \rfloor)$
Case [B]: $n_{bh} > n_{fast}$ at $t = k\lambda$ and $\Delta P((k+1)\lambda) = \lambda/(\lceil n/m \rceil \lfloor n/m \rfloor)$
Case [C]: $n_{bh} = n_{fast}$ at $t = k\lambda$ and $\Delta P((k+1)\lambda) = 0$

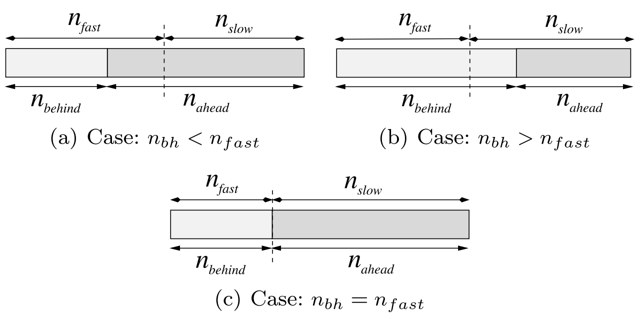


Fig. 6 Examples of the general cases for the derivation of $\Delta P((k+1)\lambda)$

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

$$P_{G_{bh\to fast}}\big((k+1)\lambda\big) = P_{\mathcal{T}_{bh}}(k\lambda) + \lambda/\lfloor n/m \rfloor \quad (4)$$

$$P_{G_{ah\to slow}}\big((k+1)\lambda\big) = P_{\mathcal{T}_{ah}}(k\lambda) + \lambda/\lceil n/m \rceil \quad (5)$$

$$P_{G_{ah\to fast}}\big((k+1)\lambda\big) = P_{\mathcal{T}_{ah}}(k\lambda) + \lambda/\lfloor n/m \rfloor \quad (6)$$

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 Eq. (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 Eq. (6) from Eq. (5) we obtain:

$$\Delta P\big((k+1)\lambda\big) = \lambda/\big(\lceil n/m \rceil \lfloor n/m \rfloor\big) \quad (7)$$

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:

$$P_{G_{ah\to slow}}\big((k+1)\lambda\big) = P_{\mathcal{T}_{ah}}(k\lambda) + \lambda/\lceil n/m \rceil \quad (8)$$

$$P_{G_{bh\to fast}}\big((k+1)\lambda\big) = P_{\mathcal{T}_{bh}}(k\lambda) + \lambda/\lfloor n/m \rfloor \quad (9)$$

$$P_{G_{bh\to slow}}\big((k+1)\lambda\big) = P_{\mathcal{T}_{bh}}(k\lambda) + \lambda/\lceil n/m \rceil \quad (10)$$

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 Eq. (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 Eq. (10) from Eq. (9) we obtain:

$$\Delta P\big((k+1)\lambda\big) = \lambda/\big(\lceil n/m \rceil \lfloor n/m \rfloor\big) \quad (11)$$

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:

$$P_{G_{ah\to slow}}\big((k+1)\lambda\big) = P_{\mathcal{T}_{ah}}(k\lambda) + \lambda/\lceil n/m \rceil \quad (12)$$

$$P_{G_{bh\to fast}}\big((k+1)\lambda\big) = P_{\mathcal{T}_{bh}}(k\lambda) + \lambda/\lfloor n/m \rfloor \quad (13)$$

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 Eq. (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 $t = 0$. □

Now we can prove Theorem 1.

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 $k \in \mathbb{N}^+$*

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

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, Eq. (2) is a good predictor of performance when $e \approx \lambda$. However, when $\lambda \ll e$, Eq. (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 infinitesimal, 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^*.

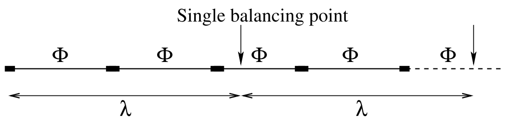


Fig. 7 A sequence of executions of a single computation phase Φ with the load-balancing period λ extending over multiple executions

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 Fig. 7). 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.

3.2.1 Lower bounds

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 Eq. (14), we get

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

This bound is reasonably tight when $\lambda \ll e$. When $\lambda \geq e \times \lceil \frac{n}{m} \rceil$, we can refine the analysis further.

We consider first the case in which the threads in $\mathcal{T}$ are *only* balanced at the beginning of each execution of Φ (i.e., static load-balancing in each execution of Φ). From Eq. (1), the completion time of the entire sequence here is given by:

$$CT_\Phi^{static} = l_{\mathcal{S}_\Phi} \times e \times \lceil n/m \rceil \tag{16}$$

Now assume that $l_{\mathcal{S}_\Phi} \geq 2$. If $\lambda < l_{\mathcal{S}_\Phi} \times e \times \lceil n/m \rceil$ and $(l_{\mathcal{S}_\Phi} \times e \times \lceil n/m \rceil) \bmod \lambda \neq 0$, then some executions of Φ in $\mathcal{S}_\Phi$ will be statically balanced while other executions will contain a *single load-balancing point*. 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 assume that whenever a single load-balancing point occurs in an execution of Φ, it results in the minimum possible completion time, $\min(CT^*_{\mathcal{T}})$, for the thread set $\mathcal{T}$ for that particular execution phase, then a lower bound on the completion time is

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

In Theorem 2 we derive the minimum possible completion time for the thread set $\mathcal{T}$ when balanced once during a computation phase, assuming $n_{fast} \geq n_{slow}$. 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 Eq. (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 $n_{fast} < n_{slow}$, Eq. (17) simply becomes

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

Theorem 2 *Let* $\min(CT^*_{\mathcal{T}})$ *be the minimum completion time of a thread set* $\mathcal{T}$ *when balanced exactly once during the computation phase,* Φ. *If* $\lambda \geq e \times \lceil \frac{n}{m} \rceil$ *and* $n_{fast} \geq n_{slow}$ *at the start of* Φ, *then*

$$\min\bigl(CT^*_{\mathcal{T}}\bigr) = \lfloor n/m \rfloor \left(1 + \frac{1}{\lceil n/m \rceil + \lfloor n/m \rfloor}\right) e$$

Proof Consider a computation phase Φ that involves a set $\mathcal{T}$ of n identical threads running on a set $\mathcal{P}$ of m homogeneous processors, where $n > m$ and $n \bmod m \neq 0$. Initially the threads in $\mathcal{T}$ are distributed evenly among the processors in $\mathcal{P}$. After that, Φ starts with the simultaneous activation of the threads in $\mathcal{T}$. We assume without loss of generality that Φ starts at $t = 0$, and the load-balancing algorithm executes at t_{lb} and that no thread completes before t_{lb}.

Given the assumption that $n_{fast} \geq n_{slow}$ at $t = 0$, the threads in $\mathcal{T}$ can be divided into three disjoint subsets:

- $\mathcal{A}$ contains the threads that always progress at a rate of $\lfloor n/m \rfloor^{-1}$ (i.e., they always run at fast speed). Note that $\mathcal{A} = \emptyset$ if $n_{fast} = n_{slow}$.
- $\mathcal{B}$ includes the threads that progress at a rate of $\lfloor n/m \rfloor^{-1}$ from $t = 0$ to t_{lb}. At t_{lb} they are moved by the load balancing, and so from t_{lb} onwards the threads in $\mathcal{B}$ progress at a rate of $\lceil n/m \rceil^{-1}$ until they complete at $t^{\mathcal{B}}_f$. In short, they go from fast to slow at t_{lb}.
- $\mathcal{C}$ comprises the threads that go from slow to fast at t_{lb}. These threads progress at a rate of $\lceil n/m \rceil^{-1}$ from $t = 0$ to t_{lb} and at a rate of $\lfloor n/m \rfloor^{-1}$ from t_{lb} until they complete at $t^{\mathcal{C}}_f$.

The threads in $\mathcal{A}$ complete at $t^{\mathcal{A}}_f = \lfloor n/m \rfloor e$. Since they never run at slow speed they finish before the threads in both $\mathcal{B}$ and $\mathcal{C}$. Therefore, threads in the subset $\mathcal{A} \subset \mathcal{T}$ do not influence $CT_{\mathcal{T}}$.

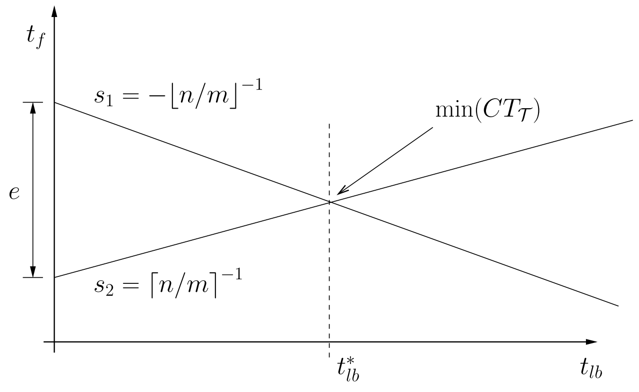


Fig. 8 Graphical representation of Eqs. (20) and (22). The minimum value of $CT_{\mathcal{T}} = \max(t^{\mathcal{B}}_f, t^{\mathcal{C}}_f)$ occurs at the intersection of Eqs. (20) and (22)

The execution time for threads in $\mathcal{B}$ is

$$e = t_{lb}/\lfloor n/m \rfloor + \bigl(t^{\mathcal{B}}_f - t_{lb}\bigr)/\lceil n/m \rceil$$

Thus

$$t^{\mathcal{B}}_f = \lceil n/m \rceil \bigl(e - t_{lb}\bigl(1/\lfloor n/m \rfloor - 1/\lceil n/m \rceil\bigr)\bigr) \tag{19}$$

$$= \lceil n/m \rceil e - t_{lb}/\lfloor n/m \rfloor \tag{20}$$

Similarly, the behavior of the threads in $\mathcal{C}$ can be represented as:

$$e = t_{lb}/\lceil n/m \rceil + \bigl(t^{\mathcal{C}}_f - t_{lb}\bigr)/\lfloor n/m \rfloor$$

Thus

$$t^{\mathcal{C}}_f = \lfloor n/m \rfloor \bigl(e + t_{lb}\bigl(1/\lfloor n/m \rfloor - 1/\lceil n/m \rceil\bigr)\bigr) \tag{21}$$

$$= \lfloor n/m \rfloor e + t_{lb}/\lceil n/m \rceil \tag{22}$$

The completion time of the thread set $\mathcal{T}$ is thus

$$CT_{\mathcal{T}} = \max\bigl(t^{\mathcal{B}}_f, t^{\mathcal{C}}_f\bigr)$$

The minimum $CT_{\mathcal{T}}$ occurs when $t^{\mathcal{B}}_f = t^{\mathcal{C}}_f$ (see Fig. 8). We can obtain t^*_{lb}, the value of t_{lb} that minimizes the completion time, by equating (20) and (22):

$$\lfloor n/m \rfloor e + t^*_{lb}/\lceil n/m \rceil = \lceil n/m \rceil e - t^*_{lb}/\lfloor n/m \rfloor$$

$$\Rightarrow \quad t^*_{lb} = \left(\frac{\lfloor n/m \rfloor \lceil n/m \rceil}{\lfloor n/m \rfloor + \lceil n/m \rceil}\right) e \tag{23}$$

By substituting Eq. (23) in Eq. (22), we obtain:

$$\min\bigl(CT^*_{\mathcal{T}}\bigr) = \lfloor n/m \rfloor \left(1 + \frac{1}{\lfloor n/m \rfloor + \lceil n/m \rceil}\right) e \tag{24}$$

□

3.2.2 Approximations

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

$$CT_{\mathcal{S}_\Phi} \approx l_{\mathcal{S}_\Phi} \times CT_{\mathcal{T}} \tag{25}$$

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:

$$CT_{\mathcal{S}_\Phi} \approx (l_{\mathcal{S}_\Phi} - \eta) \times e \times \lceil n/m \rceil + \eta \times \overline{CT}^{*}_{\mathcal{T}} \tag{26}$$

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

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

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., $CT^{q/k}_{\mathcal{T}} = CT^{1-q/k}_{\mathcal{T}}$. Hence we compute the average over the first half of the interval

$$\begin{aligned} \overline{CT_{\mathcal{T}}} &= \frac{2}{k}\sum_{q=1}^{k/2} CT^{q}_{\mathcal{T}} \\ &= \frac{2e}{k}\sum_{q=1}^{k/2}\big(q/k\lceil n/m \rceil + (1 - q/k)\lfloor n/m \rfloor\big) \end{aligned}$$

Since $\lceil n/m \rceil = \lfloor n/m \rfloor + 1$, this reduces to

$$\begin{aligned} \overline{CT_{\mathcal{T}}} &= \frac{2e}{k}\sum_{q=1}^{k/2}\big(\lceil n/m \rceil - q/k\big) \\ &= \frac{2e}{k}\left(k/2\lceil n/m \rceil - \frac{k/2(k/2+1)}{2k}\right) \end{aligned}$$

Now $k/2 + 1 \approx k/2$ for large k, so this becomes

$$\overline{CT_{\mathcal{T}}} = e\big(\lceil n/m \rceil - 1/4\big) \tag{27}$$

4 An analysis of reactive load balancing

In this section 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 in this case, 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

$$CT^{react}_{\mathcal{T}} = \sum_{i=0}^{k}\frac{1}{2^i} = 2 - 2^{-k} \tag{28}$$

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

$$\begin{aligned} n_f(0) &= \lfloor n/m \rfloor (n \bmod m) \\ &= \lfloor n/m \rfloor \big(m\lceil n/m \rceil - n\big) \\ &= 2m - n \end{aligned}$$

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

$$n_f(i) = (2m - n) \times 2^i \tag{29}$$

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 Eq. (29)

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

Substituting k back into Eq. (28) gives

$$CT^{react}_{\mathcal{T}} = 2 - 2^{\lfloor \log_2(2 - n/m) \rfloor} \tag{30}$$

5 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.[4] We show that for many cases, the lower bounds derived in Sect. 3 are useful predictors of performance, and demonstrate that actual performance of Juggle closely follows a simulation of the idealized algorithm (presented in Fig. 4). 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. Table 3 lists the three systems used in the experiments.

Whenever we report a *speedup*, it is relative to the statically balanced case with the same thread and processor configuration, i.e.,

$$\mathit{Relative\ Speedup} = \frac{CT_{static}}{CT_{dynamic}}$$

Completion times for statically balanced cases are given by Eqs. (1) and (16). Moreover, we define the *upper bound* for the relative speedup as:

$$UB(\mathit{Relative\ Speedup}) = \frac{CT_{static}}{LB(CT_{dynamic})}$$

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. Most reported results are the average of 10 runs and we do not report their variations because they are small. The experiments on multiprogrammed environments (Sect. 5.4) are the exception; in those experiments we ran each benchmark 25 times and we report performance variations.

Table 3 Systems used in experiments

	Nehalem	Tigerton	Barcelona
Processor	Xeon E5530	Xeon E7310	Opteron 8350
Manufacturer	Intel	Intel	AMD
Clock (GHz)	2.4	1.6	2.0
Cores	8	16	16
Hdw. threads	2/core	1/core	1/core
L1 D/I cache	32 KB/32 KB	32 KB/32 KB	64 KB/64 KB
L2 cache	512 KB/core	4 MB/2 cores	512 KB/core
L3 cache	2 MB/4 cores	None	4 GB/4 cores
Memory/Core	4 GB	2 GB	4 GB
NUMA nodes	2	1	4

[4]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.

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 where threads sleep on barriers (block), 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. This is necessary because we have found that often applications can start up very poorly balanced and it can take the LLB a long time to get to an off-by-one balance.

5.1 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 Fig. 9, the theory derived in Sect. 3.1 (Eq. (14)) for the ideal case closely predicts the empirical performance when $\lambda \ll e$. Figure 9 presents the results of running µbench on the 8-core Intel *Nehalem* 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.

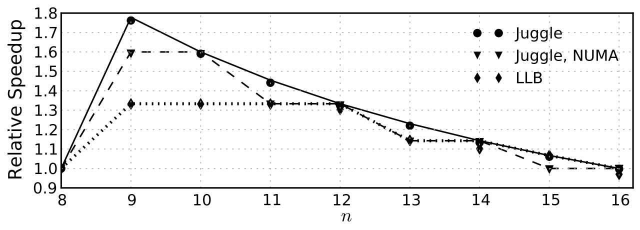


Fig. 9 Relative speedup of µbench with Juggle and LLB. The configuration parameters are $n \in [8, 16]$, $m = 8$, $e = 30$ s and $\lambda = 100$ ms. *Markers* are empirical results and *lines* are theoretical results

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 Sect. 5.3.

Figure 9 also shows a theoretical upper bound for the relative speedup when using reactive load balancing (Eq. (30)). 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 10 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 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 [11]; consequently, oversubscription cannot be regarded as a general solution for load imbalances.

We illustrate this fact in Table 4, 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 $\lfloor n/m \rfloor \le 2$.

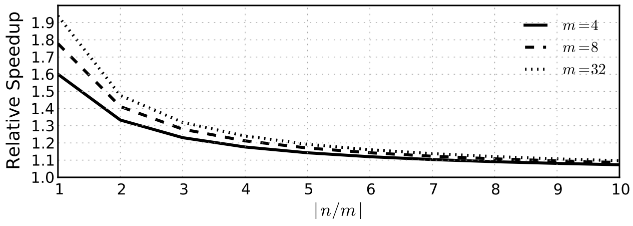


Fig. 10 Ideal relative speedup of proactive load balancing for increasing $\lfloor n/m \rfloor$; n is chosen to give the most imbalanced static thread distribution (e.g., $n = 9$ when $m = 8$, $n = 33$ when $m = 32$, etc.)

Table 4 Effects of oversubscription on the completion time, in seconds, of UPC FT, class C

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

5.2 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 the 8-core *Nehalem* with hyper-threading disabled. EP has a single phase, with $e = 30$ s on the *Nehalem*, and uses no memory, so we can ignore the impact of NUMA and balance fully across all 8 cores. Figure 11 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 Fig. 11, we have tested Juggle on a variety of configurations up to $\lfloor n/m \rfloor = 4$. A selection of these results is shown in Figs. 12 and 13. We can

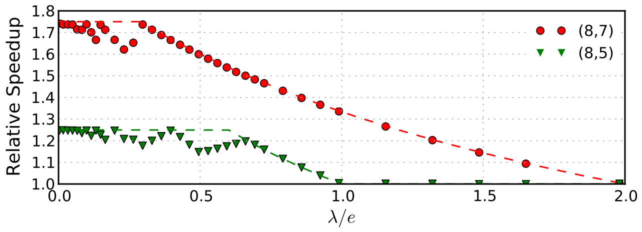


Fig. 11 Relative speedup of UPC EP class C with Juggle for cases where $\lfloor n/m \rfloor = 1$; $e = 30$ s, and λ varies. Threads and processing elements (i.e., cores) are denoted by (n, m). *Dashed lines* are the upper bounds for the relative speedup and markers are empirical results

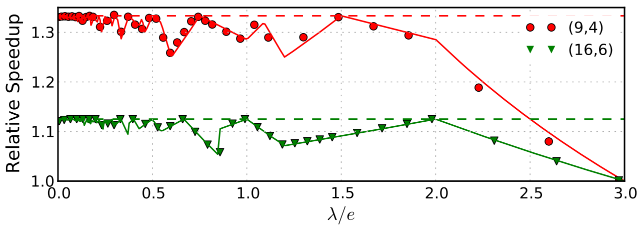


Fig. 12 Relative speedup of UPC EP class C with Juggle for cases where $\lfloor n/m \rfloor = 2$; $e = 30$ s and λ varies. Threads and processing elements are denoted by (n, m). *Dashed lines* are the upper bounds, *solid lines* are the simulation of the ideal algorithm, and *markers* are empirical results

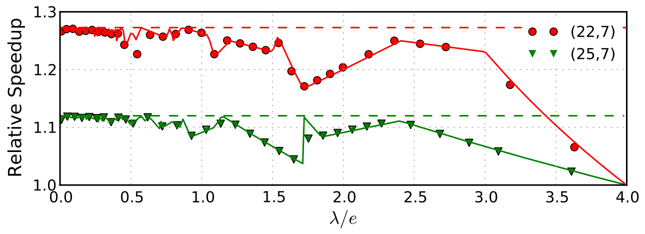


Fig. 13 Relative speedup of UPC EP class C with Juggle for cases where $\lfloor n/m \rfloor = 3$; $e = 30$ s and λ varies. Threads and processing elements are denoted by (n, m). *Dashed lines* are the upper bounds, *solid lines* are the simulation of the ideal algorithm, and *markers* are empirical results

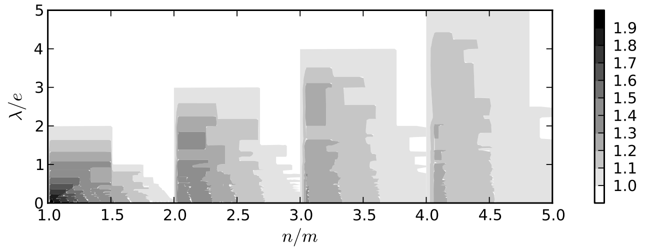


Fig. 14 Relative speedup of the simulated idealized algorithm. Each point in the heat map is colored according to relative speedup over the statically balanced case. $n = 3, 4, \dots, 64$, $m = 2, 3, \dots, n - 1$ and $\lambda/e = 0.01, 0.02, \dots, 5$

see that the upper bound for the relative speedup (the dashed line) is looser when $\lfloor n/m \rfloor > 1$. Figures 12 and 13 also present the results 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 Fig. 14.

To explore the effect of multiple computation phases, we modified EP to use a configurable number of barriers, with fixed phase sizes. Figure 15 shows that the empirical behavior closely matches the approximations (the dashed line) given by Eq. (16) when $\lambda \geq e\lfloor n/m \rfloor$, and by Eq. (25) when $\lambda < e\lfloor n/m \rfloor$.

An interesting feature in Fig. 15 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 (± 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 be such perfect correspondence between the phase length e and the balancing period λ.

5.3 NAS benchmarks

We explored the effects of memory, caching, and various architectural features such as NUMA and hyper-threading,

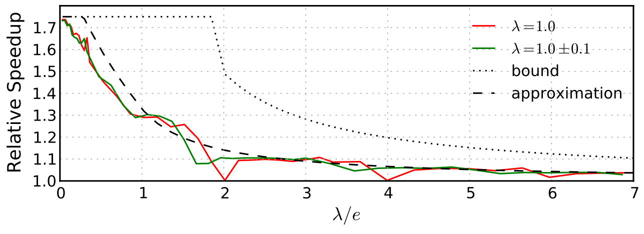


Fig. 15 Relative speedup of UPC EP class C (multiple barriers) with Juggle; $n = 8$ and $m = 7$. *Continuous lines* are the empirical results, the *dashed line* is the relative speedup approximation, and the *dotted line* is the theoretical upper bound

Table 5 Statistics about selected UPC NAS parallel benchmarks (*BM* column) on the *Nehalem*, *Barcelona* (Barc.) and *Tigerton* (Tiger.). *RSS* is the resident set size in GBytes, as reported by Linux. *IB* is the interbarrier time in milliseconds. The speedups are for $n = 8$, $m = 8$ and $n = 16$, $m = 16$ compared to $n = 1$, $m = 1$, except for BT and SP, where we report $n = 16$, $m = 8$ for the speedup of 8 cores because these require a square number of threads

BM	RSS	Speedup over 1 core				IB
		Nehalem		Tiger.	Barc.	
		$n = 8$	$n = 16$	$n = 16$	$n = 16$	
BT.B	1.1	5.5	6.0	4.1	9.9	12.7
CG.C	1.3	5.3	4.3	5.3	10.3	3
EP.C	0.02	8.0	12.8	15.9	15.9	3444
FT.C	7.1	4.3	4.5	5.6	10.5	1854
IS.C	2.3	6.7	7.3	4.8	8.4	28
MG.C	3.5	5.6	5.6	5.0	8.8	14
SP.B	0.4	7.0	6.8	5.0	11.2	5

through a series of experiments with the NAS benchmarks on the three systems shown in Table 3. These systems represent three important architectural paradigms: the *Nehalem* and *Barcelona* are NUMA systems, the *Tigerton* is UMA, and the *Nehalem* is hyperthreaded.

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. See Table 5 for statistics on the chosen benchmarks. We only use benchmarks that are present in the UPC NAS 2.4 implementation; hence we exclude DC, LU, and UA that are present in the OpenMP 3.3 NAS benchmarks. As can be seen in Table 5, all the benchmarks scale, except for CG, MG and SP when going from $n = 8$ to $n = 16$ on the *Nehalem*. This is a consequence of hyperthreading, which adds little to most benchmarks (except EP). We can also see that the speedups on the *Tigerton* are much worse than on the *Barcelona*, due to the *Tigerton*'s front-side bus architecture.

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

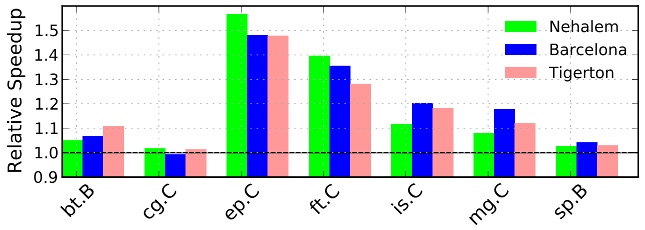


Fig. 16 Relative speedup of UPC NAS benchmarks with Juggle; $n = 16$, $m = 12$ and $\lambda = 100$ ms

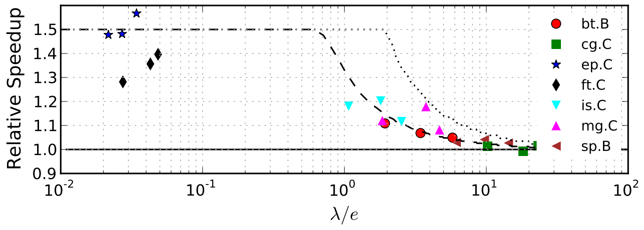


Fig. 17 The effect of λ/e on the relative speedup of UPC NAS benchmarks balanced by Juggle; $n = 16$, $m = 12$ and $\lambda = 100$ ms. The *dashed line* is the relative speedup approximation, the *dotted line* is the theoretical upper bound, and *markers* are empirical results from three systems, *Barcelona*, *Tigerton*, and *Nehalem*

Table 6 Running times in seconds for UPC NAS benchmarks, statically balanced, and with Juggle; $n = 16$, $m = 12$ and $\lambda = 100$ ms

BM	Nehalem		Tigerton		Barcelona	
	Static	Juggle	Static	Juggle	Static	Juggle
BT.B	133.0	126.8	398.0	359.0	223.5	209.2
CG.C	77.3	76.0	174.8	172.6	98.5	99.2
EP.C	38.0	24.3	60.0	40.6	48.0	32.4
FT.C	103.0	73.8	182.3	142.2	116.7	86.0
IS.C	36.0	32.3	85.0	72.0	50.5	42.0
MG.C	26.5	24.5	67.0	59.8	33.0	28.0
SP.B	95.5	93.0	222.8	216.4	141.0	135.4

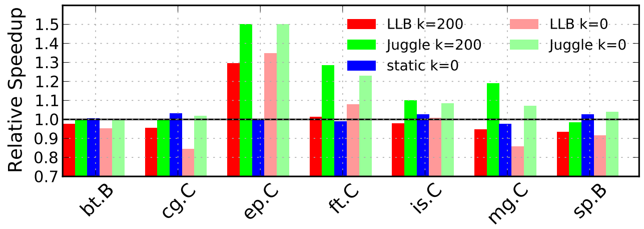


Fig. 18 Relative speedup of OpenMP NAS benchmarks balanced by Juggle and LLB on the *Barcelona* system with $n = 16$, $m = 12$ and $\lambda = 100$ ms. Relative speedup is measured against the statically balanced case with $k = 200$ ms

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 Fig. 9).

In Fig. 16 and Table 6 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 or spins on 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 17 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 Sects. 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.

It is evident from Fig. 17 that if we could decrease λ for the finer grained benchmarks such as BT and SP, we could potentially improve the relative speedup. However, to obtain significant improvements would mean reducing λ by an order of magnitude, to around 10 ms, and at that value performance degrades because of OS interference and other effects, as we show in Sect. 5.5.

UPC implements either spin-yield or busy-wait, so threads do not sleep on barriers, which means that the Linux Load Balancer (LLB) will not balance UPC applications. By contrast, in OpenMP threads on a barrier first yield for some time period, k, and then sleep (i.e. a combination of spin-yield and blocking). If k is small then OpenMP applications can to some extent be balanced by LLB.

Figure 18 and Table 7 show results of running the OpenMP NAS benchmarks on the *Barcelona* system[5] with $k = 200$ ms (the default) and $k = 0$, meaning that threads immediately sleep when they reach the barrier. LLB has some beneficial effect on EP, giving a 35 % relative speedup when $k = 0$ and a 30 % relative speedup when $k = 200$ ms. 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

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

Table 7 Running times in seconds for OpenMP NAS benchmarks on the *Barcelona* system with $n = 16$, $m = 12$ and $\lambda = 100$ ms

BM	Static		LLB		Juggle	
	$k = 0$	$k = 200$	$k = 0$	$k = 200$	$k = 0$	$k = 200$
BT.B	98.6	99.0	104.0	101.4	98.8	99.0
CG.C	104.3	107.4	127.3	112.6	105.6	107.4
EP.C	42.0	42.0	31.1	32.4	28.0	28.0
FT.C	72.7	72.0	66.7	71.0	58.6	56.0
IS.C	39.5	40.5	40.2	41.3	37.4	36.8
MG.C	40.0	39.0	45.5	41.2	36.4	32.8
SP.B	114.0	117.0	127.7	125.3	112.6	119.0

of most benchmarks, and the default synchronization with $k = 200$ ms performs slightly better than pure sleep. The benefits of Juggle are not dependent on how synchronization is implemented, which means that the more efficient spin-yield can be used instead of blocking.

OpenMP supports dynamic threading with work-stealing through its *tasking* interface, which enables OpenMP applications to better adjust to varying core counts. However, the NAS benchmarks we evaluate (OpenMP version 3.3), do not use tasking, but are written with a one-thread per core assumption. To evaluate tasking, we would have to modify the benchmarks, which violates our goal of developing an extrinsic load balancer that requires no application code changes. This is an important constraint because there are many applications written with the one-thread per core assumption and many runtimes do not even support work-stealing, e.g., UPC and MPI.[6]

5.4 Multiprogrammed environments

We tested Juggle in an unpredictable, multiprogrammed environment that, although simple, is nevertheless challenging for load balancing in the SPMD model. While running each of the UPC NAS benchmarks individually, we also ran two unrelated "cpu-hogs," single-threaded processes that use no memory or I/O and are compute-intensive. Each cpu-hog cycles continually between sleeping for some random time from 0 to 5 s, and computing for some random time from 5 to 10 s. We set one of the cpu-hogs to have a higher priority (`nice` −3), which means it will get twice the CPU bandwidth when sharing with a thread of normal priority. The cpu-hogs could represent system daemons or user-initiated processes.

We carried out the multiprogramming experiments on the *Nehalem* system with hyper-threading disabled. As can be

[6]We have obtained good results with proactive load balancing on the MPI versions of the benchmark in previous work [9], although we do not include the results in this paper.

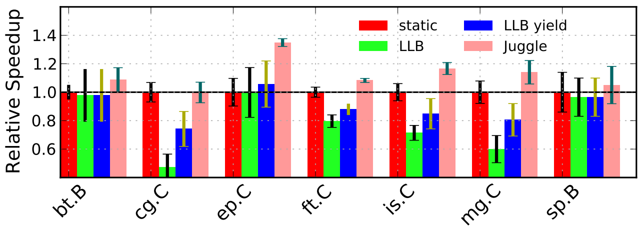


Fig. 19 Relative speedup of UPC NAS benchmarks with Juggle, LLB and LLB with polite synchronization (spin-yield), in an unpredictable multiprogrammed environment; $\lambda = 100$ ms and $n = 8$ and $m = 8$, except for BT and SP where $n = 4$, $m = 4$. The error bars show a 95 % confidence interval

seen in Table 5, hyper-threading makes little difference to most benchmarks, and in the case of FT and SP, actually slows them down. For each benchmark, we ran 8 threads on 8 cores, except for BT and SP, where we ran 16 threads on 8 cores, because those benchmarks require a square number of threads. Because the randomness of the cpu-hogs introduces noise into the experiments, we ran each benchmark 25 times.

As can be seen in Fig. 19, Juggle works effectively in this unpredictable, multiprogrammed environment, giving performance that is at least as good as static balancing, and often better (in the best case, about 35 % better for EP). Although the environment is unpredictable, it is not as deleterious for static balancing as it would be if the cpu-hogs did not sleep. The cpu-hogs are not pinned to any cores, and because they sleep Linux can schedule them on different cores every time they wake up. This means the slowdowns do not all happen to the same thread and the impact is spread out and less extreme than it might be otherwise.

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.

We can improve the performance of LLB by using a different synchronization mechanism. In addition to the default busy-wait where threads simply spin in a tight loop, UPC also has a *polite* spin-yield synchronization mechanism. This is the default when threads are oversubscribed, as is the case with BT and SP, where we used a square number of threads, 16, on 8 cores. When a benchmark starts under LLB, it could end up with two threads on a single core, in which case *polite* synchronization is beneficial. As can be seen in Fig. 19, this improves LLB results, for example, the performance of CG almost doubles. However, the improvements are still not sufficient to bring LLB up to the same level as static balancing.

A striking feature of the results is the performance variation, as shown by the error bars in Fig. 19, indicating the

Table 8 Maximum percentage variation for UPC NAS benchmarks (*BM* column) with Juggle and LLB in an unpredictable multiprogrammed environment; $\lambda = 100$ ms and $n = 8$ except for BT and SP where $n = 16$. The difference of maximum percentage variation with respect to Juggle is in parenthesis

BM	Static	LLB	LLB yield	Juggle
BT.B	9 % (−5 %)	– (–)	35 % (21 %)	14 %
CG.C	14 % (2 %)	36 % (24 %)	32 % (20 %)	12 %
EP.C	17 % (13 %)	30 % (26 %)	29 % (25 %)	4 %
FT.C	6 % (5 %)	10 % (9 %)	8 % (7 %)	1 %
IS.C	11 % (6 %)	12 % (7 %)	24 % (19 %)	5 %
MG.C	15 % (2 %)	26 % (13 %)	30 % (17 %)	13 %
SP.B	27 % (7 %)	– (–)	24 % (4 %)	20 %

95 % confidence interval. Except for one, all the benchmarks running under Juggle have a variance lower than or similar to that when running with static load balancing. But when compared with the benchmarks running under LLB, the benchmarks running with Juggle exhibit much lower variance. To illustrate this point more clearly, we show the maximum percentage variation for each benchmark across the 25 runs in Table 8. The difference of maximum percentage variations obtained with static load balancing and Juggle is up to 13 %. Only BT.B benchmark has lower variation under static load balancing than under Juggle (a difference of 5 points). On the other hand, the maximum percentage variations have a difference up to 26 % between LLB and Juggle, and up to 25 % between LLB with spin-yield and Juggle. This result points to another useful feature of Juggle: it enhances performance predictability, and should make it easier to tune applications (like in [5]), even in noisy environments.

In another set of multiprogramming tests, we explored the effect of running two benchmarks simultaneously on the *Nehalem* system, each balanced by an independent instance of Juggle. We paired benchmarks that have very similar run times so if there is any interference, the impact will be maximized. For each benchmark we ran 8 threads on same 7 cores. This is a good test case, because Juggle should be able to balance such mismatches even when multiple benchmarks are running simultaneously.

Figure 20 shows that there is no negative interference between multiple instances, even though there is a complete overlap of resources, and each Juggle instance has no awareness that there is any other instance running. Juggle improves performance over static balancing for *every* benchmark, even those that synchronize frequently, for which we would not expect a performance improvement on a non-dedicated system. Part of this can be attributed to the fact that when multiple benchmarks are running together, they run more slowly and hence the synchronization interval is greater. LLB appears to help very slightly compared to static

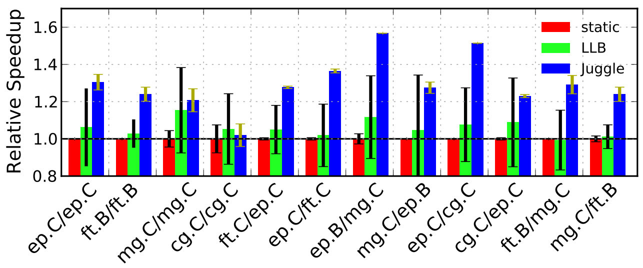


Fig. 20 Relative speedup of UPC NAS benchmarks with Juggle and LLB when running two independent instances of benchmarks simultaneously; $n = 8$, $m = 7$, and $\lambda = 100$ ms. For each label *bm1/bm2*, the value shown is for *bm1* when running simultaneously with *bm2*. The error bars show a 95 % confidence interval

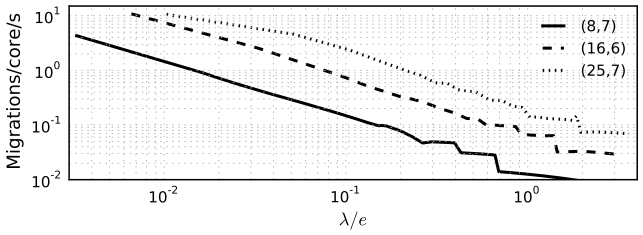


Fig. 21 Migration rate for Juggle balancing UPC EP class C on the *Nehalem* system. Thread and core counts are given as (n, m). $e = 30$ s and λ varies

balancing, but the variation is very large, producing runs that are often considerably worse than static balancing.

5.5 Evaluating the overhead

We measured the compute time taken by Juggle when balancing EP on the *Nehalem* system, with $n = 8$, $\lambda = 100$ ms, and inter-domain migrations enabled. When $m = 8$ there are no migrations and the compute time for Juggle is about 20 µs 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 µs 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 21 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 µs on the *Nehalem* system.

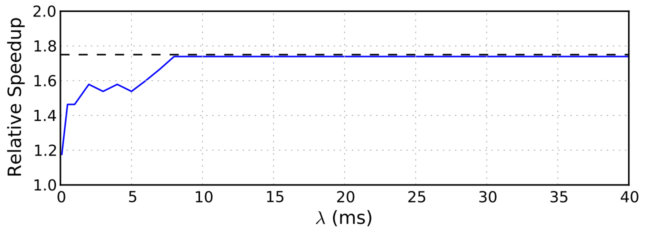


Fig. 22 Effect of decreasing λ on EP class C, with $n = 8$, $m = 7$. The *dashed line* is the theoretical maximum relative speedup

The theoretical analyzes of Sect. 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 Fig. 22, which shows how performance degrades as λ falls below 10 ms, which is the scheduling interval on this particular system. As λ reduces even further, the 100 µs overhead starts to impact performance, e.g., at 0.5 ms we expect there to be a 20 % decrease in performance due to the 100 µs 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.

6 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 [7, 19], from programming-model extensions (such as work stealing [1, 14, 17]) to hardware prioritization on hyper-threaded architectures [3].

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 [23, 24], 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 [13]), 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 [4]).

In previous work [9], we developed a proactive load-balancing approach to address off-by-one imbalances for SPMD applications, called *Speed Balancing*. Speed-balancing uses 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). 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, with Juggle, we have carefully constructed and analyzed an algorithm that guarantees the best dynamic load balance, which we have confirmed both theoretically and with simulations 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 [20] was originally designed to migrate threads twice a second, even if the imbalance in run queues was only one. More recently, Li et al. [15] 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 [18] 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, 8]. Gang Scheduling is beneficial on large-scale distributed systems where

OS-jitter is problematic [12], but it does not address the problem of off-by-one imbalances. Gang Scheduling can then be regarded as complementary to proactive scheduling, which cannot balance very fine grained applications.

7 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 µs, 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). This would require either tighter integration with the OS scheduler or changes to the runtime, e.g., to provide hooks for tracking synchronization operations. It is possible that reactive balancing will be better for fine-grained parallelism, because the balancing events will coincide with synchronization, and so will potentially introduce less overhead. A hybrid approach of using a periodically-triggered proactive balancer together with a reactive balancer might give the best of both worlds.

HPC applications that run on clusters of distributed-memory nodes often require global load balancers. Using Juggle locally can potentially make the work of a global balancer easier, because the global balancer would only need to ensure a uniform distribution of threads across nodes, and let Juggle manage the local balance. Our theory is also applicable to any global balancer that migrates threads locally on a shared-memory node, and we intend to extend the theory to incorporate the effects of non-local memory access and migration costs, which will make it generally applicable to load balancing across NUMA nodes and distributed-memory clusters.

Another aspect of relevance to HPC systems concerns the effects of OS jitter [12, 22]. In large-scale systems, even if jitter occurs very infrequently on a single node, with many nodes the probability of at least one being disrupted by jitter at any given time becomes high. With pinned threads, jitter on a single core in a node could have a very high impact, especially for applications that are written with the one-thread-per-core assumption. If the sources of jitter are sufficiently long-lived so that dynamic balancing can be effective on a shared-memory node, the improvement could be significant.

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. Nonetheless, 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, although they are often used by parallel applications for performance monitoring and tuning.

Another simplifying assumption we make is that all threads take the same time to execute on a dedicated processor (i.e., uniformly sized tasks). This assumption allows us to address extrinsic imbalances for a limited, yet important class of SPMD applications, such as those with uniformly partitioned input data sets. Our work also provides a basis from which to explore future extensions for dynamic load balancing of irregular applications. Simply measuring elapsed time will not be sufficient to determine thread progress for irregular applications. Accurately measuring progress will likely require runtime or code modifications (which we have tried to avoid), so that Juggle can query the threads about their progress. Apart from implementation issues, the theory will also become more complex, e.g., the analysis may depend on the distribution of task sizes, and the optimal algorithm may differ for different distributions.

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.

Our analysis is a step towards a deeper theoretical understanding of dynamic load balancing for SPMD parallelism. Much theoretical work remains to be done. The analysis needs to be extended to cover the case of multiple computation phases with 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. Finally, the issue of intrinsically imbalanced SPMD applications (i.e., irregularly sized tasks) needs to be explored. We hope to develop these extensions in the future to have a much more comprehensive picture of proactive and reactive load balancing in SPMD applications.

Acknowledgements The authors acknowledge the support of DOE 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.

References

1. Blumofe, R.D., Papadopoulos, D.: The performance of work stealing in multiprogrammed environments. ACM SIGMETRICS Perform. Eval. Rev. **26**(1), 266–267 (1998)
2. Boneti, C., Gioiosa, R., Cazorla, F.J., Corbalán, J., Labarta, J., Valero, M.: Balancing HPC applications through smart allocation of resources in MT processors. In: Proc. 22nd IEEE Int'l Symposium on Parallel and Distributed Processing, pp. 1–12 (2008)
3. Boneti, C., Gioiosa, R., Cazorla, F.J., Valero, M.: A dynamic scheduler for balancing HPC applications. In: Proc. 2008 ACM/IEEE Conference on Supercomputing, pp. 41:1–41:12, (2008)
4. Cedo, F., Cortes, A., Ripoll, A., Senar, M., Luque, E.: The convergence of realistic distributed load-balancing algorithms. Theory Comput. Syst. **41**(4), 609–618 (2007)
5. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: Proc. 2008 ACM/IEEE Conference on Supercomputing, pp. 4:1–4:12, (2008)
6. Feitelson, D.G., Rudolph, L.: Gang scheduling performance benefits for fine-grain synchronization. J. Parallel Distrib. Comput. **16**, 306–318 (1992)
7. Fonlupt, C., Marquet, P., luc Dekeyser, J.: Data-parallel load balancing strategies. Parallel Comput. **24**(11), 1665–1684 (1998)
8. Gupta, A., Tucker, A., Urushibara, S.: The impact of operating system scheduling policies and synchronization methods on performance of parallel applications. ACM SIGMETRICS Perform. Eval. Rev. **19**(1) (1991)
9. Hofmeyr, S., Iancu, C., Blagojević, F.: Load balancing on speed. In: Proc. 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 147–158 (2010)
10. Hofmeyr, S., Colmenares, J.A., Iancu, C., Kubiatowicz, J.: Juggle: proactive load balancing on multicore computers. In: Proc. 20th ACM Int'l Symposium on High Performance and Distributed Computing, pp. 3–14 (2011)
11. Iancu, C., Hofmeyr, S., Blagojevic, F., Zheng, Y.: Oversubscription on multicore processors. In: Proc. 2010 IEEE Int'l Symposium on Parallel and Distributed Processing, pp. 1–11 (2010)
12. Jones, T., Dawson, S., Neely, R., Tuel, W., Brenner, L., Fier, J., Blackmore, R., Caffrey, P., Maskell, B., Tomlinson, P., Roberts, M.: Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In: Proc 2003 ACM/IEEE Conference on Supercomputing, p. 10 (2003)
13. Khan, Z., Singh, R., Alam, J., Kumar, R.: Performance analysis of dynamic load balancing techniques for parallel and distributed systems. Int. J. Comput. Netw. Secur. **2**, 2 (2010)
14. Kukanov, A., Voss, M.J.: The foundations for scalable multi-core software in Intel Threading Building Blocks. Intel Technol. J. **11**(4) (2007)
15. Li, T., Baumberger, D., Hahn, S.: Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In: Proc. 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2009)
16. Nishtala, R., Yelick, K.: Optimizing collective communication on multicores. In: Proc. 1st USENIX Workshop on Hot Topics in Parallelism (2009)
17. Olivier, S., Prins, J.: Scalable dynamic load balancing using UPC. In: Proc. 37th Int'l Conference on Parallel Processing, pp. 123–131 (2008)
18. Ousterhout, J.: Scheduling techniques for concurrent systems. In: Proc. 3rd Int'l Conference on Distributed Computing Systems, pp. 22–30 (1982)
19. Plastino, A., Ribeiro, C.C., Rodriguez, N.: Developing SPMD applications with load balancing. Parallel Comput. **29**(6), 743–766 (2003)
20. Roberson, J.: ULE: A modern scheduler for FreeBSD. In: Proc. USENIX BSD Conference (BSDCON), pp. 17–28 (2003)
21. Sancho, J.C., Kerbyson, D.J., Lang, M.: Characterizing the impact of using spare-cores on application performance. In: Proc. 16th Int'l Euro-Par Conference on Parallel Processing, Part I. LNCS, vol. 6271, pp. 74–85 (2010)
22. Tsafrir, D., Etsion, Y., Feitelson, D.G., Kirkpatrick, S.: System noise, OS clock ticks, and fine-grained parallel applications. In: Proc. 19th ACM Annual Int'l Conference on Supercomputing (ICS), pp. 303–312 (2005)
23. Willebeek-LeMair, M., Reeves, A.: Strategies for dynamic load balancing on highly parallel computers. IEEE Trans. Parallel Distrib. Syst. **4**(9) (1993)
24. Xu, C., Lau, F.C.: Load Balancing in Parallel Computers: Theory and Practice. Kluwer Academic, Dordrecht (1997)

Steven Hofmeyr is a researcher at Lawrence Berkeley National Laboratory. Previously he was founder and Chief Scientist at cyber-security company Sana Security, which was acquired by AVG in 2009. He received a Ph.D. in Computer Science from the University of New Mexico in 1999. His research interests include: information security, modeling of complex systems, such as the Internet and financial markets; particularly understanding the impact of policies and regulation for controlling cyber-threats on a large scale; future operating systems for manycore architectures; and scheduling and load balancing for parallel programming. He was named one of Infoworld's top 12 Innovators of the Year in 2004, and received the MIT Technology Review TR100 Innovators of the Year award in 2003 for Advances in Network Security.

Juan A. Colmenares is a postdoctoral scholar in the Parallel Computing Laboratory (Par Lab) at the University of California, Berkeley. He received a Ph.D. degree in Computer Engineering from the University of California, Irvine, in 2009. He also obtained a M.Sc. degree in Applied Computing in 2001 and a B.Sc. degree in Electrical Engineering in 1997, both from the University of Zulia, Venezuela. His research interests include operating systems for many-core architectures, real-time distributed computing, and multimedia applications.

Costin Iancu received in 2001 the PhD in Computer Science from University of California at Santa Barbara. He is currently a staff scientist at the Lawrence Berkeley National Laboratory. His research interests lie in the area of programming languages, with emphasis on runtime and operating systems support for parallel programming languages. He is probably best known for the implementation of the Berkeley Unified Parallel C (BUPC) compiler.

John Kubiatowicz received a double B.S. in Electrical Engineering and Physics, 1987, M.S. in Electrical Engineering and Computer Science, 1993, and a PhD in Electrical Engineering and Computer Science. Minor in Physics, 1998, all from M.I.T. He joined the faculty of EECS at UC Berkeley in 1998. Current research includes exploring the design of extremely-wide area storage utilities and developing secure protocols and routing infrastructures that provide privacy, security, and resistance to denial of service, while still allowing the caching of data anywhere, anytime. Also, exploring the space of Introspective Computing, namely systems which perform continuous, on-line adaptation. Applications include on-chip tolerance of flaky components and continuous optimization to adapt to server failures and denial of service attacks. Honors and awards include the Diane S. McEntyre Award for Excellence in Teaching, 2003, Scientific American 50, 2002, MoundsView High School Distinguished Alumni Award, 2001, Berkeley IT Award for Excellence in Undergraduate CS Teaching, 2000, Presidential Early Career Award for Scientists and Engineers (PECASE), 2000, George M. Sprowls Award for best PhD thesis in EECS at MIT, 1998, IBM Graduate Fellowship, 1992–1994, and Best Paper, International Conference on Supercomputing, 1993.