Limits of Work-Stealing Scheduling

Željko Vrba[1,2], Håvard Espeland[1,2], Pål Halvorsen[1,2], and Carsten Griwodz[1,2]

[1] Simula Research Laboratory, Oslo
[2] Department of Informatics, University of Oslo

Abstract. The number of applications with many parallel cooperating processes is steadily increasing, and developing efficient runtimes for their execution is an important task. Several frameworks have been developed, such as MapReduce and Dryad, but developing scheduling mechanisms that take into account processing *and* communication requirements is hard. In this paper, we explore the limits of work stealing scheduler, which has empirically been shown to perform well, and evaluate load-balancing based on graph partitioning as an orthogonal approach. All the algorithms are implemented in our Nornir runtime system, and our experiments on a multi-core workstation machine show that the main cause of performance degradation of work stealing is when very little processing time, which we quantify exactly, is performed per message. This is the type of workload in which graph partitioning has the potential to achieve better performance than work-stealing.

1 Introduction

The increase in CPU performance by adding multiple execution units on the same chip, while maintaining or even lowering *sequential* performance, has accelerated the importance of parallel applications. However, it is widely recognized that shared-state concurrency, the prevailing parallel programming paradigm on workstation-class machines, is hard and non-intuitive to use [1]. Message-passing concurrency is an alternative to shared-state concurrency, and it has for a long time been used in distributed computing, and now also in modern parallel program frameworks like MapReduce [2], Oivos [3], and Dryad [4]. However, message passing frameworks also have an increasing importance on multi-core architectures, and such parallel program runtimes are being implemented and ported to single multi-core machines [5–8].

In this context, we have experimented with different methods of scheduling applications defined by process graphs, also named process networks, which explicitly encode parallelism and communication between asynchronously running processes. Our goal is to find an efficient scheduling framework for these multi-core parallel program runtimes. Such a framework should support a wide range of complex applications, possibly using different scheduling mechanisms, and use available cores while taking into account the underlying processor topology, process dependencies and message passing characteristics.

E. Frachtenberg and U. Schwiegelshohn (Eds.): JSSPP 2009, LNCS 5798, pp. 280–299, 2009.

Particularly, in this paper, we have evaluated the *work-stealing* load-balancing method [9], and a method based on *graph partitioning* [10], which balances the load across CPUs, and reduces the amount of inter-CPU communication as well as the cost of migrating processes. Both methods are implemented and tested in Nornir [8], which is our parallel processing runtime for executing parallel programs expressed as Kahn process networks [11].

It has been theoretically proven that the work-stealing algorithm is optimal for scheduling *fully-strict* (also called *fork-join*) computations [12]. Under this assumption, a program running on P processors, 1) achieves P-fold speedup in its parallel part, 2) using at most P times more space than when running on 1 CPU. These results are also supported by experiments [13, 14]. Saha et al. [14] have presented a run-time system aimed towards executing fine-grained concurrent applications. Their simulations show that work-stealing scales well on up to 16 cores, but they have not investigated the impact of parallelism granularity on application performance. Investigation of this factor is one of the contributions of this paper.

In our earlier paper [8] we have noted that careful static assignment of processes to CPUs can match the performance of work-stealing on finely-granular parallel applications. Since static assignment is impractical for large process networks, we have also evaluated an automatic scheduling method based on graph partitioning by Devine et al. [10], which balances the load across CPUs, and reduces the amount of inter-CPU communication as well as the cost of process migration. The contributions of this paper on this topic are two-fold: 1) showing that graph partitioning can sometimes match work-stealing when workload is very fine-grained, and 2) an investigation of *variation* in running time, an aspect neglected by the authors.

Our main observations are that work stealing works nice for a large set of workloads, but orthogonal mechanisms should be available to address the limitations. For example, if the work granularity is small, a graph partitioning scheme should be available, as it shows less performance degradation compared to the work-stealing scheduler. The graph-partitioning scheme succeeds in decreasing the amount of inter-CPU traffic by a factor of up to 7 in comparison with the work-stealing scheduler, but this reduction has no influence on the application running time. Furthermore, applications scheduled with graph-partitioning methods exhibit unpredictable performance, with widely-varying execution times between consecutive runs.

The rest of this paper is structured as follows: in section 2 we describe the two load-balancing strategies and compare our work-stealing implementation to that of Intel's Threading Building Blocks (TBB),[1] which includes an industrial-strength work-stealing implementation. In section 3 we describe our workloads, methodology and present the main results, which we summarize and relate to the findings of Saha et al. [14] in section 4. We conclude in section 5 and discuss broader issues in appendices.

[1] `http://www.threadingbuildingblocks.org/`

2 Dynamic Load-Balancing

We shall describe below the work-stealing and graph-partitioning scheduling methods. We assume an $m : n$ threading model where m user-level **processes** are multiplexed over n kernel-level **threads**, with each thread having its own run queue of ready processes. The affinity of the threads is set such that they execute on different CPUs. While this eliminates interference between Nornir's threads, they will nevertheless share their assigned CPU with other processes in the system, subject to standard Linux scheduling policy.[2]

2.1 Work Stealing

A work-stealing scheduler maintains for each CPU (kernel-level thread) a queue of ready processes waiting for access to the processor. Then, each thread takes ready processes from the *front* of its own queue, and also puts unblocked processes at the front of its queue. When the thread's own run queue is empty, the thread steals a process from the *back* of the run-queue of a randomly chosen thread. The thread loops, yielding the CPU (by calling `sched_yield`) before starting a new iteration, until it succeeds in either taking a process from its own queue, or in stealing a process from another thread. All queue manipulations run in constant-time ($O(1)$), independently of the number of processes in the queues.

The reasons for accessing the run queues at different ends are several [15]: 1) it reduces contention by having stealing threads operate on the opposite end of the queue than the thread they are stealing from; 2) it works better for parallelized divide-and-conquer algorithms which typically generate large chunks of work early, so the older stolen task is likely to further provide more work to the stealing thread; 3) stealing a process also migrates its future workload, which helps to increase locality.

The original work-stealing algorithm uses non-blocking algorithms to implement queue operations [9]. However, we have decided to simplify our scheduler implementation by protecting each run queue with its own lock. We believed that this would not impact scalability on our machine, because others [14] have reported that even a *single, centralized queue* protected by a *single, central lock* does not hurt performance on up to 8 CPUs, which is a decidedly worse situation for scalability as the number of CPUs grows. Since we use locks to protect the run queues, and our networks are static, our implementation does not benefit from the first two advantages of accessing the run queues at different ends. Nevertheless, this helps with increasing locality: since the arrival of a message unblocks a proces, placing it at the front of the ready queue increases probability that the required data will remain in the CPU's caches.

Intel's TBB is a C++ library which implements many parallel data-structures and programming patterns. TBB's internal execution engine is also based on

[2] It is difficult to have fully "idle" system because the kernel spawns some threads for its own purposes. Using POSIX real-time priorities would eliminate most of this interference, but would not represent a realistic use-case.

work-stealing, and it uses a non-blocking queue which employs exponential back-off in case of contention. However, the scheduler is limited to executing only fully-strict computations, which means that a process must run to completion, with the only allowed form of blocking being waiting that children processes exit.[3] Thus, the TBB scheduler is not applicable to running unmodified process networks, where processes can block on message send or receive.

2.2 Graph Partitioning

A graph partitioning algorithm partitions the vertices of a weighted graph into n disjoint partitions of approximately equal weights, while simultaneously minimizing the cut cost.[4] This is an NP-hard problem, so heuristic algorithms have been developed, which find approximate solutions in reasonable time.

We have implemented in Nornir the load-balancing algorithm proposed by Devine et al. [10]. This is one of the first algorithms that takes into account not only load-balance and communication costs, but also costs of process migration. The algorithm observes weights on vertices and edges, which are proportional to the CPU time used by processes and the traffic volume passing across channels. Whenever a significant imbalance in the CPU load is detected, the process graph is repartitioned and the processes are migrated. In our implementation, we have used the state-of-art PaToH library [16] for graph and hypergraph partitioning.

When *rebalancing* is about to take place, the process graph is transformed into an *undirected rebalancing graph*, with weights on vertices and edges set such that the partitioning algorithm minimizes the cost function given by the formula $\alpha t_{comm} + t_{mig}$. Here, α is the number of computation steps performed between two rebalancing operations, t_{comm} is the time the application spends on communication, and t_{mig} is time spent on data migration. Here, α represents a trade-off between good load-balance, small communication and migration costs and rebalancing overheads; see appendix B for a broader discussion in the context of our results.

Constructing the rebalancing graph consists of 4 steps (see also figure 1):

1. Vertex and edge weights of the original graph are initialized according to the collected accounting data.
2. Multiple edges between the same pair of vertices are collapsed into a single edge with weight α times the sum of weights of the original edges.
3. n new, zero-weight nodes, $u_1 \dots u_n$, representing the n CPUs, are introduced. These nodes are fixed to their respective partitions, so the partitioning algorithm will not move them to other partitions.
4. Each node u_k is connected by a *migration edge* to every node v_i iff v_i is a task currently running on CPU k. The weight of the migration edge is set to the cost of migrating data associated with process v_i.

[3] For example, the reference documentation (document no. 315415-001US, rev. 1.13) explicitly warns against using the producer-consumer pattern.

[4] Sum of weights of edges that cross partitions.

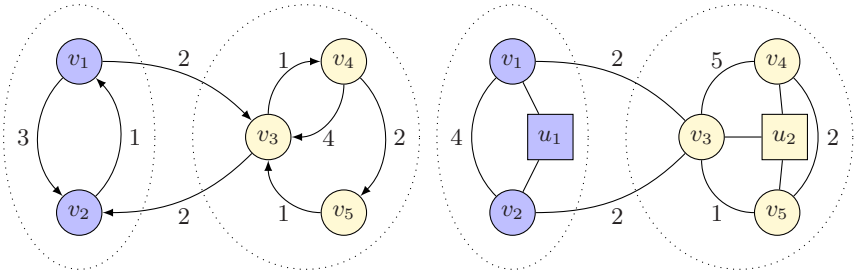


Fig. 1. An example of transforming a process graph into a rebalancing graph with $\alpha = 1$. Current partitions are delimited by ovals and distinguished by nodes of different colors.

For the *initial partitioning* phase, performed before the network starts running, the process graph is transformed into an undirected graph as described above, but with small differences: 1) unit weights are assigned to channels and processes, as the actual CPU usage and communication intensities are not known, and 2) the additional CPU nodes (u_k) and migration edges are omitted. Partitioning this graph gives an initial assignment of processes to CPUs and is a starting point for future repartitions.

Since our test applications have quickly shifting loads, we have implemented a heuristic that attempts to detect load imbalance. The heuristic monitors the idle time τ *collectively* accumulated by all threads, and invokes the load-balancing algorithm when the idle time has crossed a preset threshold. When the algorithm has finished, process and channel accounting data are set to 0, in preparation for the next load-balancing. When a thread's own run-queue is empty, it updates the collective idle time and continues to check the run-queue, yielding (`sched_yield`) between attempts. Whenever *any* thread succeeds in dequeuing a process, it sets the accumulated idle time to 0.

After repartitioning, we avoid bulk migration of processes. It would require locking of all run-queues, migrating processes to their new threads, and unlocking run-queues. The complexity of this task is linear in the number of processes in the system, so threads could be delayed for a relatively long time in dispatching new ready processes, thus decreasing the total throughput. Instead, processes are only reassigned to their new threads by setting a field in their control block, but without physically migrating them. Each thread takes ready processes *only* from its own queue, and if the process's run-queue ID (set by the rebalancing algorithm) matches that of the thread's, the process is run. Otherwise, the process is reinserted into the run-queue to which it has been assigned by the load-balancing algorithm.

3 Comparative Evaluation of Scheduling Methods

We have evaluated the load-balancing methods on several synthetic benchmarks which we implemented and run on Nornir. The programs have been compiled as

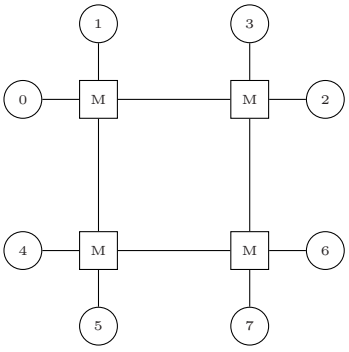


Fig. 2. Topology of the machine used for experiments. Round nodes are cores, square nodes are NUMA memory banks. Each CPU has one memory bank and two cores associated with it.

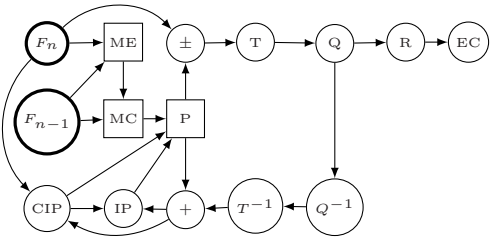
64-bit with GCC 4.3.2 and maximum optimizations (`-m64 -O3 -march=opteron`). Since PaToH is distributed in binary-only form, these flags have no effect on the efficiency of the graph-partitioning code. The benchmarks have been run on an otherwise idle 2.6 GHz AMD Opteron machine with 4 dual-core CPUs (see figure 2), 64 GB of RAM, running linux kernel 2.6.27.3. Each experiment has been repeated 10 consecutive times, with collection of accounting data turned on.

3.1 Description of Workloads

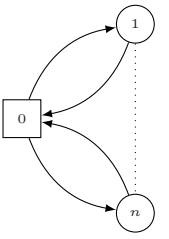
Figure 3(a) shows a process network implementing an **H.264** video-encoder, and it is only a slight adaptation of the encoder block diagram found in [17]. The blocks use an artificial workload consisting of loops which consume the amount of CPU time which would be used by a real codec on average. To gather this data, we have profiled x264, an open-source H.264 encoder, with the *cachegrind* tool and mapped the results to the process graph. Each of P, MC and ME stages has been parallelized as shown in figure 3(b) because they are together using over 50% of the processing time. The number of workers in each of the parallelized stages varies across the set $\{128, 256, 512\}$.

k-means is an iterative algorithm used for partitioning a given set of points in multidimensional space into k groups; it is used in data mining and pattern recognition. To provide a non-trivial load, we have implemented the MapReduce topology as a process network (see Figure 3(c)), and subsequently implemented the Map and Reduce functions to perform the k-means algorithm. The number of processes in each stage has been set to 128, and the workload consists of 300000 randomly-generated integer points contained in the cube $[0, 1000)^3$ to be grouped into 120 clusters.

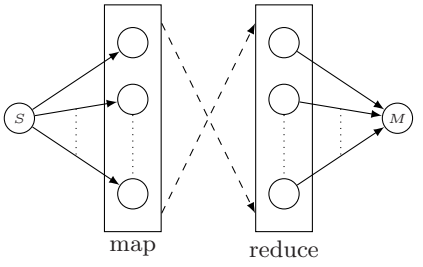
The two **random networks** (see figure 3(e) for an example) are randomly generated directed graphs, possibly containing cycles. To assign work to each process, the workload is determined by the formula nT/d, where n is the number of messages sent by the source, T is a constant that equals ~ 1 second of



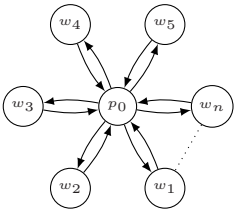
(a) H.264 process network. F_n and F_{n-1} are inputs to the network (current and previous frames)



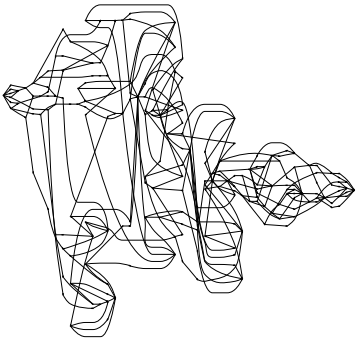
(b) Parallelization of each of P, MC, and ME stages



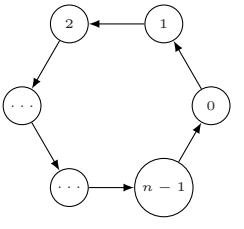
(c) MapReduce topology. Dashed lines denote that each process from the map stage is connected to each process in the reduce stage.



(d) Worker processes connected in a scatter/gather topology to a central process



(e) Random graph with 212 nodes, 333 edges and 13 cycles



(f) Ring topology

Fig. 3. Process networks used for benchmarking

CPU-time, and d is the work division factor. In effect, each message sent by the source (a single integer) carries $w = T/d$ seconds of CPU time. The workload w is distributed in the network (starting from the source process) with each process reading n_i messages from all of its in-edges. Once all messages

are read, they are added together to become the t units of CPU-time the process is to consume before distributing t to its n_o forward out-edges. Then, if a process has a back-edge, a message is sent/received, depending on the edge direction, along that channel. As such, the workload w distributed from the source process will equal the workload w collected by the sink process. Messages sent along back-edges do not contribute to the network's workload; their purpose is solely to generate more complex synchronization patterns. We have used two networks: RND-A has 239 nodes, 364 edges and no cycles; RND-B has 213 nodes, 333 edges and 13 cycles. The work division factor has been varied over the set $\{1, 10, \ldots, 10000, 20000, \ldots, 90000\}$.

In the **ring** benchmark, n processes, $0 \ldots n-1$, are created and connected into a ring topology (see figure 3(f)); in our benchmark we have used $n = m = 1000$. Process 0 sends an initial and measures the time it takes to make m round-trips, while other processes just forward messages and do no other processing otherwise.

The **scatter/gather** network has a single central process (p_0) connected to n worker processes (see Figure 3(d)). The central process scatters m messages to the workers, each performing a set amount of work w for each message. When complete, a message is sent from the worker process to the central process, and the procedure is repeated for a given number of iterations. This topology corresponds to the communication patterns that emerge when several MapReduce instances are executed such that the result of the previous MapReduce operation is fed as the input to the next. We have fixed $n = 50$ and varied the work amount $w \in \{1000, 10000, 20000, \ldots, 10^5\}$.

3.2 Methodology

We use real (wall-clock) time to present benchmark results because we deem that it is the most representative metric since it accurately reflects the real time needed for task completion, which is what the end-users are most interested in. We have also measured system and user times (`getrusage`), but do not use them to present our results because 1) they do not reflect the reduced running time with multiple CPUs, and 2) resource usage does not take into account sleep time, which nevertheless may have significant impact on the task completion time.

In the Kahn process network formalism, processes can use only blocking reads and can wait on message arrival only on a single channel at a time. However, to obtain more general results, we have carefully designed the benchmark programs so that they execute correctly even when run-time deadlock detection and resolution is disabled. This is an otherwise key ingredient of a KPN run-time implementation [8], but it would make our observations less general as it would incur overheads not present in most applications.

The benchmarks have been run using 1, 2, 4, 6, and 8 CPUs under the work-stealing (WS) and graph-partitioning policies (GP). For the GP policy, we have varied the idle-time parameter τ (see section 2.2) from 32 to 256 in steps of 8. This has generated a large amount of raw benchmark data which cannot be fully presented in the limited space. We shall thus focus on two aspects: running time

and amount of local and remote communication, i.e., the number of messages that have been sent between the processes on the same (resp. different) CPU. For a given combination of the number of CPUs, and work division, we compare the WS policy against the *best* GP policy.

We have evaluated the GP policy by varying the idle time parameter τ over a range of values for each given work division d. A point in the plot for the given d corresponds to the experiment with the median running time belonging to the best value of τ. The details of finding the best τ value are somewhat involved, and are therefore given in appendix.

Computing the median over a set of 10 measurements would generate artificial data points.[5] In order to avoid this, we have discarded the last of the 10 measurements before computing the median.

Since a context-switch also switches stacks, it is to be expected that cached stack data will be quickly lost from CPU caches when there are many processes in the network. We have measured that the cost of re-filling the CPU cache through random accesses increases by $\sim 10\%$ for each additional hop on our machine (see figure 2). Due to an implementation detail of Nornir and Linux's default memory allocation policy, which first tries to allocate physical memory from the same node from which the request came, all stack memory would end up being allocated on a single node. Consequently, context-switch cost would depend on the node a process is scheduled on. To average out these effects, we have used the `numactl` utility to run benchmarks under the interleave NUMA (non-uniform memory access) policy, which allocates physical memory pages from CPU nodes in round-robin manner. Since most processes use only a small portion of the stack, we have ensured that their stack size, in the number of pages, is relatively prime to the number of nodes in our machine (4). This ensures that the "top" stack pages of all processes are evenly distributed across CPUs.

3.3 Results

The **ring** benchmark measures scheduling and message-passing overheads of Nornir. Table 1 shows the results for 1000 processes and 1000 round-trips, totalling 10^6 [send $\rightarrow$ context switch $\rightarrow$ receive] transactions. We see that GP performance is fairly constant for any number of CPUs, and that contention over run-queues causes WS performance to drop as the number of CPUs increases from 1 to 2. The peak throughput in the best case (1 CPU, no contention) is $\sim$ 750000 transactions per second. This number is approximately doubled, i.e., transaction cost halved, when accounting mechanisms are turned off. Since detailed accounting data is essential for GP to work, we have run also WS experiments with accounting turned on, so that the two policies can be compared against a common reference point.

The **k-means** program, which executes on a MapReduce topology, is an example of an application that is hard to schedule with automatic graph partitioning.

[5] Median of an *even* number of points is defined as the average of the two middle values.

Table 1. Summary of ring benchmark results (1000 processes and 1000 round-trips). t_n is running time on n CPUs.

	t_1	t_2	t_4	t_6	t_8
GP	1.43	1.65	1.78	1.58	1.71
WS	1.33	2.97	2.59	2.66	2.86

If the idle time parameter τ is too low, repartitioning runs overwhelmingly often, so the program runs *several minutes*, as opposed to 9.4 seconds under the WS method. When the τ is high enough, repartitioning runs only once, and the program finishes in 10.2 seconds. However, the transition between the two behaviours is discontinuous, i.e., as τ is slowly lowered, the behaviour suddenly changes from good to bad. Because of this, we will not consider this benchmark in further discussions.

From figure 4 it can be seen that the WS policy has *the least median running time for most workloads*; it is worse than the GP policy only on the ring benchmark (not shown in the figure; see table 1) and the RND-B network when work division is $d \geq 30000$. At this point, performance of message-passing and scheduling becomes the limiting factor, so the running time increases proportionally with d. On the **H.264** benchmark, the GP policy shows severe degradation in performance as the number of workers and the number of CPUs increases. The root cause of this is the limited parallelism available in the H.264 network; the largest speedup under WS policy is ~ 2.8. Thus, the threads accumulate idle time faster than load-balancing is able to catch-up, so repartitioning and process migration frequently takes place (90 – 410 times per second, depending on the number of CPUs and workers). The former is not only protected by a global lock, but its running time is also proportional with the number of partitions (CPUs) and the number of nodes in the graph, as can be clearly seen in the figure.

As the **RND-B** benchmark is the only case where GP outperforms WS (when $d > 30000$), we have presented further data of interest in figure 5. We can see that WS achieves consistently better peak speedup and at lower d than GP, achieving almost perfect linear scalability with the number of CPUs. Furthermore, we can see that the peak GP speedup has no correlation with peaks and valleys of the proportion of locally sent messages, which constitute over 70% of all message traffic on any number of CPUs. We can also see that the proportion of local traffic under WS decreases proportionally with the increase in the number of CPUs.

Furthermore, we see that WS achieves peak speedup at $d = 100$, which is the largest d value before message throughput starts increasing. Similarly, the peak speedup for GP is at $d = 1000$, which is again at the lower knee of the message throughput curve, except on 8 CPUs where the peak is achieved for $d = 10000$. At $d \geq 30000$, the throughput of messages under the GP policy becomes greater than throughput under the WS policy, which coincides with upper knee of the throughput curve and the point where speedup under GP speedup becomes greater than WS speedup.

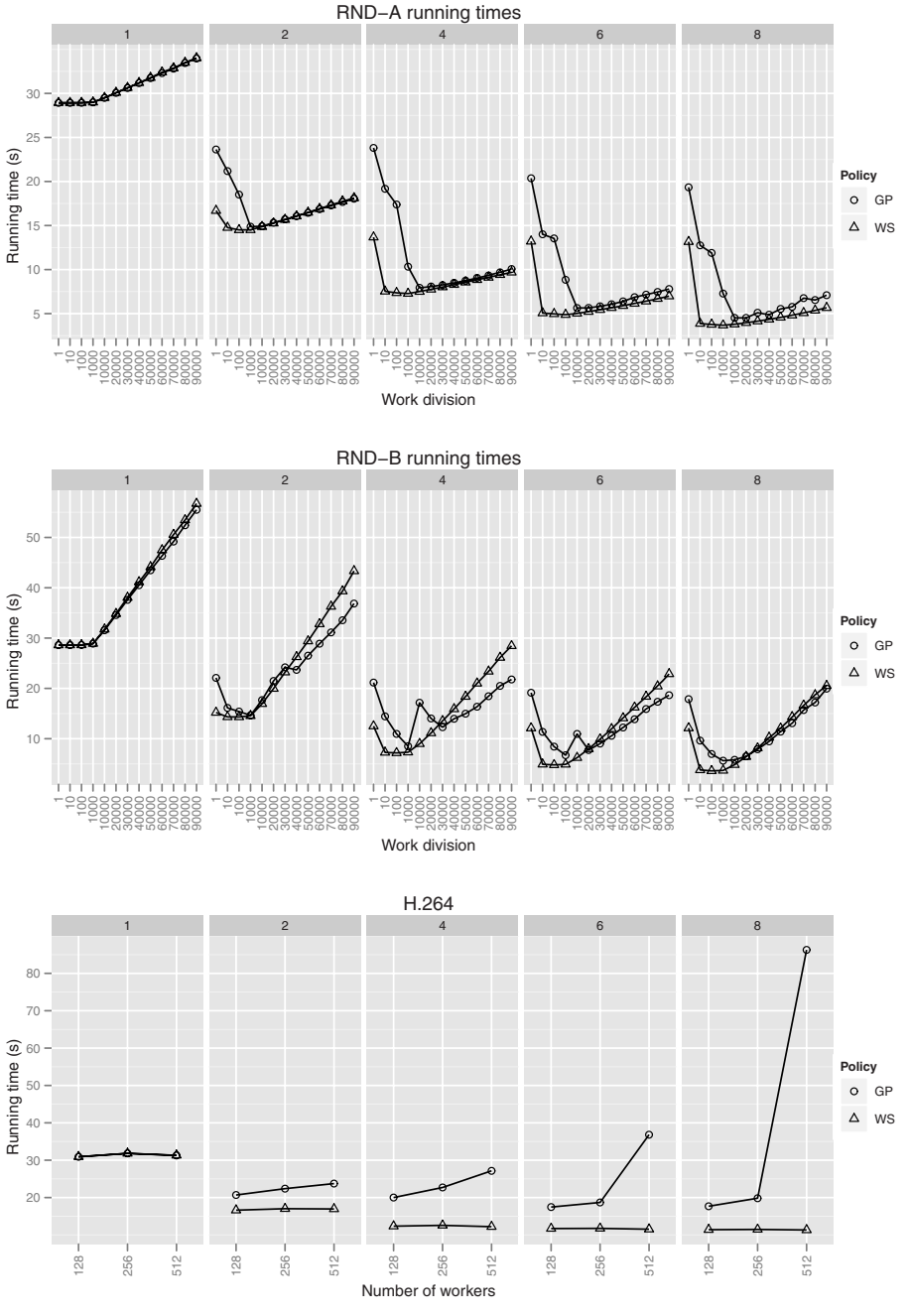


Fig. 4. Median running times for WS and GP policies on 1,2,4,6 and 8 CPUs

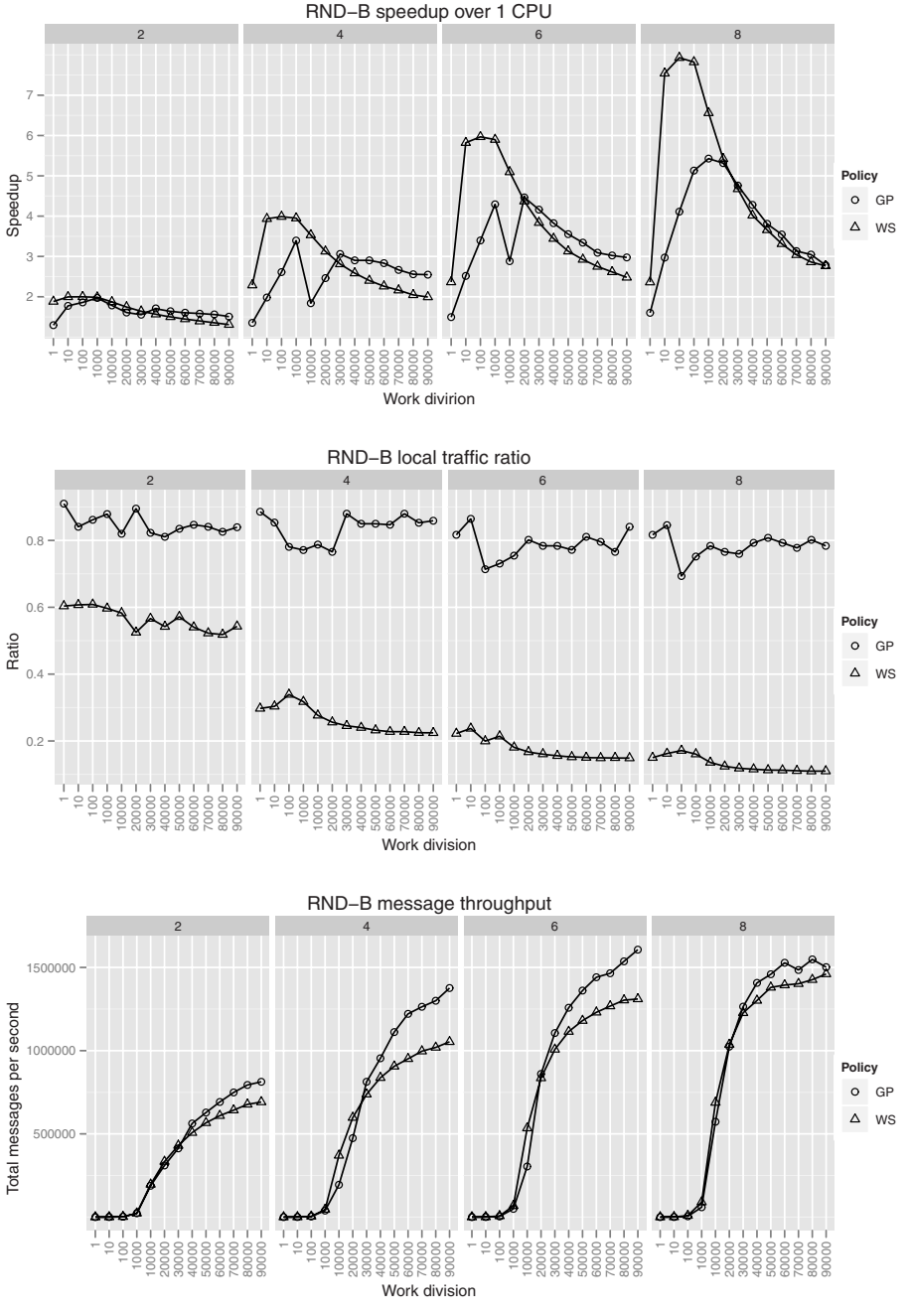


Fig. 5. Further data on RND-B benchmark. The local traffic ratio is calculated as $l/(l+r)$ where l and r are volume of local and remote traffic. Message throughput is calculated as $(l+r)/t$, t being the total running time.

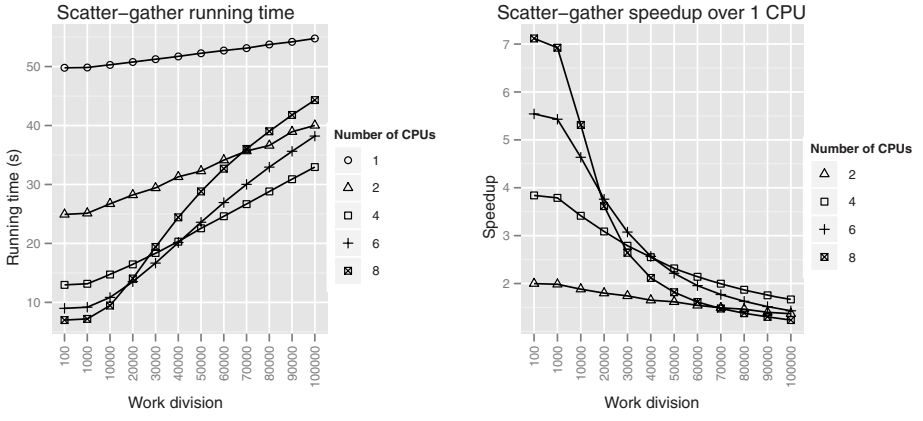


Fig. 6. Performance of the scatter/gather benchmark under WS scheduler. Note that the scale on the x-axis is non-linear and non-uniform.

Figure 6 shows the running time of the **scatter/gather** benchmark for the WS policy on 1 – 8 CPUs. The maximum speedup (7.1 on 8 CPUs) is achieved for $d = 100$ and drops sharply for $d > 1000$; at $d = 20000$ the speedup on 8 CPUs is nearly halved. Thus, we can say that WS performs well as long as each process uses at least $100\mu s$ of CPU time per message, i.e., as long as the computation-to-communication ratio is $\geq\sim 75$. When this ratio is smaller, communication and scheduling overheads overtake, and WS suffers drastic performance degradation.

Figure 7 uses the box-and-whiskers plot[6] to show the distribution of running times and number of repartitionings achieved for all benchmarked τ values of GP policy. The plots show the running time and the number of repartitions for the **RND-B** benchmark on 8 CPUs and $d = 1000$. From the graphs, we can observe several facts:

- WS has undetectable variation in running time (the single line at $\tau = 0$), whereas GP has large variation.
- The number of repartitionings is inversely-proportional with τ, but it has no clear correlation with either variance or median running times.
- When $\tau \geq 72$, the *minimal* achieved running times under the GP policy show rather small variation.

We have thus investigated the relative performance of GP and WS policies, but now considering the *minimal* real running amongst all GP experiments for all values of τ. The graphs (not shown) are very similar to those of figure 5, except that GP speedup is slightly ($< 2\%$ on 8 CPUs) larger.

[6] This is a standard way to show the distribution of a data set. The box's span is from the lower to the upper quartile, with the middle bar denoting the median. Whiskers extend from the box to the lowest and highest measurements that are not outliers. Points denote *outliers*, i.e., measurements that are more than 1.5 times the box's height below the lower or above the upper quartile.

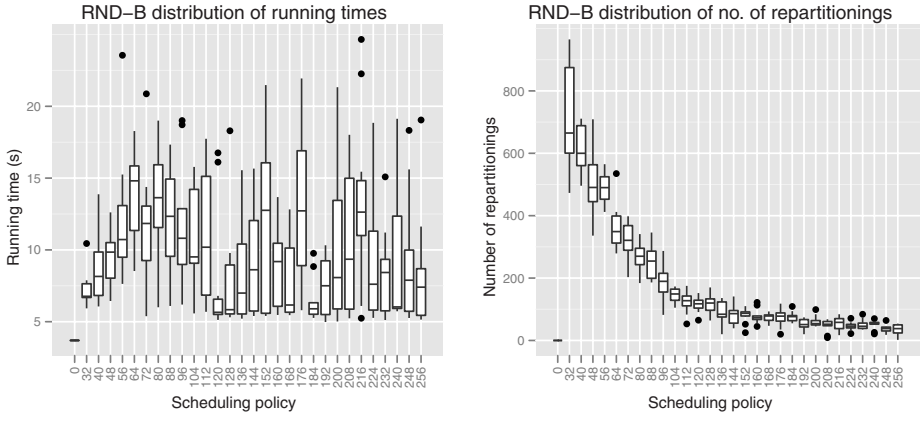


Fig. 7. Illustration of GP variance in running time for RND-B on 8 CPUs and $d = 1000$, which is one step past the WS peak speedup. x-axis is the value of the idle time parameter τ for the GP policy. $\tau = 0$ shows the results for WS.

4 Discussion

In the previous section we have analyzed performance of applications running under graph-partitioning and work-stealing schedulers. WS has been *formally proven* to be optimal *only* for the restricted class of fully-strict computations, but it nevertheless gives best performance also on our benchmark programs, none of which is fully-strict. We can summarize our findings as follows:

- WS gives best performance, with speedup almost linearly proportional with the number of CPUs, *provided* that 1) there is enough parallelism in the network, and 2) the computation to communication ratio, which directly influences scheduling overheads, is at least ~ 75.
- GP and WS show similar patterns in running time, but GP never achieves the same peak speedup as WS.
- There exists an optimal work division d at which the largest speedup is achieved; this granularity is different for WS and GP.
- Increasing d beyond peak speedup leads to a sharp increase in message throughput. This increase quickly degrades performance because message-passing and context-switch overheads dominate the running time.
- GP has large variance in running time; neither the median running time nor its variance is correlated with the idle time parameter τ.
- GP achieves a greater proportion of local traffic than WS, and this ratio falls very slightly with the number of CPUs. The proportion of local traffic under WS falls proportionally with the number of CPUs.
- We have not found any apparent correlation between locality and running time or speedup.

Regarding GP, there are two main obstacles that must be overcome before it can be considered as a viable approach for scheduling general-purpose workloads on multi-processor machines:

- Unpredictable performance, as demonstrated by figure 7.
- The difficulty of automatically choosing the right moment for rebalancing.

As our experiments have shown, monitoring the accumulated idle time over all CPUs and triggering rebalancing after the idle time has grown over a threshold is not a good strategy. Thus, work-stealing should be the algorithm of choice for scheduling general-purpose workloads. Graph partitioning should be reserved for specialized applications using fine-grained parallelism, that in addition have enough knowledge about their workload patterns so that they can "manually" trigger rebalancing.

Saha et al. [14] emphasize that fine-grained parallelism is important in large-scale CMP design, but they have not attempted to *quantify* parallelism granularity. By using a cycle-accurate simulator, they investigated the scalability of work-stealing on a CPU having up to 32 cores, where each core executes 4 hardware threads round-robin. The main finding is that contention over run-queues generated by WS can limit, or even worsen, application performance as new cores are added. In such cases, applications perform better with static partitioning of load with stealing turned off. The authors did not describe how did they partition the load across cores for experiments with work-stealing disabled.

We deem that their results do not give a full picture about WS performance, because contention depends on three additional factors, neither of which is discussed in their paper, and all of which can be used to reduce contention. Contention can be reduced by 1) overdecomposing an application, i.e., increasing the total *number of processes* in the system proportionally with the number of CPUs; by 2) decreasing the number of CPUs to match the *average parallelism* available in the application, which is its intrinsic property; or 3) by increasing the *amount of work* a process performs before it blocks again. The first two factors decrease the probability that a core will find its run-queue empty, and the third factor increases the proportion of useful work performed by a core, during which it does not attempt to engage in stealing.

Indeed, our H.264 benchmark shows that even when the average parallelism is low (only 2.8), the WS running time on 6 and 8 cores does not increase relative to that on 4 CPUs, thanks to overdecomposition. If there were any scalability problems due to contention, the H.264 benchmark would exhibit slowdown similar to that of the ring benchmark.

5 Conclusion and Future Work

In this paper, we have experimentally evaluated performance of two load-balancing algorithms: work-stealing and an algorithm by Devine et al., which is based on graph-partitioning. We have used as the workload a set of synthetic message-passing applications described as directed graphs. Our experimental results confirm the previous results [13, 14] which have reported that WS leads

to almost linear increase in performance given enough parallelism, and expand those results by identifying limitations of WS. We have and experimentally found the lower threshold of computation to communication ratio (~ 75), below which the WS performance drops sharply. GP suffers the same performance degradation, but the overall application performance may (depending on the exact workload) be slightly better in this case than under WS. The presented numbers are specific to our implementation; we would expect the threshold of computation to communication ratio to increase under less-efficient implementations of work-stealing, and vice-versa.

GP never achieved the same peak speedup as WS, but this is not its biggest flaw. The main problem with GP is its instability – running times exhibit a great variance, and the ratio of worst and best running time can be more than 4, as can be seen in figure 7. In our research group, we are currently investigating alternative approaches to load-balancing, which would yield results that are more stable than those obtained with today's methods based on graph-partitioning.

As opposed to the large variance of running time under GP, the proportion of local traffic in the total traffic volume is stable and shows only a very mild decrease as the number of CPUs increases. On 8 cores, the proportion of local traffic was *at least* 70%. On the other hand, proportion of local traffic under WS decreases proportionally with the increase in the number of CPUs. On 8 cores, the proportion was *at most* 18%. For most work division factors, GP had ~ 8 times larger proportion of local traffic than GP. Contrary to our expectations, the strong locality of applications running under GP does not have a big impact on the running time on conventional shared-memory architectures. One of possible reasons for this is that our workloads are CPU-bound, but not memory-bound, so GP effectively helps in reducing only the amount of inter-CPU synchronization. However, the overhead of inter-CPU synchronization on our test machine is very low, so the benefits of this reduction become annihilated by the generally worse load-balancing properties of GP.

Nevertheless, we believe that this increase in locality would would lead to significant savings in running time on distributed systems and CPUs with more complex topologies, such as Cell, where inter-CPU communication and process migration are much more expensive than on our test machine. However, practical application of GP load-balancing in these scenarios requires that certain technical problems regarding existing graph partitioning algorithms, described in appendix, be solved.

Ongoing and future activities include evaluations on larger machines with more processors, possibly also on Cell, and looking at scheduling across machines in a distributed setting. Both scenarios have different topologies and interconnection latencies. In a distributed scenario, we envision a two-level scheduling approach where GP will be used to distribute processes across nodes, while WS will be used for load-balancing within a single node.

We have also found weaknesses in an existing, simulation-based results about WS scalability [14]. Based on the the combined insight from theirs and our results, we have identified a new, orthogonal dimension in which we would further

like to study WS performance characteristics: the relation between the number of processes in the system and the number of CPUs. Ultimately, we would like to develop an analytical model of WS performance characteristics, which would take into consideration the number of processes in the system, work granularity (which is inversely proportional with the amount of time a CPU core spends on useful work), as well as the system's interconnection topology.

Acknowledgments

We thank to Paul Beskow for fruitful discussions and proofreading the paper, and to the anonymous reviewers for their very helpful comments.

References

1. Lee, E.A.: The problem with threads. Computer 39(5), 33–42 (2006)
2. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: Proceedings of Symposium on Opearting Systems Design & Implementation (OSDI), Berkeley, CA, USA, p. 10. USENIX Association (2004)
3. Valvag, S.V., Johansen, D.: Oivos: Simple and efficient distributed data processing. In: 10th IEEE International Conference on High Performance Computing and Communications, 2008. HPCC 2008, September 2008, pp. 113–122 (2008)
4. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: distributed data-parallel programs from sequential building blocks. In: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems, pp. 59–72. ACM, New York (2007)
5. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating mapreduce for multi-core and multiprocessor systems. In: Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA), Washington, DC, USA, pp. 13–24. IEEE Computer Society, Los Alamitos (2007)
6. de Kruijf, M., Sankaralingam, K.: MapReduce for the Cell BE Architecture. University of Wisconsin Computer Sciences Technical Report CS-TR-2007 1625 (2007)
7. He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: a mapreduce framework on graphics processors. In: PACT 2008: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, pp. 260–269. ACM, New York (2008)
8. Vrba, Ž., Halvorsen, P., Griwodz, C.: Evaluating the run-time performance of kahn process network implementation techniques on shared-memory multiprocessors. In: Proceedings of the International Workshop on Multi-Core Computing Systems, MuCoCoS (2009)
9. Arora, N.S., Blumofe, R.D., Plaxton, C.G.: Thread scheduling for multiprogrammed multiprocessors. In: Proceedings of ACM symposium on Parallel algorithms and architectures (SPAA), pp. 119–129. ACM, New York (1998)
10. Catalyurek, U., Boman, E., Devine, K., Bozdag, D., Heaphy, R., Riesen, L.: Hypergraph-based dynamic load balancing for adaptive scientific computations. In: Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS 2007). IEEE, Los Alamitos (2007); Best Algorithms Paper Award
11. Kahn, G.: The semantics of a simple language for parallel programming. Information Processing 74 (1974)

12. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. Technical report, Cambridge, MA, USA (1996)
13. Blumofe, R.D., Papadopoulos, D.: The performance of work stealing in multiprogrammed environments (extended abstract). SIGMETRICS Perform. Eval. Rev. 26(1), 266–267 (1998)
14. Saha, B., Adl-Tabatabai, A.R., Ghuloum, A., Rajagopalan, M., Hudson, R.L., Petersen, L., Menon, V., Murphy, B., Shpeisman, T., Sprangle, E., Rohillah, A., Carmean, D., Fang, J.: Enabling scalability and performance in a large scale cmp environment. SIGOPS Oper. Syst. Rev. 41(3), 73–86 (2007)
15. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the Cilk-5 multithreaded language. In: Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, Montreal, Quebec, Canada, June 1998, pp. 212–223 (1998); Proceedings published ACM SIGPLAN Notices, Vol. 33(5) (May 1998)
16. Catalyurek, U.V., Aykanat, C.: Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. IEEE Transactions on Parallel and Distributed Systems 10(7), 673–693 (1999)
17. Richardson, I.E.G.: H.264/mpeg-4 part 10 white paper, `http://www.vcodex.com/files/h264_overview_orig.pdf`
18. Mann, H.B., Whitney, D.R.: On a test of whether one of two random variables is stochastically larger than the other. Annals of Mathematical Statistics (1947)
19. Chevalier, C., Pellegrini, F.: Pt-scotch: A tool for efficient parallel graph ordering. Parallel Comput. 34(6-8), 318–331 (2008)

A Selecting the Best GP Result Set

To evaluate the GP policy, we have varied the idle time parameter τ over a range of values for each given work division d. A single result set consists of 10 measurements for a given combination of d and τ. We cannot choose the best τ for a given d by simply selecting the τ having the lowest average (or median) running time. The reason is that consecutive runs of an experiment with the *same* combination of d and τ can have high variance.

To choose the best τ for a given d, we compare all result sets against each other and select the set $s(\tau)$ which compares smaller against the largest number of other sets. Formally, for a fixed d, we choose τ as follows:

$$\tau = \min(\arg\max_{\tau \in T} |\{\tau' \in T : (\tau' \neq \tau) \wedge (s(\tau) < s(\tau'))\}|)$$

where $|X|$ denotes cardinality of set X, $T = \{8, 16, \ldots, 256\}$ is the set of τ values over which we evaluated the GP policy, and $s(\tau)$ is the result set obtained for the given τ. To compare two result sets, we have used the one-sided Mann-Whitney U-test [18][7] with 95% confidence level; whenever the test for $s(\tau)$ against $s(\tau')$ reported a p-value less than 0.05, we considered that the result set $s(\tau)$ comes from a distribution with stochastically smaller [18] running time.

[7] Usually, the Student's t-test is used. However, it makes two assumptions, for which we do not know whether they are satisfied: 1) that the two samples come from a normal distribution 2) having the same variance.

B Migration Cost

α is just a scale factor expressing the relative costs of communication and migration, so it can be fixed to 1, and t_{mig} scaled appropriately; in our experiments we have used $\alpha = 1$ and $t_{mig} = 16$. This is an arbitrary low value reflecting the fact that our workloads have low migration cost because they have very light memory footprint. Since load-imbalance is the primary reason for bad performance of GP, another value for t_{mig} would not significantly influence the results because graph partitioning *first* establishes load-balance and *then* tries to reduce the cut cost. Nevertheless, even such a low value succeeds in preventing that a process with low communication volume and CPU usage is needlessly migrated to another CPU.

Workloads with a heavy memory footprint could benefit if the weight of their migration edges is a *decreasing* function $c(t_b)$ of the amount of time t_b a process has been blocked. As t_b increases, the probability that CPU caches will still contain relevant data for the given process decreases, and the cost of migrating this process becomes lower.

C NUMA Effects and Distributed Process Networks

We have realized that the graph-partitioning model described in section 2 does not always adequately model application behavior on NUMA architectures because it assumes that processes migrate to a new node together with their data. However, NUMA allows that processes and their data reside on separate nodes, which is also the case in our implementation. Nevertheless, the model describes well applications that use NUMA APIs to physically migrate their data to a new node. Furthermore, the graph-partitioning algorithm assumes that the communication cost between two processes is constant, regardless of the CPUs to which they are assigned. This is not true in general: for example, the cost of communication will be $\sim 10\%$ bigger when the processes are placed on CPUs 0 and 7 than when placed on CPUs 0 and 2.

These observations affect very little our findings because of three reasons: 1) the workloads use little memory bandwidth, so their performance is limited by message-passing, context-switch and inter-CPU synchronization overheads, 2) NUMA effects are averaged out by round-robin allocation of physical pages across all 4 nodes, 3) synchronization cost between processes assigned to the same CPU is minimal since contention is impossible in this case.

In a distributed setting, load-balancing based on graph models is relevant because of several significant factors: processes *must* be migrated together with their data, high cost of data migration, and high cost of communication between processes on different machines. Indeed, we have chosen to implement Devine's et.al. algorithm [10] because they have measured improvement in application performance in a distributed setting. The same algorithm is applicable to running other distributed frameworks, such as MapReduce or Dryad.

D Graph Partitioning: Experiences and Issues

Nornir measures CPU consumption in nanoseconds. This quickly generates large numbers which PaToH partitioner [16] used in our experiments cannot handle, so it exits with an error message about detected integer overflow. Dividing all vertex weights by the smallest weight did not work, because it would still happen that the resulting weights are too large. To handle this situation, we had two choices: run the partitioning algorithm more often, or aggressively scale down all vertex weights. The first choice made it impossible to experiment with infrequent repartitionings, so we have used the other option in our experiments: all vertex weights have been transformed by the formula $w' = w/1024 + 1$ before being handed over to the graph partitioner. This loss of precision, however, causes an *a priori* imbalance on input to the partitioner, so the generated partitions have worse balance than would be achievable if PaToH internally worked with 64-bit integers. This rounding error may have contributed to the limited performance of GP load-balancing, but we cannot easily determine to what degree.

As exemplified above, the true weight of an edge between two vertices in the process graph may depend on which CPUs the two processes are mapped to. This issue is addressed by *graph mapping* algorithms implemented in, e.g., the Scotch library [19]. However, SCOTCH does does not support pinning of vertices to given partitions, which is the essential ingredient of Devine's algorithm. On the other hand, PaToH supports pinning of vertices, but does not solve the mapping problem, i.e., it assumes that the target graph is a complete graph with equal weights on all edges. Developing algorithms that support both pinned vertices *and* solve the mapping problem is one possible direction for future research in the area of graph partitioning and mapping algorithms.