Limits of work-stealing scheduling

Željko Vrba,[1,2] Håvard Espeland,[1,2] Pål Halvorsen,[1,2] 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 unintuitive 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.

Both strategies are implemented in Nornir [8], which is our parallel processing runtime for executing programs expressed as Kahn process networks [9], but they are also applicable to any of the existing parallel processing frameworks.

Particularly, in this paper, we have evaluated the *work-stealing* load-balancing method [10] which is designed to be used in a multi-programmed environment. The authors have theoretically shown that the algorithm is optimal for scheduling programs that are structured as *fully-strict computations*, also called *fork-join* parallelism. Under this assumption, they have proven [11] that with P processors, 1) the parallel part of the program exhibits P-fold speedup, the and 2) that the additionally used space is less than P times the space used by the execution on 1 CPU. These results are also supported by experiments [12].

Saha et. al. [13] have presented a run-time system aimed towards executing fine-grained concurrent applications, much like Nornir. Their simulations show that work-stealing scales almost perfectly up to 16 cores, but they have not attempted to *quantify* work granularity, i.e., the amount of time a process runs before it blocks, and relate it to application performance. Explicit quantification of this time, in absolute and relative terms, is one of the contributions of this paper.

In our earlier paper [8] we have noted that static assignment of processes to CPUs can achieve as good as, or, for some workloads, even better, performance than work-stealing. In cases where static assignment performed better, there was a significant amount of inter-CPU communication. Since static assignment is impractical for large process networks, we have also experimented with a scheduling strategy based on *graph partitioning* [14], which balances the load across CPUs, and reduces the amount of inter-CPU communication as well as the cost of migrating processes.

Furthermore, we compare the performance of work stealing with the two other schedulers. Unsurprisingly, the performance of the static scheduler for compute-intensive applications with irregular structure suffered because of load-imbalances that occurred at run-time, but with well-known communication patterns, a static hand-optimized schedule may give good performance. With respect to the graph partitioning approach, we have observed that the graph partitioning scheme can compete with work stealing at certain repartitioning frequencies and work granularities. 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 30 in comparison with the work-stealing scheduler, but this reduction has no influence on the application running time.

2 Dynamic load-balancing

The static scheduler where the programmer (possibly manually) pins processes to the CPU at compile time is very limited in dynamic scenarios, and our search for more flexible scheduling and load balancing solutions reviled two promising approaches: 1) *work stealing* [10], because of its experimentally proven performance, and 2) periodically invoked *graph partitioning* algorithms, which reduce amount of communication between cooperating processes.

The two methods are described below. 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.

2.1 Work stealing

We have experimented with several (enhanced) versions of work stealing (see section 4), but in this context, the experimental results shown negligible differences, i.e., we here focus on the original idea as presented in [10]. For each CPU (kernel-level thread), there is 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[3] 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. Both of these operations have constant-time ($O(1)$) complexity, i.e., their running time is independent of the number of processes in the queue. The original motives for accessing run queues at different ends are, as explained in [15], two-fold: 1) it reduces contention by having stealing threads operate on the opposite end of the queue than the thread they are stealing from, and 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.

The original work-stealing algorithm uses non-blocking algorithms to implement queue operations [10]. However, since others [13] have shown that even a centralized queue protected by a single lock does not hurt performance on up to 8 CPUs, we have decided to simplify our scheduler implementation by protecting each run queue with its own lock. Thus, our implementation does not benefit from the first advantage of accessing run queues at different ends since we use locks to protect the queues. However, when considering message-passing applications, another advantage arises. Since a process is unblocked by the arrival of a message, placing it at the front of the ready queue increases probability that it will find the required data in CPU caches once it is scheduled.

[3] This is the opposite of the FIFO policy usually used in OS-schedulers, which have to be fair over all jobs in the system. However, we are considering scheduling of tasks within a *single job*, which terminates when all of its processes have terminated.

2.2 Graph partitioning

The goal of a graph partitioning algorithm over a graph with weighted vertices (processing requirements) and edges (communication requirements) is to divide the vertices of the graph into n (equal to the number of cores in the system.) disjoint sets of approximately equal weights, while at the same time minimizing the weights of edges between any two partitions. This is an NP-hard problem and only heuristic algorithms yield solutions in reasonable time.

In our initial experiments, we have used the SCOTCH [16] graph partitioning library, but the results quickly showed that processes often unnecessarily migrated, which has a detrimental effects to CPU's caches. We have therefore also implemented the algorithm proposed by Devine et al. [14] using the PaToH library [17] in our scheduling component. This was one of the first algorithms that took into account not only costs of communication, but also costs of process migration, and we use this algorithm in our experiments presented in section 3.

The algorithm observes weights on vertices and edges, corresponding to CPU loads of individual processes and communication volumes between them. The algorithm repartitions the graph, which also includes migration of processes and their data, whenever it detects a significant imbalance of CPU load or that communication costs exceed migration costs considerably. The algorithm minimizes the cost function $\alpha t_{comm} + t_{mig}$, where 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. Since α is just a scale factor expressing the relative costs of communication and migration, we have set $\alpha = 1$ and $t_{mig} = 16$ in our implementation. This value was only intended to reflect the expected low cost of migration and serve as a starting point for further experimentation. However, based on current experiment results, we believe that changing this value would not have a significant impact on benchmark results; this issue is discussed in more detail in Section 4.

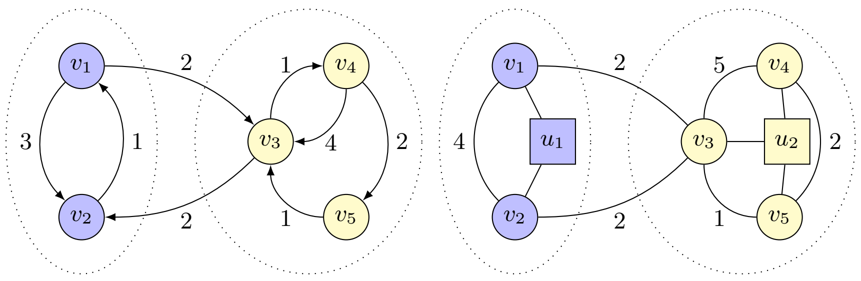


Fig. 1. An example of transforming a process graph into a rebalancing graph with $\alpha = 1$. Current partitions are delimited by ovals and distinguished by nodes of different colors.

For the purposes of *rebalancing*, the process graph is transformed into an *undirected rebalancing graph* in 4 steps (see also figure 1):

1. The process and channel accounting data is used to set weights on the original graph.
2. All edges between the same pair of nodes are transformed into a single edge whose weight is the sum of weights of individual collapsed channels multiplied by α.
3. n new nodes, $u_1 \dots u_n$, representing the n CPUs to which processes can be assigned, are introduced and *fixed* to their respective partitions.
4. For each node u_k, a *migration edge* is created between u_k and every node v_i representing a task that is currently assigned to CPU k. The weight of the migration edge is set to the cost of migrating data associated with process v_i (in our case, 16, as explained above).

For the *initial partitioning* phase, which is done before the network starts running, the process graph is transformed into an undirected graph as described in the previous section, but with small differences: 1) since the actual CPU times and communication intensities are not known, unit weights are assigned to channels and vertices, and 2) the additional CPU nodes 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 *collectively* accumulated by all threads, and invokes the repartitioning algorithm when the idle time has crossed a preset threshold. After the algorithm has finished, it resets process and channel accounting data to 0, in preparation for the next partitioning. When a thread attempts to take a process from an empty run-queue, it updates the collective idle time and continues to check the run-queue with exponentially increasing sleep times (up to 32μs) 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

The load-balancing methods have been evaluated on several process networks with different topologies: an H.264 encoder topology, a MapReduce topology, a scatter/gather topology and two randomly generated directed graphs, i.e., one without cycles, and another containing 13 cycles. The acyclic graph has 239 nodes and 364 edges, while the cyclic graph has 213 nodes, 333 edges and 13 cycles. In the presentation of results, we designate work stealing, graph partitioning methods by 2-character strings `WS` and `HP`, respectively. For the graph partitioning load-balancer, we have used the state of the art PaToH library [17].

The test programs have been compiled as 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 partitioning code. The benchmarks have been run on an otherwise idle 2.6 GHz AMD Opteron machine with 4 dual-core CPUs, 64 GB of RAM, and running linux kernel 2.6.27.3. Each experiment has been repeated 10 consecutive times. We have configured our run-time system to collect detailed accounting about all aspects of operation, such as number of messages towards the same or other CPUs.

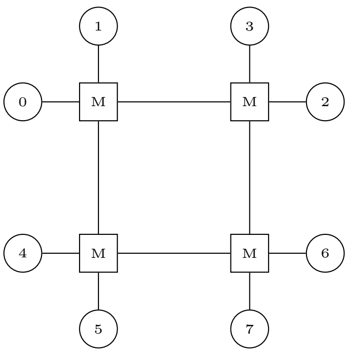
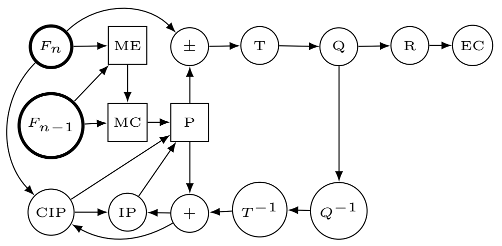


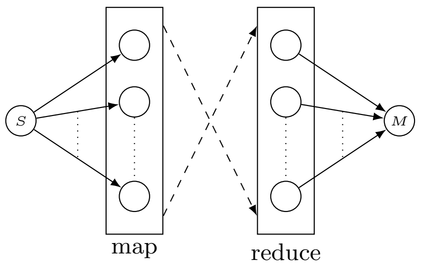
Fig. 2. Topology of the machine used for experiments. Round nodes are cores, square nodes are NUMA memory banks. Each CPU has one memory bank and two cores associated with it.

3.1 Description of workloads

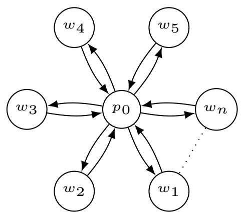
Figure 3(a) shows a process network implementing an **H.264** video-encoder, and it is only a slight adaptation of the encoder block diagram found in [18]. We have used cachegrind to profile x.264, an open-source H.264 codec, and mapped the results to the process graph. We have found that the P, MC and ME stages use together over 50% of the CPU, so we have parallelized each of these by attaching 512 processes by a pair of channels (request-response) to each block.



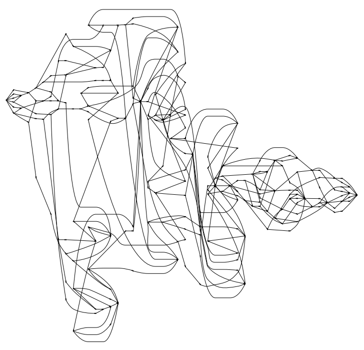
(a) H.264 process network. F_n and F_{n-1} are inputs to the network (current and previous frames).



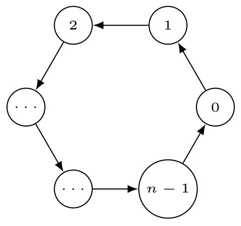
(b) MapReduce topology. Dashed lines denote that each process from the map stage is connected to each process in the reduce stage.



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



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



(e) Ring topology.

Fig. 3. Process networks used for benchmarking.

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

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(b)), 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(d) 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/k, where n is the number of messages sent by the source, T is a constant that equals ~ 1 second of CPU-time, and k is the work granularity. In effect, each single message sent by the source (a single integer) carries $w = T/k$ 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.

The **scatter/gather** network has a single central process (p_0) connected to n worker processes. The central process scatters m messages to the workers, each which performs a set amount of work w for each message. When complete, a message is sent from the worker process to the central process (see Figure 3(c)). The process 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.

3.2 Results

Table 1 summarizes experiments that have been run. The designations from the left column will be used in further presentation of results. The running times are presented as wall-clock time. This is the most representative metric, because 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.

The **k-means** program, which executes on a MapReduce topology, turned out to be a pathological case for load-balancing based on graph partitioning. Experiments have shown that, if the repartitioning interval is too small, the

Workload	Description
H264	512 processes per stage, 120 encoded "frames", with $\sim$ 0.25s total work per frame.
RING	1000 processes, 1000 round-trips.
RND-A{1,7}	Random graph, 239 nodes and 364 edges. $w \in \{1, 10, \ldots, 10000, 50000, 90000\}$
RND-B{1,7}	Random graph, 213 nodes, 333 edges and 13 cycles. $w \in \{1, 10, \ldots, 10000, 50000, 90000\}$
SG	Scatter-gather on 1000 workers, with $w \in \{10^3, 10^4, 10^5, 10^6, 2 \cdot 10^6, \ldots, 10^7\}$ on 1 and 8 CPUs.
K-MEANS	k-means on MapReduce topology. 128 workers, 120 clusters, 300000 points.

Table 1. Summary of benchmark workloads (18 in total).

Method	Description
`WS`	Work-stealing (see Section 2.1) on 1 and 8 CPUs.
`HP,n`	Graph-partitioning with accumulated idle time parameter set to `n`, which is multiplied by the number of CPUs used to run the benchmark (see section 2.2). This is the total number of unsuccessful dequeue attempts by all threads, and is varied from 32 to 256 in steps of 8. Performed on 8 CPUs only.

Table 2. Summary of tested scheduling methods (30 in total).

partitioning algorithm runs very often, and the total running time is *several minutes*, and if it is too large, the partitioning runs only once, at network startup. The transition between the two behaviors happens for the values of idle time parameter around 40; the exact value is hard to determine because the graph partitioning algorithm is non-deterministic, and on each run it may produce different load-imbalances which trigger repartitioning. However, for the values of $n \geq 48$, the program always exhibits the "good" behavior. In the case, the program finishes in 10.2 seconds under the `HP` method. Under the `WS` method, the program finishes in 9.4 seconds. Because of this anomaly, we will not consider this benchmark in further discussions.

Figures 4, **??**, and **??** show how median, minimum and maximum running time depend on the scheduling methods. More precisely, each run of the experiment (see Table 1) is run under scheduling methods summarized in Table 2. For each scheduling method, 10 experiment runs are performed. Then, for each set of 10 measurements, the median (resp., minimum and maximum) value is found; now every method is represented by a single value. Then, over this new set of values, the method having the minimum (resp. minimum, maximum) value is selected. For the selected method, shown on the x-axis together with the workload, two box-plots[4] are shown. The upper figure shows the distribution of

[4] Also called box and whiskers plot. This is a standard way to show the distribution of a data set. The box's span is from the lower quartile to the upper quartile, with

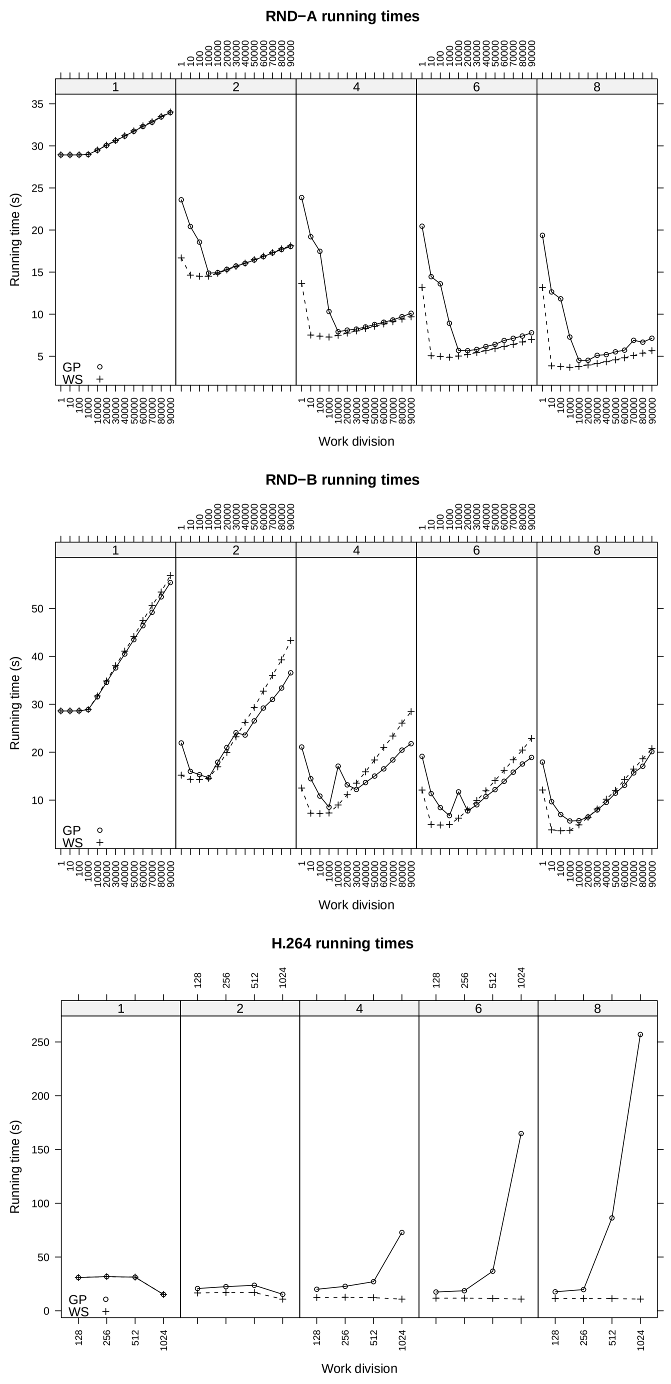


Fig. 4. Running times for WS and GP policies on 1,2,4,6 and 8 CPUs.

all 10 measured running times, while the lower figure shows the distribution of $\log(v_l/v_r)$, v_l being the number of messages sent locally (between processes on the same CPU) and v_r being the number of messages sent remotely (between processes on different CPUs).

From figures 4 and **??** it can be seen that the *`WS` has the least median and minimum running time* for all workloads except the ring, and RND-B5 to RND-B7 workloads, where the graph-based load-balancing (`HP`) wins. Common for them is that every process in the network performs very little work per message; ring is the extreme case where each process does no processing and just forwards the message. This creates heavy contention on run-queues and, consequently, decreases performance. The `HP` method accesses another thread's run-queue only after encountering processes that have been reassigned to other threads, i.e., it accesses its own queue most of the time.

From Figure **??** it is clear that the *`HP` method achieves the worst running time* on all workloads except the ring benchmark. Interestingly, it is also the *best* method for scheduling RND-B5 to RND-B7 workloads, just with different idle time between two repartitionings. From figures **??** and **??** we see also that there is no correlation between the value of the idle time parameter and the running time.

From figures 4, **??** and **??** we see that the `HP` method consistently achieves better ratio of local to remote traffic on random graph benchmarks than the `WS` method. The improvement is by a factor of $\sim$ 10–30 (note that the ratio graphs have logarithmic scale). From figure **??** we also see that, under the `HP` method, the median ratio of the amount of local and remote traffic is ~ 3. This ratio is approximately the same in cases where the `HP` method exhibits both the best and the worst running time (workloads RND-B5 to RND-B7 in figures **??** and **??**).

Figure 6 shows distributions of relative speedups of all workloads relative to the running time on one CPU. It shows that `WS` method achieves nearly perfect linear speedup on random networks for moderate work granularities. For small granularities, the speedup is limited by having too few active processes, and for high granularities the speedup is limited by high contention over run-queues, similarly to the ring benchmark. In **H.264** workload, the speedup is limited by data dependencies in the process graph. We can also see that the `HP` method *never* achieves a speedup greater than 6. In bad cases (bad choice of the idle time parameter) the speedup can be barely a bit better than sequential performance (see the bottom graph in Fig. 6 for RND-B workloads).

Figure 7 shows how the running time of the `SG` benchmark depends on different work granularities on 1 and 8 CPUs. For each work granularity w, process p_0 (see Figure 3) sends a message to 1000 worker processes, each message containing work amount of $1/w$ CPU seconds; this process is repeated in $1000w$ iterations.

---

the bold bar denoting the median. Whiskers extend to the lowest (from the bottom edge) and to the highest (from the top edge) measurement that is not an outlier. Circles denote possible *outliers*, i.e., data points that are more than 1.5 times the box's height (interquartile range) below the lower or above the upper quartile.

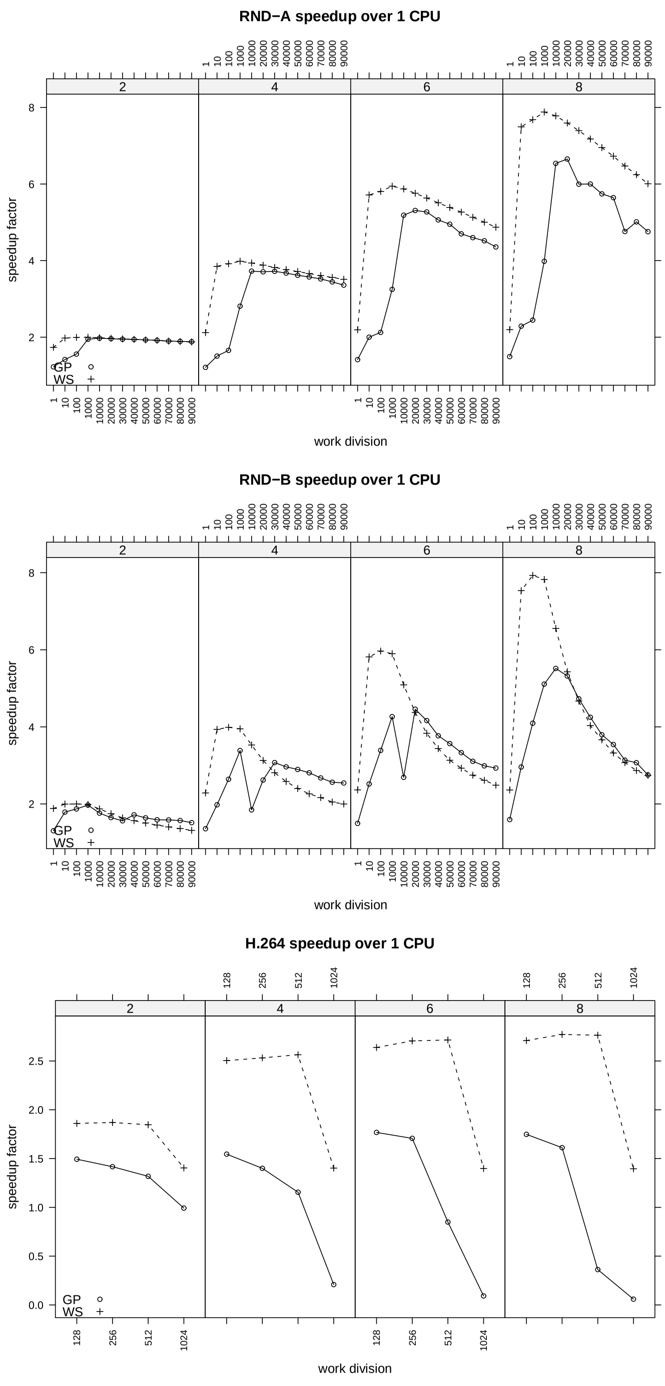


Fig. 5. Relative speedup of WS and GP policies over 1 CPU.

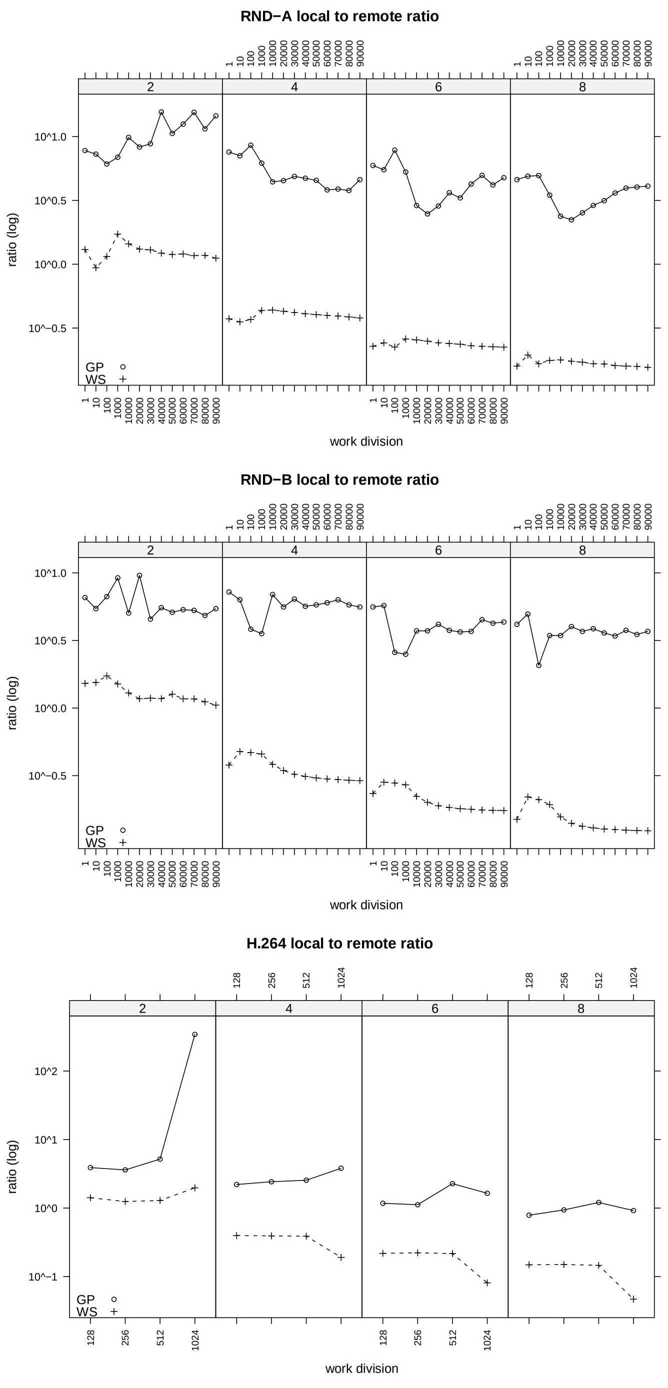


Fig. 6. Local to remote ratio of WS and GP policies.

On 1 CPU, for $w = 1000$, the total real running time of the benchmark is 1 second.

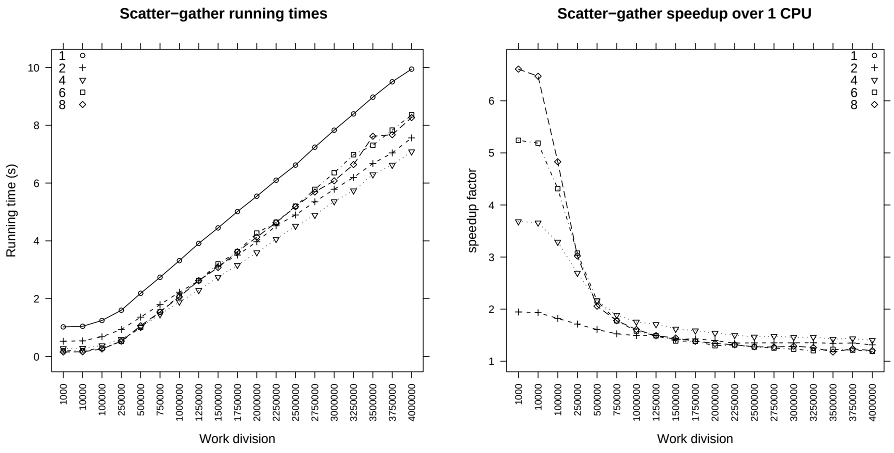


Fig. 7. Performance of work-stealing method for varying work divisions. Note that the scale on the x-axis is non-linear and non-uniform.

It can be seen that `WS` on 8 CPUs shows speedups over 1 CPU of 6.6, 6.5, and 4.9 for work granularities $w \in \{10^3, 10^4, 10^5\}$ respectively. Thus, we can say that the `WS` performs well as long as each process uses at least $100\mu s$ of CPU time per message. The steepest fall is between $w = 10^5$, and $w = 10^6$, where the speedup is only 1.6; after this point the speedup continues to decrease slowly to 1.1 for $w = 10^7$. The main source of this performance degradation is contention over run-queues. This in turn is caused by frequent process block/unblock operations which are a result of little work done per message.

3.3 Summary

We can summarize our findings as follows:

- The `WS` method is the scheduling method that gives best performance, i.e., almost linear speedup with the number of CPUs compared to performance on 1 CPU, *provided* that processes use at least $\sim 100\mu s$ of CPU time between each unblock and block.
- Smaller work quantities have negative impact also on the `HP` method, but much less so than on the `WS` method. Thus, the `HP` method should be selected for such workloads, even though it *never* achieved speedup comparable to that of `WS`, and sometimes much worse.
- There is no clear correlation between application performance and the idle time parameter of the `HP` method. Also, the `HP` is non-deterministic, which results in widely varying execution times between runs of the same workload.

– The HP method consistently results in more local than remote traffic, by a factor of up to ~ 30 compared with the WS method. However, this improvement in locality has *no* influence on application performance.

Thus, work-stealing should be the algorithm of choice for scheduling general-purpose workloads. However, specific applications, especially those that use very fine-grained parallelism, could benefit of specialized methods such as HP. However, we deem that the main problem with the HP method is the big spread of speedups that it achieves in consecutive runs (see figure 6). This is a significant finding also for the distributed case, and developing methods that give a narrower distribution of speedups, i.e., make performance more predictable, is one possible research direction in this area.

From the above considerations, we have also concluded that the source of pathologically bad performance of the ring workload under WS is not the amount of inter-CPU synchronization, but the frequency with which the processes are blocking.

4 Discussion

4.1 NUMA effects and distributed process networks

Since a context-switch includes a stack switch and is performed often with many processes and messages in the network, it is to be expected that cached stack data will be quickly lost from CPU caches. 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 (2-dimensional hypercube with 2 cores attached to each memory bank; see figure 2). Due to an implementation detail of our run-time system and Linux's default allocation policy, all stack memory would be allocated on a single node, which would make context-switch cost dependent on which node the process is scheduled. To average out these effects, we have run all benchmarks under the interleave NUMA (non-uniform memory access) policy which allocates physical memory pages from CPU nodes in round-robin manner.[5] 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 4 (the number of nodes in our machine). This ensures that the "top" stack pages of all processes are evenly distributed across CPUs.

We have also realized that the graph-partitioning model described in section 2 does not adequately model the behavior of applications on NUMA architectures in all situations. The model assumes that processes migrate together with their data to a node where they are executed, while on NUMA, processes and their data may reside on separate nodes, which is the case in our implementation. However, data migration is possible also on NUMA architectures, and the model describes well applications that migrate processes *and* data.

[5] This is achieved by running benchmark programs under the control of the `numactl` program.

Furthermore, the graph-partitioning algorithm assumes that the cost of an edge between nodes A and B is constant, regardless of the CPUs to which the nodes are assigned. This is not true in general, because we have measured that on our NUMA machine (see figure 2) the cost of communication between A and B will be $\sim 10\%$ bigger when they are, for example, 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, 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 a distributed setting, two factors play a significant role that make load-balancing algorithms based on graph-partitioning relevant: high cost of migrating process data, and high cost of sending messages between different machines. Indeed, we have chosen to implement Devine's et.al. algorithm [14] 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.

4.2 Implementations of graph partitioning: mapping problem

Our run-time system measures CPU consumption in nanoseconds, which quickly generates rather large numbers, which the PaToH library could not handle – it exited with an error message about detected integer overflow. We could not either divide all vertex weights by the smallest weight, because it would still happen that the resulting weights are still too large. To handle this situation, we had two choices: either 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 implemented the other option: 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 *a priori* imbalance on input to the partitioner, so the generated partitions have worse balance than would be achievable if PaToH would internally work with 64-bit integers.

As mentioned above, an edge in a source graph may change its weight, depending on nodes the processes it connects are mapped to. The *graph mapping* problem takes this phenomenon into account. Graph mapping algorithms take as input *two* weighted graphs, the source and target graphs. The source graph described the process network, while the target graph describes how the edge weights of the source graph change depending on the mapping. The SCOTCH library [16] implements mapping heuristics, but it does not support pinning of vertices to given partitions, which is the essential ingredient of Devine's algorithms. The PaToH library [17], which we have used in our benchmarks, 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 this area.

4.3 Migration cost

In our experiments, we have used a fixed value $t_{mig} = 16$. The graph partitioner's *first* priority is to establish load-balance and its *second* priority is to minimize the communication cost between processes in different partitions. Since load-imbalance is the primary reason for graph partitioning yielding worse performance than work stealing, another value for t_{mig} would not have influence the outcome of the experiments.

None of our workloads has a heavy memory footprint. However, such processes could benefit if the weight of their migration edges would be 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.

4.4 Variations of work stealing

We have also tested two additional variations on work stealing. In the first variant, the thread sleeps with exponentially increasing times, up to $16\mu s$, between steal attempts. Compared to the default version which yields between steal attempts, this version used somewhat less CPU time, with little negative impact on the real running time of the application. The exception is the ring benchmark, which exhibited somewhat worse running time.

The other variation was configured to insert processes into two parallel data structures: a queue of processes and a balanced tree sorted by the amount of local and remote traffic that each process made since it was migrated to its current thread. The stealing thread would then steal the process which exhibited the most remote traffic and least local traffic, in an attempt to minimize the amount of remote traffic. Due to using more complex data structures, this variant performed slightly worse than the default work-stealing method, and did not exhibit any better ratio of local to remote traffic compared to the default work stealing.

4.5 Interaction with the OS scheduler

Our (kernel-level) threads, which execute (user-level) processes, are each pinned to exactly one CPU in the system. The kernel may preempt and schedule out these threads to execute other applications which are present on the system. For work-stealing, the authors have proven [10] that the application speedup is proportional to the average CPU allocation during its run-time.

4.6 Comments on Intel's scalability simulations

Saha et. al. [13] have presented a run-time system aimed towards executing fine-grained concurrent applications, much like our run-time system which we used to implement and execute the experiments. Their simulations show that work-stealing scales almost perfectly up to 16 cores, but they have not attempted to *quantify* work granularity and relate it to application performance.

Our experiments complement their findings in that we show that the work stealing indeed *is* the best method for scheduling all workloads on middle-sized machines as long as processes do not block too often. Our experiments on 8 CPUs with random and scatter/gather graphs consistently show that the minimal needed amount of work that a process has to perform between being woken up and being blocked again is $\sim 100\mu s$.

4.7 Miscellaneous remarks

Even though we have run the experiments on our implementation [8] of Kahn process networks [9], we have carefully designed them so that they execute correctly even when behaviors that are specific for KPNs, most notably run-time deadlock detection and resolution, are disabled. However, processes are still limited only to blocking, uninterruptible reads and have the ability to wait for message arrival only on a single channel at a time.

5 Conclusion and future work

Scheduling and load balancing mechanisms taking into account the properties of parallel programs are of vital importance for performance on multi-core architectures. In this paper, we have evaluated scheduling algorithms in the context of modern parallel program runtimes. In particular, we have evaluated the *work stealing* approach. Our experimental results confirm the results previously presented [12] that work stealing in many cases performs well, but we have also identified some limitations, e.g., the performance decrease when very little processing is performed per message before blocking on read for another message. In such a case, graph partitioning algorithms may improve the performance, but also here performance suffers. We also envisioned a larger impact of different memory latencies to different memory banks, and work stealing performed about 30 times the number of remote versus local message passing operations compared to graph partitioning. This would give a huge difference when the edge costs increase more, as in a distributed system or in a system with a larger and more complex processor topology increasing the communication costs further, but in our tests, however, this has no effect on application performance with respect to job finishing time.

Ongoing and future activities include tests on larger machines with more processors and looking at scheduling across machines in a distributed setting. Both will give other properties of the processor topology, possibly requiring slightly different algorithms. Especially, in a between-machine scenario, the high-level partitioning will be an important part while at the same time achieving an efficient, load balanced schedule locally on each node.

Acknowledgments

We thank Paul Beskow for fruitful discussions and proofreading the paper.

References

1. Lee, E.A.: The problem with threads. Computer **39**(5) (2006) 33–42
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, USENIX Association (2004) 10–10
3. Valvag, S.V., Johansen, D.: Oivos: Simple and efficient distributed data processing. In: High Performance Computing and Communications, 2008. HPCC '08. 10th IEEE International Conference on. (Sept. 2008) 113–122
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, New York, NY, USA, ACM (2007) 59–72
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, IEEE Computer Society (2007) 13–24
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 '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, New York, NY, USA, ACM (2008) 260–269
8. Željko Vrba, 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. Kahn, G.: The semantics of a simple language for parallel programming. Information Processing **74** (1974)
10. 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), New York, NY, USA, ACM (1998) 119–129
11. 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)
12. Blumofe, R.D., Papadopoulos, D.: The performance of work stealing in multiprogrammed environments (extended abstract). SIGMETRICS Perform. Eval. Rev. **26**(1) (1998) 266–267

13. 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) (2007) 73–86
14. 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'07), IEEE (2007) Best Algorithms Paper Award.
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) 212–223 Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.
16. Chevalier, C., Pellegrini, F.: Pt-scotch: A tool for efficient parallel graph ordering. Parallel Comput. **34**(6-8) (2008) 318–331
17. Catalyurek, U.V., Aykanat, C.: Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. Parallel and Distributed Systems, IEEE Transactions on **10**(7) (1999) 673–693
18. Richardson, I.E.G.: H.264/mpeg-4 part 10 white paper (2002)