

Work stealing: an annotated bibliography

Daniel B. Neill

Department of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3891
neill@cs.cmu.edu

1 Notes

- Key papers to look at: Blumofe & Leiserson (1994), Squillante and Nelson (1991), Mitzenmacher (1998), Squillante et al (2001).
- Early implementations of work-stealing: Burton & Sleep (1981), Halstead (1984), Mohr et al (1990). Halstead's Multilisp: multiprocessor version of Lisp. Mohr et al: Lazy task creation. When deciding whether to spawn a new task for a function or execute it inline, put potential new task on a queue, allow other processors to steal, if not stolen then execute on original machine.
- Work stealing vs. work sharing. In work sharing, whenever a processor attempts to generate new threads, it (or a centralized scheduler) attempts to move some threads to underutilized processors. In work stealing, underutilized processors attempt to take threads from other processors. Migration of threads occurs less frequently with work stealing than work sharing, especially when load is high and noone wants to steal (Blumofe & Leiserson, 1994). Eager et al (1986) argue that work-sharing outperforms work-stealing at light to moderate system loads, while work-stealing outperforms work-sharing at high loads, if the costs of task transfer under the two strategies are comparable. However, they argue that costs are likely to be greater under work-stealing, making work-sharing preferable. This is because work-stealing policies must transfer tasks which have already started executing, while work-sharing policies can transfer prior to beginning execution. (Cost of task transfer is assumed to be processor cost at sending node.) Mirchandaney et al (1989) have similar results, assuming that the cost of task transfer results from network delays (exponentially distributed task-transfer times). Analysis techniques for both: decomposition of Markov chain, matrix-geometric approach, validation by simulations. (Note that both papers neglect overhead due to probing.)

However, Squillante and Nelson (1991) argue that the costs of moving a task are very different in shared-memory systems: in a distributed system, costs are incurred by the processor from which the task is migrated, possibly with an additional network delay (after this, the processing requirements of the task are identical). For shared-memory systems, an idle processor can search the queues of other processors and remove a task without disturbing the busy processors; the time required for probing and removal of processes is small (only a few memory references). The major cost of task migration is a larger service demand at the processor that migrated the task, reflecting the time needed to establish the cache's working set at this processor. Thus the direct costs of migration are shifted from the busy processor to the idle processor, and work-stealing has much greater potential benefits. Second, an indirect cost of task migration in a shared memory multiprocessor is increased contention for the communication medium (i.e. memory bus) and the shared memory itself; in distributed systems, on the other hand, contention for the communication network is unlikely. Finally, shared-memory multiprocessors have no difficulty migrating a task which has already started execution; this is very different than the results of Eager et al for the distributed case.

- Blumofe and Leiserson’s work stealing algorithm (1994). WS algorithm has one dequeue per processor; processor treats as stack but other processors can steal from other end. Assumes that processors will work on their own tasks if possible, but steal from a randomly chosen processor if their dequeue is empty. They model contention by assuming that steal requests are serially queued by an adversary (worst-case analysis). They present bounds on expected execution time, space required, and total communication cost, and show that work stealing has much lower communication cost than work sharing. Used in Cilk (multithreaded programming language).
- Squillante and Nelson (1991) present a threshold-based queueing model of shared-memory multiprocessor scheduling. They assume identical processors, distributions of arrival rates λ , and distributions of processing times μ . Also, time to probe and migrate and waiting task (if a processor with more than T tasks can be found within L_p unique and random probes) is exponentially distributed, and processing time at new processor higher because of affinity (also exponentially distributed but with parameter $\alpha < \mu$). Queueing model results in discrete state space, continuous-time Markov process. Large and complex state space; decompose by assuming processor states stochastically independent and identical. Approximate for finite number of processors, so compared with simulation. General form allows modeling of degradation in system performance due to task migration (increased processing time for both migrated and non-migrated tasks). Even when migration costs are large, and contention compounds these costs by degrading system performance, task migration may still be beneficial. Threshold policies prevent processor thrashing: instability when tasks passed back and forth and most of system time spent on migration.
- Mitzenmacher (1998) uses differential equations to study work stealing. His method is an approximation for a finite number of processors, though it is exact as the number of processors goes to infinity. Basic model: identical processors, task arrivals Poisson with parameter λ , service times exponential with parameter μ , WS algorithm, stealing instantaneous. Extensions: threshold stealing (only steal from queues with at least T tasks), preemptive stealing (steal if your queue has less than R tasks), repeated steal attempts, varying service and arrival distributions, transfer time for tasks, multiple choices (either for initial task placement or for stealing from most heavy load), stealing multiple tasks (e.g. as in Rudolph et al), varying processor speeds, varying arrival rates.
- Affinity scheduling: Squillante & Lazowska (1993), Squillante et al (2001), Acar et al (2001). In a shared-memory multiprocessor system, it may be more efficient to schedule a task on one processor than another. One reason for this is that the processors, or their associated resources, may be heterogeneous; we consider this case in more detail below. Another reason is *processor-cache affinity*: when a task returns for execution and is scheduled on a processor, it experiences an initial burst of cache misses. However, if a significant portion of the task’s working set is already in the cache, this penalty is reduced. Thus we have a tradeoff between load balancing (assigning tasks to less loaded processors) and using locality (assigning tasks to processors with high affinity).

Squillante and Lazowska propose and compare various scheduling algorithms which trade off load balancing and processor-cache affinity. They use a detailed cache model to examine cache reload time, as well as examining the effects of increased bus traffic. Analysis techniques: a combination of mean value analysis (for fixed-processor scheduling), bounding approximations, and simulation. A shared pool of tasks is assumed: idle processors search this pool and choose a task associated with that processor if possible. This outperforms a simple FIFO queue (good load balancing, bad locality) or a “fixed processor” model where each processor has its own queue and tasks are not shared (good locality, bad load balancing). No work stealing since pool of tasks is shared.

Squillante et al assume a generic form of affinity in which the service rates μ_{ij} are higher for jobs in a processor’s own queue ($i = j$). However, some queues have higher priorities c_j than others, so processor i will choose the job with highest $\mu_{ij}c_j$. Also, we can assume that some processors are not allowed to process some queues by setting $\mu_{ij} = 0$. Squillante et al shows that a threshold-based priority algorithm works well: a threshold T_j is set for each queue j , and queues with a number of tasks exceeding this threshold are given priority. They present an algorithm which determines where the thresholds should be set. Analysis techniques: queueing theory to give approximate results for the

two-queue two-server case, and simulations for all cases. Queueing model potentially useful; assume costs identical so only idle queues steal?

Acar et al present a work-stealing algorithm that uses locality information, and thus outperforms the standard work-stealing algorithm on benchmarks. Each process maintains a queue of pointers to threads that have affinity for it, and attempts to steal these first. They also bound the number of cache misses for the work stealing algorithm, using a “potential function” argument.

- Applications to thread scheduling. Arora et al (1998) present a non-blocking implementation of the work-stealing algorithm. They assume that the scheduler maps threads onto processes while an adversarial kernel maps processes to processors, and bound the expected execution time.
- More theoretical results. Blleloch et al (1995) improve the space bounds of Blumofe & Leiserson for a global shared-memory multiprocessor system. Fatourou & Spirakis (2000) extend the Blumofe & Leiserson model to strict multithreaded computations (thread dependent on ancestor, not just parent). Berenbrink et al (2001) show that the work stealing algorithm is *stable* even under a very unbalanced distribution of loads.
- Other load balancing strategies. Hamidzadeh & Lilja (1996) exemplify centralized scheduling strategies: use a dedicated processor to compute “smarter” scheduling, take locality into account. Lo and Dandmudi (1996) attempt to obtain both the benefits of centralized scheduling (better load sharing) and distributed scheduling (no performance bottleneck at a centralized queue) through a hierarchical load sharing approach. Rudolph et al (1991) assume that load-balancing operations are performed by a processor with probability inversely proportional to the size of its workpile; balancing consists of choosing a random processor and redistributing tasks to equalize the size of the two workpiles.
- Heterogeneous systems: Mirchandaney et al (1990). Type 1 systems have processors identical with respect to processing capabilities and speeds, but different arrival rates λ . Type 2 systems may also have different processing rates μ . As in their earlier paper, they assume that job transfers encounter significant delays due to network processing at source and destination, and transmission time. Again, load-sharing (Forward) and load-stealing (Backward) algorithms are considered. Analysis techniques: Markov chain; decomposition into simpler chains then exact solution using matrix-geometric techniques. Models validated with simulations. Most interesting idea: *biased probing* (attempt to share load with nodes that are likely to have smaller load; or attempt to steal load from nodes that are likely to have larger loads). Biased probing results in significant (30-40%) performance gains for work-stealing though not for work-sharing. Bender & Rabin (2002) focus on the case where processors have different speeds, each processor maintains an estimate of its own speed, and communication between processors has a cost. They propose a version of the WS algorithm where a faster processor can interrupt a slower processor and steal its current task (if the victim’s queue is empty, so normal stealing is impossible, and if the victim is slower by a factor of at least β), and they bound the execution time of this algorithm.

2 Alphabetical list of references

- U.A. Acar, G.E. Blleloch, and R.D. Blumofe (2000). The data locality of work stealing. *Proc. 12th ACM Symposium on Parallel Algorithms and Architectures*, 1-12.
- N.S. Arora, R.D. Blumofe, and C.G. Plaxton (1998). Thread scheduling for multiprogrammed multiprocessors. *Proc. 10th ACM Symposium on Parallel Algorithms and Architectures*, 119-129.
- M.A. Bender and M.O. Rabin (2002). Online scheduling of parallel programs on heterogeneous systems with applications to Cilk. *Theory of Computing Systems* **35**, 289-304.
- P. Berenbrink, T. Friedetzky, and L.A. Goldberg (2001). The natural work stealing algorithm is stable. *IEEE Symposium on Foundations of Computer Science*, 178-187.
- G.E. Blleloch, P.B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Proc. 7th ACM Symposium on Parallel Architectures and Applications*, 1-12.

- R.D. Blumofe and C.E. Leiserson (1994). Scheduling multithreaded computations by work stealing. *Proc. 35th IEEE Conference on Foundations of Computer Science*, 356-368.
- F.W. Burton and M.R. Sleep (1981). Executing functional programs on a virtual tree of processors. *Proc. 1981 Conference on Functional Programming Languages and Computer Architecture*, 187-194.
- D.L. Eager, E.D. Lazowska, and J. Zahorjan (1986). A comparison of receiver-initiated and sender-initiated load sharing. *Performance Evaluation* **6**, 53-68.
- P. Fatourou and P. Spirakis (2000). Efficient scheduling of strict multithreaded computations. *Theory of Computing Systems Journal* **33**(3), 173-232.
- R.H. Halstead, Jr. (1984). Implementation of Multilisp: Lisp on a multiprocessor. *Proc. 1984 Symposium on Lisp and Functional Programming*, 9-17.
- B. Hamidzadeh and D.J. Lilja (1996). Dynamic scheduling strategies for multiprocessors. *Proc. 16th Intl. Conference on Distributed Computing*, 208-215.
- M. Lo and S.P. Dandamudi (1996). Performance of hierarchical load sharing in heterogeneous systems. *Proc. Intl. Conf. Parallel and Distributed Computing Systems*, 370-377.
- R. Mirchandaney, D. Towsley, and J.A. Stankovic (1989). Analysis of the effects of delays on load sharing. *IEEE Trans. Computers* **38**(11), 1513-1525.
- R. Mirchandaney, D. Towsley, and J.A. Stankovic (1990). Adaptive load sharing in heterogeneous distributed systems. *J. Parallel and Distributed Computing* **9**, 331-346.
- M. Mitzenmacher (1998). Analyses of load stealing models based on differential equations. *Proc. 10th ACM Symposium on Parallel Algorithms and Architectures*, 212-221.
- E. Mohr, D.A. Kranz, and R.H. Halstead, Jr. (1991). Lazy task creation: a technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel and Distributed Systems* **2**(3), 264-280.
- L. Rudolph, M. Slivkin-Allalouf, and E. Upfal (1991). A simple load balancing scheme for task allocation in parallel machines. *Proc. 3rd ACM Symposium on Parallel Algorithms and Architectures*, 237-245.
- M.S. Squillante and E.D. Lazowska (1993). Using processor-cache affinity information in shared-memory multiprocessor scheduling. *IEEE Trans. Parallel and Distributed Systems* **4**(2), 131-143.
- M.S. Squillante and R.D. Nelson (1991). Analysis of task migration in shared-memory multiprocessor scheduling. *Proc. ACM Conference on the Measurement and Modeling of Computer Systems*, 143-155.
- M.S. Squillante, C.H. Xia, D.D. Yao, and L. Zhang (2001). Threshold-based priority policies for parallel-server systems with affinity scheduling. *Proc. American Control Conference*, 2992-2999.