Efficient Distributed Macroscheduling on Networks of Workstations

Philip Lisiecki
MIT Laboratory for Computer Science
Cambridge, MA 02139
`lisiecki@theory.lcs.mit.edu`

A network of workstations provides a natural platform for running parallel applications. Since these workstations are unreliable and since workstations are usually intended for primary use by some particular user, these parallel jobs must be adaptive: As resources change, the jobs must be able to expand or contract to match the environment. Empirical evidence shows that network workstations do generally remain idle—typically over half the time—so overcoming the problems of adaptive, secure, fault tolerant, parallel computing is indeed a worthwhile endeavor to provide a cheap, effective pool of computational power. This paper addresses the issues of running multiple parallel jobs on such a network of workstations; these issues include fault tolerance, security, detection of idleness, and fair and efficient distributed scheduling.

A natural division of the work of implementing such an adaptively parallel system is to differentiate "macroscheduling" and "microscheduling." Here, macroscheduling refers to recruiting idle workstations and assigning parallel jobs to them. Each job has its own microscheduler, which uses this varying set of workstations, assigned by the macroscheduler, to execute its single parallel job. This paper presents a macroscheduler [4] which was designed and implemented for use with the Cilk Network of Workstations (Cilk-NOW) microscheduler [1].

Fault Tolerance

The macroscheduler can recover from temporary network problems automatically, and workstation outages do not impede progress on a parallel job. Fault tolerance is also essential to ensure ease of use, as constant user intervention to recover from network and workstation faults is tedious and results in lost computation time. These fault tolerance properties are assured by the basic architecture of the system, as well as the distributed scheduling algorithm.

Security

The macroscheduler uses secure protocols that do not open a workstation to unauthorized users running foreign code on a machine. The desired degree of security is that which a given system uses to authenticate its remote execution (`rsh`) protocol. To provide this security, I have created an abstraction on top of active messages [5] called "secure" active messages which offers a quick, point-to-point, secure communication mechanism, which is then used to implement the various protocols in the macroscheduler.

Idleness

Indicators such as load averages and user idle times allow the macroscheduler to determine whether a machine is in use. Even with fairly strict idleness conditions, workstations tended to be available about 52% of the time according to an experiment run of the `theory` network at M.I.T. [4].

Scheduling

The objective of the scheduling algorithm is to achieve a "fair" allocation of processors to jobs in an efficient manner. Here, fair means that each job should have an equal share of the processors. The algorithm presented can be extended fairly easily to allow jobs to be assigned relative priorities. The distributed scheduler presented here, according to simulations, achieves a "near-fair" allocation, i.e., within one processor per job of a fair allocation, within expected $O(\log P)$ steps, where P is the number of processors (workstations).

The scheduling algorithm is random and distributed among the processors, wherever possible. This helps ensure reliability and fault tolerance, as it limits dependence on centralized, global components of the system. Furthermore, a distributed algorithm has the potential to work faster than a global algorithm since a global algorithm must talk to *all* of the processors, while a distributed algorithm only requires a given processor to communicate with a few others at one time. Some global information is permissible, however, such as what other processors are present for scheduling at any time, and how many processors are currently involved in each job. Note that since scheduling decisions are made in parallel and updated global values take time to propagate to all the processors, this global information may not be perfectly up-to-date; the scheduling algorithm must compensate for that effect.

The fundamental idea behind the scheduler is to have each processor periodically select another processor at random and see whether that processor's job is more in need of resources than the local job. If so, the job may be switched. A predefined, unsynchronized interval called a *quantum* is used to determine when to perform a scheduling step: Each processor engages in the scheduling algorithm once at some point during each quantum.

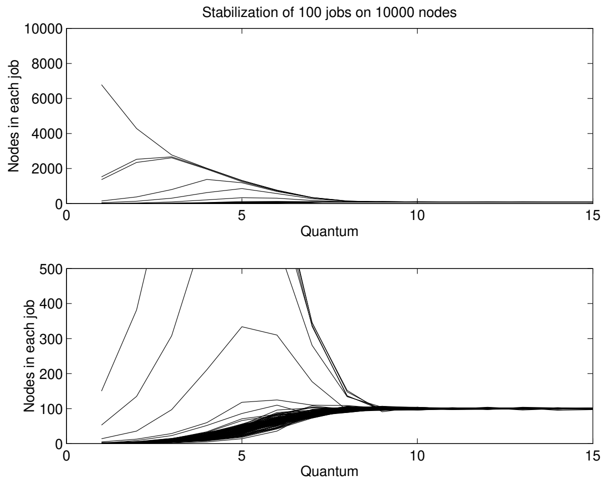

Figure 1: Example scheduling run.

The simplest imaginable switching policy is to switch whenever the other processor's job is more in need of resources. Unfortunately, the naivete of this scheme backfires rather severely since the processor counts may be out of date: So many processors switch jobs that the imbalance of processors switches polarity, often leaving the magnitude of the mismatch greater than it was to start. Simulations of this strategy reveal that it is completely unstable, often to the point that one job ends up with all the processors within a small number of quanta.

A slightly more complicated switching policy, but still inadequate, is to switch when an imbalance is seen with some predefined fixed probability, or with some probability determined by the total number of processors and/or the total number of jobs. The former strategy fails as the size of the scheduling problem that can be handled is determined by the value of the constant, while the latter does not achieve equilibrium quickly, as the constant must be low enough that overswitching is unlikely even when two processors are almost balanced; hence, the imbalance is taken up at *most* one processor per quantum, meaning this algorithm is $\Omega(P)$.

The actual algorithm employed uses a small piece of global information, but does not rely on it being completely up-to-date; namely, each processor knows approximately the number of processors, m_B, working on the locally running job. When a pairing occurs, the remote processor returns the approximate processor count, m_A, for its job. A switching probability of

$$\rho \quad = \quad \frac{m_B^2 - m_A^2}{2m_A m_B},$$

assures maximal progress toward equilibrium without overshooting in the case of only two jobs [4]. Simulations suggest that this algorithm is similarly stable and efficient when many jobs are present.

This scheduler converges to within one processor per job of the fair allocation in a short period of time, which simulations suggest to be expected $O(\log P)$. Figure 1 shows a typical run of the scheduling algorithm with 100 jobs on 10000 processors to. Note that the processors that begin significantly below the fair allocation rise exponentially until they are close to the fair distribution. Similarly, the processor with the highest initial allocation falls off exponentially until it is close to equilibrium. Some of the above-average processors initially move away from equilibrium, but when they hit the maximal job, they fall off with that job. Once the largest and smallest jobs are within about 50% of the fair equilibrium, all of the jobs converge neatly to the fair equilibrium.

Conclusion

Other research in scheduling on networks of workstations tends to focus on only one of the two levels of scheduling. Systems such as Piranha [2] focus on running a single job in parallel, using an oversimplified model to balance jobs among processors. Other systems, such as Condor focus only on running single-processor, nonparallel batch jobs on an idle workstation. (See [3] for a comparison of various batch job systems.) The former approach neglects the practicality that multiple users may want to exploit the network of workstations to accomplish their work, while the latter approach fails to utilize a significant portion of idle workstations when the number of jobs is not enough to occupy all of the workstations and generally relies on a centralized scheduler. By combining the Cilk-NOW adaptively parallel, fault-tolerant microscheduler with a distributed, fault-tolerant macroscheduler, Cilk-NOW offers the best of both worlds.

Currently, work is under way to provide a formal proof of the $O(\log P)$ convergence rate of an approximated version of this algorithm.

References

[1] Robert Blumofe. *Managing Storage for Multithreaded Computations*. PhD dissertation, Massachusetts Institute of Technology, 1995.

[2] David Gelernter and David Kaminsky. Supercomputer with recycled garbage: Preliminary experience with Piranha. Technical report, Yale University, December 1991.

[3] Jospeh A. Kaplan and Michael I. Nelson. A comparison of queueing, cluster and distributed computing systems. Technical report, National Air and Space Administration, June 1994.

[4] Philip Lisiecki. Macro-level scheduling in the cilk network of workstations environment. Master's thesis, M.I.T., May 1996.

[5] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th International Symp. on Computer Architecture*, Gold Coast, Australia, May 1992.