EFFICIENCY OF RANDOMLY ASSIGNING TASKS TO PROCESSORS

Wm. Randolph Franklin, Chandrasekhar Narayanaswami, Mohan S. Kankanhalli, and Varol Akman

BILKENT UNIVERSITY

Department of Computer Engineering and Information Science

Technical Report BU-CEIS-94-03

EFFICIENCY OF RANDOMLY ASSIGNING TASKS TO PROCESSORS

Wm. Randolph Franklin
Electrical, Computer, and Systems Eng. Dept.
Rensselaer Polytechnic Institute
Troy, New York 12180–3590, USA
E-mail: wrf@ecse.rpi.edu

Chandrasekhar Narayanaswami
IBM Corporation
11400 Burnet Road, Mail Stop 9260
Austin, Texas 78758, USA
E-mail: chandra@austin.ibm.com

Mohan S. Kankanhalli
Institute of Systems Science
National University of Singapore
Kent Ridge, Singapore 0511
E-mail: mohan@iss.nus.sg

Varol Akman
Dept. of Computer Eng. and Information Science
Bilkent University
Bilkent, Ankara 06533, Turkey
E-mail: akman@trbilun.bitnet

January 28, 1994

Abstract

We have implemented the uniform grid technique—due to the first author—on several parallel machines. It is complicated to globally equalize the load across the processors to correct for the fact that some grid cells have more data than others. Therefore, in our implementations we randomly assigned grid cells to different processors. The actual performance observed was very good and we obtained a near-linear speedup for many different algorithms. In this paper, we analyze the efficiency of this random assignment method of load balancing for parallel machines. The problem of analyzing the efficiency of leaving the loads uneven may be abstracted as follows.

Suppose that we have N independent tasks of unit weight and P processors to execute them. Assume that we are randomly, uniformly, and independently assigning each job to one of the processors. Since some processors will have more tasks than others, the efficiency, or the optimal time, N/P, divided by the expected time until the last processor finishes will be less than unity. This paper analyzes this and presents finite numbers and asymptotic equations. For example, $N = 1000$, $P = 100$ gives 53% efficiency. $N = 1000$, $P = 1000$ gives 18% efficiency. These efficiency figures are surprisingly high and they explain the good performance of our implementations.

1 Introduction

Efficient geometric algorithms are crucial in various key areas of industry such as VLSI design and CAD [9]. The *uniform (a.k.a. adaptive) grid* [3] spatial subdivision scheme divides the extent of a geometric scene uniformly into many smaller subregions (cells) of identical shape and volume. The idea is to exploit the limited spatial extent of individual geometric entities by inserting them into the subdivision and performing subregion-wise computations on them, thereby minimizing global computations. Figure 1 shows an example. Uniform grid has been used to develop and implement parallel algorithms for segment intersection, visible surface determination [5], Boolean operations on polygons and polyhedra, etc., on actual parallel machines such as the Sequent, CM-2, and the Intel iPSC/1 Hypercube [6].

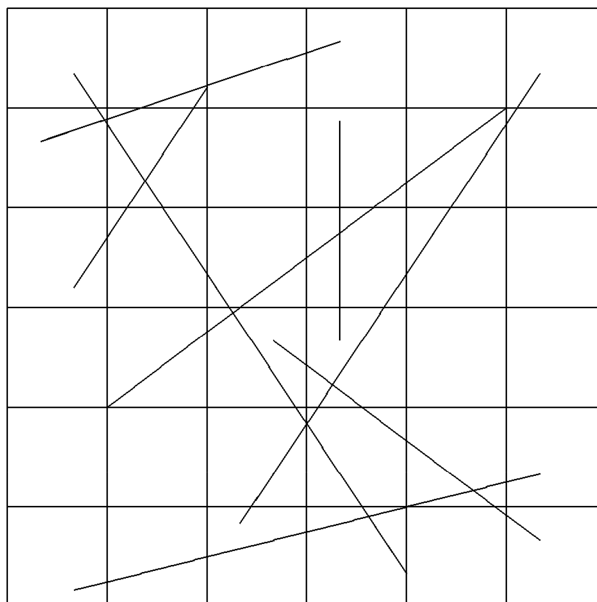

Figure 1: The uniform grid partitioning.

The problem of analyzing the efficiency of scheduling tasks by random assignment was motivated by a referee's query concerning implementing the uniform grid technique on a parallel machine [3, 7, 8, 10, 12]. We illustrate the motivation by considering a specific problem, line-segments intersection, to be solved using the uniform grid technique. Suppose that we wish to determine all intersections of a large number, say one million, of small line segments. The process is as follows.

1. Place, say, a 1000×1000 grid over the data.

2. For every edge, determine in parallel which cells each edge passes through.

3. Logically invert the resulting data structure so we know the edges in each cell.

4. For every cell, process in parallel each cell by comparing all its occupants pair-by-pair to determine intersections. If there are more cells than processors, the obvious solution is to group the cells into blocks. A better solution is to randomly assign cells to processors, so as to break up any dense clusters.

Although the edges are not randomly distributed, any correlations between them are local, and become relatively less important as their number grows. Indeed, the number of pairs of edges that are actually tested for intersection does track the theoretical number that would hold if the edges were random. So we assume randomness. Then the number of edges in any given cell follows a Poisson distribution. The problem is the time taken by the slowest step, 4 above, since some cells have more edges than others.

Random assignment has been used for memory allocation by Rettberg et al. [13]. This reference mentions the problem of hot spots and uses randomization of memory locations with address hashing to mitigate them. It also uses statistics of random arrivals to analyze the problem.

However, randomly assigning jobs to parallel processors is quite contrary to current practice, where much work is often spent to calculate the optimum assignment [14]. It would seem that random assignment just has to be intolerably inefficient. This paper shows that that is not so, even when there is one processor per job.

Sometimes the workload is distributed among the processors even more unevenly than this model gives, as can happen when ray-tracing and volume-rendering in computer graphics [2]. In this case, dynamic load-balancing—where each processor gets a new job from a master allocator whenever it finishes the previous one—is appropriate.

2 Analysis

We assume a PRAM (Parallel Random Access Machine) model of parallel computation [11]. Assume that we have N tasks, each of which takes unit time to execute, and which do not interact with each other. Assume that we have P processors on which to execute them. If we can assign N/P tasks to each processor, then all the processors will finish in time N/P, which is 100% efficient. Unfortunately, that may require more global planning than we can afford, so an alternate solution is to randomly and independently assign each task to some processor. This is not 100% efficient since some processors will have more tasks than others. How efficient is it?

Let the average number of tasks per processor, $\lambda = N/P$. By our assumption of independence, the probability distribution of tasks per processor is Poisson:

$$f_\lambda(k) = \frac{e^{-\lambda}\lambda^k}{k!}$$

A Poisson probability has $\mu = \sigma^2 = \lambda$. If $\lambda \gg 1$, then a normal approximation is reasonable. The probability distribution function, pdf, is the probability that the random variable is at most k:

$$F_\lambda(k) = \sum_{i=0}^{k} f_\lambda(i)$$

Unfortunately it has no closed form. However, it is monotonic and smooth. Note that for small λ, we must use the exact discrete distribution since the continuous approximation is very coarse. In fact, for $\lambda = 1$, while $\sum_{k=0}^{\infty} \frac{e^{-1}}{k!} = 1$, $\int_{k=0}^{\infty} \frac{e^{-1}}{k!} dk = 0.83$. For $\lambda = 3$, $\int_{k=0}^{\infty} \frac{e^{-3}3^k}{k!} dk = 0.98$, and for $\lambda = 5$, $\int_{k=0}^{\infty} \frac{e^{-5}5^k}{k!} dk = 0.998$.

We wish to know when the last of the P processors will finish. If we have P random variables, each with pdf F, then the pdf of their maximum is $G = F^P$. The probability density function of the completion time of the last processor is $g(i) = G(i) - G(i-1)$, and so $\overline{g}$, the expected time for the last processor to complete, can be calculated. The efficiency is then $\lambda/\overline{g}$. We used Maple [4] and represented f, F, g, G as discrete arrays truncated when $f_\lambda(i) = 10^{-8}$. For example, $f_{1000}(1172) \approx 10^{-8}$. Some sample results are shown in Table 1.

Jobs (N)	Processors (P)	Efficiency ($\lambda/\overline{g}$)
1	1	1.00
2	1	1.00
2	2	0.66
4	1	1.00
4	2	0.72
4	4	0.48
10	1	1.00
10	2	0.80
10	5	0.54
10	10	0.37
30	1	1.00
30	3	0.79
30	10	0.51
30	30	0.29
100	1	1.00
100	10	0.66
100	100	0.24
1000	1	1.00
1000	10	0.86
1000	100	0.53
1000	250	0.37
1000	1000	0.18
3000	1	1.00
3000	30	0.83
3000	100	0.67
3000	300	0.49
3000	1000	0.30
3000	3000	0.16

Table 1: Sample results for efficiency.

Of interest is the question of the maximum P that may be used for a given λ, and still achieve 50% efficiency, i.e., a completion time of 2λ. (It's easier to fix λ than to fix N.) Since λ will be large enough, we will approximate f as a normal density with

$\mu = \sigma^2 = \lambda$. The median is the mean under the normality assumption. Therefore we calculate the median which is easier to calculate. Thus we want $t_{med} \in G(t_{med}) = 1/2$. I.e., $F^P(2\lambda) = 1/2$.

If

$$Q(z) = \int_x^\infty \frac{1}{\sqrt{2\pi}} e^{-x^2/2} dx$$

then

$$F(2\lambda) = 1 - Q\left(\sqrt{\lambda}\right)$$

so we want P such that

$$Q\left(\sqrt{\lambda}\right) = 1 - 2^{-1/P}$$

Now,

$$Q(z) \approx \frac{e^{-x^2/2}}{x\sqrt{2\pi}} \left(1 + 1/x^2 + \cdots\right)$$

For large λ, we get

$$P \approx \frac{e^{\lambda/2}\sqrt{2\pi\lambda}}{\ln 2}$$

How good is this approximation? If $\lambda = 10$, this gives $P \approx 1200$. The actual efficiency for $\lambda = 10$, $P = 1200$, and hence $N = 12000$, is 46%.

What if $P = N$, so that λ, the average load, is 1? Let E be the efficiency. Then $G(1/E) = 1/2$, so that $F_{\lambda=1}(1/E) = 2^{-1/N}$. Thus E falls *very* slowly with increasing N, being about 0.12 at $N = 1000000$.

These efficiencies compare favorably to many parallel algorithms, whose efficiencies fall with $\log(N)$.

If the above efficiencies are too low, they may be improved by even a modest amount of load averaging. Assume that the P processors are grouped in blocks of L processors each. Within each block, the processors take jobs from a common queue, so that all the processors in one block finish at the same time. All the processors in another block will finish together at, probably, a different time. Since random Poisson variables add, the efficiency in this case is exactly the same as the efficiency with N jobs and P/L processors, with no averaging (although the actual time is a factor of L less). For example, with $N = 3000$, and $P = 3000$, the efficiency is 0.16, from Table 1. If the processors are grouped in pairs of three, then the efficiency rises to 0.30, while groups of ten give an efficiency of 0.49.

3 Summary

The above analysis proves that the uniform grid should work well in parallel, so our previous actual implementation results [10, 12] were not deceiving us.

This analysis would apply to any other geometric algorithm where it is not convenient to globally assign the work evenly to all the processors. In other words, central planning is not always necessary.

4 Acknowledgments

This research was funded by the NSF under grants ECS 83–51942 and CCR–9102553. We are grateful to Professor D. J. Hand for his initial encouragement and kind advice.

References

[1] M. Abramowitz and I. A. Stegun, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, U. S. Government Printing Office, 1972.

[2] B. Corrie and P. Mackerras, "Parallel Volume Rendering and Data Coherence on the Fujitsu AP1000," Technical Report TR–CS–92–11, The Australian National University, Department of Computer Science, Computer Sciences Laboratory, 1992.

[3] V. Akman, W. R. Franklin, M. Kankanhalli, and C. Narayanaswami, "Geometric Computing and Uniform Grid Technique," *Computer-Aided Design*, **21**, 7 (1989), pp. 410–420.

[4] B. Char, *First Leaves: A Tutorial Introduction to Maple V*, Springer-Verlag, 1992.

[5] W. R. Franklin and V. Akman, "Adaptive Grid for Polyhedral Visibility in Object Space: An Implementation," *Computer Journal*, **31**, 1 (1988), pp. 56–60.

[6] W. R. Franklin, C. Narayanaswami, M. Kankanhalli, M. Seshan, and V. Akman, "Efficiency of Uniform Grids for Intersection Detection on Serial and Parallel Machines," in *New Trends in Computer Graphics*, Eds. N. Magnenat-Thalmann and D. Thalmann, Springer-Verlag, 1988, pp. 288–297.

[7] W. R. Franklin, C. Narayanaswami, M. Kankanhalli, D. Sun, M. C. Zhou, and P. Wu, "Uniform Grids: A Technique for Intersection Detection on Serial and Parallel Machines," *Proc. Auto Carto 9: Ninth International Symposium on Computer-Assisted Cartography*, Baltimore, MD, 1989, pp. 100–109.

[8] W. R. Franklin and M. S. Kankanhalli, "Parallel Object-Space Hidden Surface Removal," *ACM Computer Graphics*, **24**, 4 (1990), pp. 87–94.

[9] W. R. Franklin, C. Narayanaswami, M. Kankanhalli, V. Akman, and P. Y. F. Wu, "Efficient Geometric Algorithms for CAD," in *Geometric Modeling for Product Engineering*, Eds. M. J. Wozny, J. U. Turner, and K. Preiss, Elsevier, 1990, pp. 485–498.

[10] M. S. Kankanhalli, "Techniques for Parallel Geometric Computations," Ph.D. Thesis, Electrical, Computer, and Systems Eng. Dept., Rensselaer Polytechnic Institute, Troy, NY, 1990.

[11] R. M. Karp and V. Ramachandran, "Parallel Algorithms for Shared-Memory Machines," in *Handbook of Theoretical Computer Science: Volume A—Algorithms and Complexity*, Ed. J. van Leeuwen, Elsevier, 1990, pp. 869–942.

[12] C. Narayanaswami, "Parallel Processing for Geometric Applications," Ph.D. Thesis, Electrical, Computer, and Systems Eng. Dept., Rensselaer Polytechnic Institute, Troy, NY, 1990.

[13] R. D. Rettberg, W. R. Crowther, P. P. Carvey, and R. S. Tomlinson, "The Monarch Parallel Processor Hardware Description," *Computer*, **23**, 4 (1990), pp. 18–30.

[14] *Journal of Parallel and Distributed Computing*, Special Issue on "Scheduling and Load Balancing," **16**, 4 (1992).