ELSEVIER Theoretical Computer Science 162 (1996) 341–349

Theoretical Computer Science

Random allocation of jobs with weights and precedence

Soumen Chakrabarti*

Computer Science Division, University of California, Berkeley, CA 94720, USA

Abstract

We analyze random allocation applied to irregular and dynamic task-parallel programs such as branch and bound. The precedence between jobs is revealed on-line, and the processing times of jobs are diverse and unknown before job completion. The objective is to assign jobs to processors and to schedule them to minimize makespan. We show that random allocation achieves makespan close to a natural lower bound. Some empirical experience with irregular parallel applications is reported.

1. Introduction

We analyze the performance of random allocation schemes applied to irregular and dynamic task-parallel programs. Execution of the program defines a job precedence graph with vertices representing jobs and directed edges representing precedence constraints. The precedence graph is revealed on-line and is irregular in shape, and the processing times of jobs are diverse and unknown before job completion. The objective function to minimize is makespan, the maximum completion time of a job. Although the exact problem is $\mathcal{NP}$-hard, good approximations are possible if the algorithm assigning jobs to processors is centralized, and thus has perfect global knowledge. For example, Graham's list-scheduling algorithm [9] will result in a finish time at most $2 - 1/P$ times optimal, where P is the number of processors. This, however, leads to a severe communication bottleneck at the processor where the pool of jobs resides. Our goal, therefore, is to study decentralized allocation which avoids such bottlenecks.

The bottleneck can be relieved in a variety of ways, each of which reduces communication cost by sacrificing global load information and thus risking some load imbalance. We study *work sharing*, where busy processors forward jobs to random

* E-mail: soumen@cs.berkeley.edu. Supported partly by ARPA/DOD (DABT63-92-C-0026), DOE (DE-FG03-94ER25206), and NSF (CCR-9210260, CDA-8722788 and CDA-9401156).

S0304-3975/96/$15.00 © 1996—Elsevier Science B.V. All rights reserved
PII S0304-3975(96)00036-9

processors. Some other techniques are *work stealing*, where idle processors ask for work [5], and *diffusion*, where neighbors exchange local load information and then move some jobs from busy to lazy processors [8].

1.1. Models and notation

The input comprises a set J of jobs presented to the algorithm in a distributed and on-line fashion. Job j has running time t_j, also referred to as its "weight". We assume these are powers of 2; this will affect the results only in constant factors. We assume that t_j can be known only when job j completes. The total work or weight in a job set J is denoted $t(J) = \sum_{j \in J} t_j$. The number of jobs in J is denoted $|J|$ or $n(J)$. The average job weight is $\bar{t}(J) = t(J)/|J|$. Let $t_{\max}(J) = \max_{j \in J}\{t_j\}$ and $t_{\min}(J) = \min_{j \in J}\{t_j\}$. We characterize the diversity using a single parameter $T(J) = t_{\max}(J)/t_{\min}(J)$. Equivalently, we scale jobs so that $t_{\min} = 1$ and $t_{\max} = T$. This is in keeping with recent analyses of online load-balancing algorithms [4], and is more broadly applicable than results with assumptions about distribution or variance.

J will have an associated acyclic precedence relation $\prec \subset J \times J$. Let $\pi_j = \max_{j' \prec j}\{\pi_{j'}\} + t_j$ and let $\pi(J) = \max_{j \in J}\{\pi_j\}$ be the critical path. We assume there is a unique root job. The number of edges on a path from j to the root is denoted $h(j)$; the path will be clear from context. Also let $h(J)$ be the maximum number of edges on any precedence path in J.

J will be omitted when clear from context. We assume that the job times and job graph are oblivious of the decisions made by the scheduler. In exhaustive traversal, J is finite and the goal is to execute all jobs in J in any order obeying $\prec$. In heuristic search or branch and bound, the job graph J provided may be very large or even infinite. Each job j has an associated cost $c(j)$, with the requirements that $j_1 \prec j_2 \Rightarrow c(j_1) < c(j_2)$ and all costs are distinct without loss of generality. The goal of the execution is to start at the root and execute jobs obeying $\prec$, until the leaf node with minimum cost c^* is identified. It is not necessary to generate and execute all jobs in J.

Sequential algorithm. A common sequential strategy is the "best first" traversal. All available jobs with completed predecessors are maintained in a priority queue. While the queue is nonempty, the job j with least $c(j)$ is removed and executed. The jobs executed are $\tilde{J} = \{j \in J\colon c(j) < c^*\} \subseteq J$. In the special case of complete traversal, $\tilde{J} = J$. For all models, we assume that operations on a priority queue for job selection take negligible time compared to job execution time. In this model, the sequential algorithm takes time $t(\tilde{J})$. We assume $|J| \geqslant |\tilde{J}| \geqslant P$.

Parallel algorithm. Parallel execution starts with the root job in one processor. A job can be started when all predecessors have been completed. When a processor completes executing a job j, all successors of j become available to that processor. Processors can negotiate to transfer available jobs among themselves.

There is no coordinated global communication for load-balancing purposes. We study the setting with a local priority queue of jobs in the memory of each processor as

in [10]. Thus, priority is preserved within each local queue but not across processors. An idle processor nonpreemptively executes the best job from its local queue, if any. Any newly available job with completed predecessors is enqueued into the priority queue of a processor chosen uniformly at random. The destination processor is not interrupted.

Communication. The machine model consists of processors with individual local memory connected by a communication network. We ignore the topology of the interconnect as in the LogP model [6]. Communicating a job takes unit time at the two processors involved in the transfer.

Pruning and termination. In parallel branch and bound, each processor has to periodically propagate the cost of the least cost leaf it has expanded, so that all processors know the cost of the global best cost leaf in order to use it for pruning. Also, barrier synchronizations are required to detect situations where all local queues are empty so that the processors can terminate. We note that these can be done infrequently with low overhead, so they do not affect the time bounds we derive.

1.2. Discussion of results

For exhaustive search $\tilde{J} = J$, and greedy centralized schedules give a simple bound on makespan in terms of variables defined above: $\Theta(t/P + \pi)$. We show that the situation changes somewhat in the branch and bound setting: $\Omega(t(\tilde{J})/P + h(\tilde{J}) \cdot T(J))$ may be necessary even for an ideal centralized scheduler with no communication cost. Then we give a delay sequence type analysis of parallel branch and bound with $\tilde{J} \neq J$ in a complete network: we show that with probability at least $1 - \varepsilon$, the makespan is $O(t/P + hT\log hT + T\log(n/\varepsilon))$, where $t = t(\tilde{J})$, $h = h(\tilde{J})$, $n = n(\tilde{J})$, and $T = T(J)$. We also report on experience with some irregular programs. This is necessary for two reasons. First, our analysis is probabilistic and asymptotic; in practice, constant factors would be important. Second, although the above result establishes near-optimal load balance, our model does not reflect the gains from avoiding communication bottlenecks.

Other results of the diffusion type are based on occasionally matching busy and idle processors and transferring jobs [8, 13]. These are not appropriate for relatively fine-grain jobs which is our focus. Notice also that diversity in job execution times makes coordination even harder unless a processor can suspend long jobs and participate in global communication.

Work stealing is the strategy of least communication for the particular case where the job graph J is an out-tree, all jobs must be executed, and the relative order of execution is immaterial (provided it obeys $\prec$). In work stealing, the graph is expanded depth-first locally in each processor, and idle processors steal jobs nearest to the root [14, 5]. In many application such as parallel search or branch and bound, the total work done is very sensitive to the job order, and one wishes to deviate from the best sequential order as little as possible [7, 2].

2. Analysis

2.1. Weighted occupancy problem

At first we consider the weighted occupancy problem, where there is no precedence among jobs. There are n weighted balls (jobs), ball j having weight t_j. These balls are thrown uniformly at random into P bins (processors). We want to bound the weight of the heaviest bin.

Lemma 1. *For random allocation of weighted balls to bins, with probability at least* $1 - \varepsilon$, *each bin has* $\mathrm{O}(t/P + T(\log\log T + \log P/\varepsilon))$ *weight.*

Proof. Classify the balls into weights $1, 2, \ldots, T$, where there are n_i balls of weight 2^i. Fix one bin. The probability that there are at least m_i balls of weight 2^i in this bin is at most $\binom{n_i}{m_i}P^{-m_i} \leqslant (en_i/Pm_i)^{m_i}$, which is less than $\varepsilon/P\log T$ for $m_i = \mathrm{O}(n_i/P + \log\log T + \log P/\varepsilon)$. Adding over i and all P bins gives the result. $\square$

2.2. The scheduling problem

First we show that in case of branch and bound, permitting diverse job times does change the setting from the exhaustive traversal, or the unit time case. Let J and $\tilde{J}$ be as before. For unit time jobs, $\Omega(n(\tilde{J})/P + h(\tilde{J}))$ is a lower bound that can be achieved by a central scheduler. For diverse times, the analogous bound of $\mathrm{O}(t(\tilde{J})/P + \pi(\tilde{J}))$ is not always achievable.

Lemma 2. *With a centralized scheduler, the execution time for branch and bound is* $\Theta(t(\tilde{J})/P + h(\tilde{J})T(J))$.

Proof. For the lower bound we will produce a J and $\hat{J} \subset J$ with $t(\tilde{J}) = \mathrm{o}(\pi(\tilde{J}))$ such that even a centralized scheduler will need $\Omega(h(\tilde{J})T(J))$ time. The instance is shown in Fig. 1(a). In the first time-step, one processor expands the root job, generating P children that all P processors start expanding at the second time-step. $P - 1$ of these are jobs in $J\setminus\tilde{J}$ with $t_j = T$, meant to keep $P - 1$ processors busy for time T, so the last processor is left alone to expand part of $\tilde{J}$ as shown. In the figure, job x has P children and job y has $T - 1$, so that when the new set of P nodes are generated, the $P - 1$ processors just freed grab the new decoys. This can be arbitrarily repeated.

For the upper bound, suppose the makespan is τ and s is a job finishing at time τ. Label s as $s_{h(s)}$ and starting at s, move up towards the root. If a task s_ℓ has more than one parent, go to the one that completed last of all parents, and call it $s_{\ell-1}$. Thus, trace a path root $= s_1, \ldots, s_{h(s)} = s$. Note that $\sum_\ell t_{s_\ell} \leqslant \pi(s)$. Suppose s_ℓ runs in the interval $[B_\ell, E_\ell]$. Note that s_ℓ appears in the central job pool at time $E_{\ell-1} + 1$, and in the interval $[E_{\ell-1} + T, B_\ell]$, if nonempty, all processors are executing jobs inside $\tilde{J}$ since they all picked some other jobs j' with $c(j') < c(s_\ell)$, meaning $j' \in \tilde{J}$. Thus, $P(\tau - \pi(\tilde{J}) - T \cdot h(\tilde{J})) \leqslant t$, which proves the claim since $\pi = \mathrm{O}(hT)$. $\square$

Next, we show an upper bound for random allocation. Unlike in the previous section, occupancy results cannot be used directly, since unlike a batch, a DAG schedule is not composable from arbitrary task subsets. Further, in analyses related to global task pools as above, arguments depend significantly on statements to the effect that during certain intervals of time, most processors do useful work. We can no longer say this when each processor has a local task pool: processors can remain idle even though there are tasks yet to expand, because they can be in the queues of other processors. We handle this using a delay sequence argument. The following lemma is similar to Ranade's construction [12].

Lemma 3. *Suppose the execution finishes at time τ. Then the following 4-tuple (s, Q, R, Π) exists*:

- *s is a job that finished no earlier than τ. Let $S = (s_1, \ldots, s_{h(s)})$ be a path of "special" jobs from the root of the DAG to job $s = s_{h(s)}$.*
- *$Q = (q_1, \ldots, q_{h(s)})$ is an ordered list, where q_ℓ is the processor that executed s_ℓ, for $1 \leqslant \ell \leqslant h(s)$.*
- *$R \subset \tilde{J}$ is a set of jobs, and*
- *$\Pi_1, \ldots, \Pi_{h(s)}$ is a partition of $[1, \tau]$ such that*
  - *$R \cap \{s_1, \ldots, s_{h(s)}\} = \emptyset$.*
  - *Each job in R become "ready" and arrives into q_j during interval Π_j,* for some j, $1 \leqslant j \leqslant h(s)$.
  - *$t(R) \geqslant \tau - \pi(J) - h(\tilde{J})T(J)$.*

Proof. Label s as $s_{h(s)}$ and starting at task s, move up towards the root. If a task s_ℓ has more than one parents, go to the one that completed last of all parents, and call it $s_{\ell-1}$. Thus, trace a path root $= s_1, \ldots, s_{h(s)} = s$.

To obtain R, work backwards from the time τ and consider the latest time instant $\tau' < \tau$ such that $q_{h(s)}$ was empty at time $\tau' - 1$. This means that all tasks executed by $q_{h(s)}$ during the interval $[\tau', \tau]$ also arrived there during this interval. Call these tasks $R_{h(s)}$. Include $R_{h(s)}$ into R, and set $\Pi_{h(s)} = [\tau', \tau]$. Then continue the construction from $\tau' - 1$ in an iterative manner.

From Fig. 1(b), it can be seen that the processing times of nodes in R must cover all of $[1, \tau]$, except for the time spent in processing nodes $s_1, \ldots, s_{h(s)}$, which is at most $\pi(\tilde{J})$, and decoy jobs, which accounts for at most $h(\tilde{J})T(J)$. Thus, let $R = \bigcup_\ell R_\ell$ and observe that $t(R) \geqslant \tau - \pi(\tilde{J}) - h(\tilde{J})T(J)$. We have also constructed a ordered partition $\Pi = (\Pi_1, \ldots, \Pi_{h(s)})$ of $[1, \tau]$. □

Theorem 4. *With probability at least* $1 - \varepsilon$, *the execution time for branch and bound is* $O(t/P + hT\log hT + T\log(n/\varepsilon))$, *where* $t = t(\tilde{J})$, $h = h(\tilde{J})$, $n = n(\tilde{J})$, and $T = T(J)$.

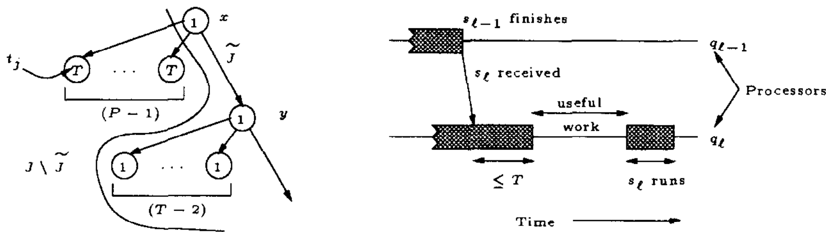

Fig. 1. (a) A bad instance for the central scheduler. (b) Accounting for time in the delay sequence.

Proof. We will bound the probability over all s, Q, Π and R that a 4-tuple as above will occur. Given fixed values for s, Q, Π, R and τ, a conforming execution happens with probability at most $P^{-(h(s)+|R|)}$. Thus, our target expression is $\sum_{s,Q,\Pi,R} P^{-(h(s)+|R|)} = \sum_{s,\Pi,R} P^{-|R|}$, since, given s, the number of choices for Q is just $P^{h(s)}$. In the sum $\sum_{s,\Pi,R} P^{-|R|}$, s can be chosen in n ways. The number of ways to pick Π is at most $\binom{\tau+h}{h} \leqslant (6\tau)^h$. It only remains to evaluate $\sum_R P^{-|R|}$, where the sum is over all R such that $t(R) \geqslant \tau - \pi(\tilde{J}) - h(\tilde{J})T(J)$. $\sum_R P^{-|R|}$ is the probability, over all $R_1, \ldots, R_{h(s)}$, that R_ℓ got assigned to q_ℓ, $1 \leqslant j \leqslant h(s)$. This is the same as the probability that some bin gets a weight of at least $\tau - \pi(\tilde{J}) - h(\tilde{J})T(J)$ when $n - h(s)$ balls were randomly assigned to P bins. This can be bounded using Lemma 1.

Specifically, setting $\tau - \pi - hT = \Omega(t/P + T(\log\log T + \log(P/\delta)))$ bounds the probability of makespan being τ to $n \cdot (6\tau)^h \cdot \delta$, which we need to be less than ε. It will suffice to pick τ such that $\tau > \Omega(t/P + hT + T\log(nP(6\tau)^h \log T/\varepsilon))$; this holds for the choice of the makespan in the statement of the theorem. □

3. Experience

Unlike in analyses of centralized schemes, our results are probabilistic and hide constants at several places. It is therefore interesting to evaluate the cost and benefit of decentralization in practical settings. We report on experiments with two applications. The first is a parallel symmetric tridiagonal eigenvalue solver. The second is a parallel symbolic multivariate polynomial equation solver. Both our applications lead to tree-shaped precedence between jobs, and the job times are diverse. In both cases, most of shared data can be replicated at small communication cost, so random allocation is feasible. Random allocation with diverse time have also been use in N-body simulation [11] and integer linear programming [7].

For each application, we added instrumentation to the sequential program to emit the task tree with task times, and input this tree to a simulator that simulated the

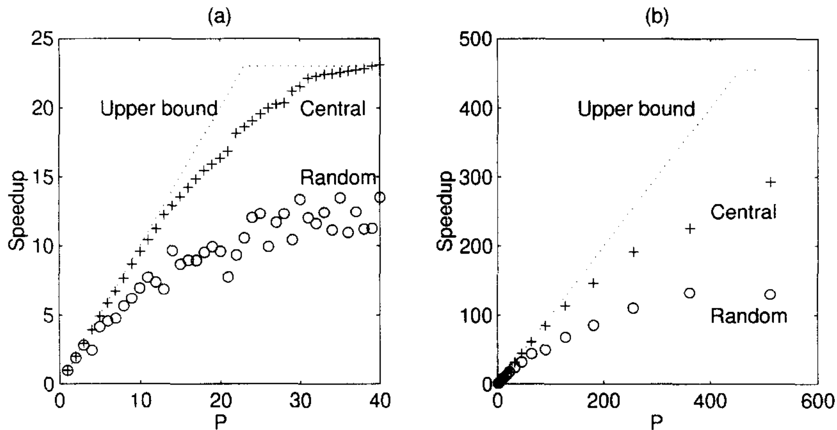

Fig. 2. Comparison of speedup between Graham's list schedule and random allocation with zero communication cost: (a) symbolic algebra; (b) eigensolver.

parallel execution of the randomized load-balancing algorithm, as well as Graham's list schedule. By an idealized simulation without communication cost and other overheads, we first isolate and study only the loss in load balance owing to random allocation. In Fig. 2, the eigensolver instance has $t = 108\,247\,480$ μs, $n = 2999$, $h = 20$, $T = 184\,377$ μs $\div$ 4486 μs ≈ 41, $\pi = 237917$ μs, and $t/\pi \approx 455$. The symbolic algebra instance has $t = 11\,053\,339$ μs, $n = 142$, $h = 11$, $T = 174\,860$ μs $\div$ 1184 μs $\approx$ 148, $\pi = 474\,880$ μs, and $t/\pi \approx 23$.

In the graphs shown in Fig. 2 we have presented the speedup without explicitly measuring communication costs. We also measured actual speedup on the CM5 multiprocessor. Comparing the speedup curves enabled us to judge the closeness of the simulation to reality. In Fig. 3, we present a break-up analysis of parallel running time for the eigensolver, comparing a centralized task queue with the random distributed allocation. Although load imbalance is larger for random allocation, the benefit due do decentralization is overwhelming.

4. Extensions

Several questions arise from this and related results. It would be interesting to tighten the makespan estimates in this paper, as well as to provide randomized lower bounds. The performance of multiple round strategies [3, 1] for dynamic job graphs remains open, as is their effect on further reducing the load imbalance. Finally, extending the results to capture network contention as in the atomic message model [11] seems like a natural goal.

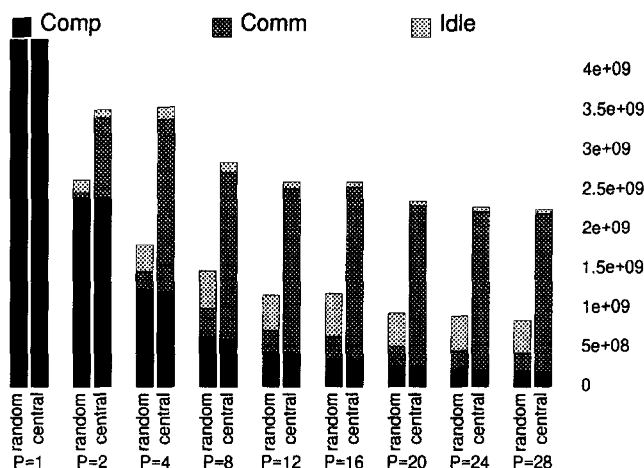

Fig. 3. Performance comparison of Graham's list schedule and random allocation on the CM5. The y-axis represents time, broken down into computation, communication for accessing the job pool(s), and idle time.

Acknowledgements

Thanks to Abhiram Ranade and Kathy Yelick for helpful discussions and the anonymous referees for comments on presentation.

References

[1] M. Adler, S. Chakrabarti, M. Mitzenmacher and L. Rasmussen, Parallel randomized load balancing, in: *Proc. Symp. on the Theory of Computing* (*STOC*) (ACM, New York, 1995).

[2] G. Attardi and C. Traverso, Strategy-accurate parallel buchberger algorithms, in: *Proc. Internat. Conf. on Parallel Symbolic Computation*, 1994.

[3] Y. Azar, A Border, A. Karlin and E. Upfal, Balanced allocations, in: *Proc. Symp. on the Theory of Computing* (*STOC*) (ACM, New York, 1994).

[4] Y. Azar, B. Kalyanasundaram, S. Plotkin, K. Pruhs and O. Waarts, Online load balancing of temporary tasks, in: *Workshop on Algorithms and Data Structures* (*WADS*) Lecture notes in Computer Science (Springer, Berlin, 1993) 119–130.

[5] R. Blumwofe and C. Leiserson, Scheduling multithreaded computations by work stealing, in: *Foundations of Computer Science* (*FOCS*), Santa Fe, NM (IEEE, New York, 1994).

[6] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Sumbramonian and T. von Eicken, Logp: towards a realistic model of parallel computation, in: *Principles and Practice of Parallel Programming* (*PPoPP*) (ACM-SIGPLAN, 1993) 1–12.

[7] J. Eckstein, Parallel branch and bound algorithms for general mixed integer programming on the CM-5, *SIAM. J. Optim.*, to appear.

[8] B. Ghosh and S. Muthukrishnan, Dynamic load balancing on parallel and distributed networks by random matchings, in: *Proc. Symp. on Parallel Algorithms and Architectures* (*SPAA*), Cape May, NJ, (ACM, New York, 1994)

[9] R.L. Graham, Bounds on multiprocessor timing anomalies, *SIAM J. Appl. Math.* **17** (1996) 416–429.

[10] R.M. Karp and Y. Zhang, A randomized parallel branch-and-bound procedure, *J ACM* **40** (1993) 765–789. Preliminary version in *ACM STOC* (1988) 290–300.

[11] P. Liu, *The Parallel implementation of N-body algorithms*, Ph.D Thesis, DIMACS Center, Rutgers University, Piscataway, NJ 08855-1179, May 1994. Also available as DIMACS Tech. Report 94–27.
[12] A. Ranade, A simpler analysis of the Karp-Zhang parallel branch-and-bound method, Tech. Report UCB/CSD 90/586, University of California, Berkeley, CA 94720, August 1990.
[13] L. Rudolph, M. Slivkin-Allalouf and E. Upfal, A simple load balancing scheme for task allocations in parallel machines, in: *Proc. Symp. on Parallel Algorithms and Architecture (SPAA)* (1991) 237–245.
[14] L.-C. Wu and H.T. Kung, Communication complexity for parallel divide-and-conquer, in: *Foundations of Computer Science (FOCS)* (1991) 151–162.