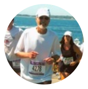
Empirical study of dynamic scheduling on rings of processors

ARTICLE *in* PARALLEL COMPUTING · SEPTEMBER 1999

Impact Factor: 1.51 · DOI: 10.1016/S0167-8191(99)00039-3 · Source: DBLP



Arnold Rosenberg
Northeastern University
287 PUBLICATIONS **5,131** CITATIONS

Paul R. Cohen
The University of Arizona
373 PUBLICATIONS **5,110** CITATIONS

An Empirical Study of Dynamic Scheduling on Rings of Processors[*]

Miranda E. Barrows[§] Dawn E. Gregory[†] Lixin Gao[‡] Arnold L. Rosenberg[§] Paul R. Cohen[§]

Abstract. We empirically analyze and compare two low-overhead, deterministic policies for scheduling dynamically evolving tree-structured computations on rings of identical processing elements (PEs). Our computations have each task either halt or spawn two independent children and then halt; they abstract, for instance, computations generated by multigrid methods. Our simpler policy, KOSO, has each PE keep one child of a spawning task and pass the other to its clockwise neighbor in the ring; our more sophisticated policy, KOSO*, operates similarly, but allows child-passing only from a more heavily loaded PE to a more lightly loaded one. Both policies execute waiting tasks in increasing order of their depths in the evolving task-tree. Our study focuses on two conjectures: (*a*) Both policies yield good parallel speedup on large classes of the computations we study. (*b*) Policy KOSO* outperforms policy KOSO in many important situations. We verify these conjectures via a suite of experiments, supplemented by supporting mathematical analyses. We view our methodology of experimental design and analysis as a major component of our study's contribution, which will prove useful in other such studies.

1 Introduction

The promise of parallel computers to accelerate computation relies on an algorithm designer's ability to keep all (or most) of a computer's processors fruitfully occupied all (or most) of the time. The problem of balancing computational loads so as to approach this goal has received considerable attention for decades. In this paper, we study the problem of efficiently scheduling dynamically evolving tree-structured computations, on parallel computers whose underlying structure is a ring of identical processing elements (PEs). The challenge of efficiently scheduling such computations on such computers resides in the following facts.

• The dynamic nature of our computations precludes knowledge of the ultimate "shape" of their task-dependency graphs; such knowledge can help one devise efficient schedules [3, 4].

• The large diameters and communication latencies of rings render both randomized load-balancing and PRAM-like algorithms unacceptably time consuming; when efficient, such devices aid in devising good schedules [7, 9, 10].

• The low communication bandwidths of rings render massive transfers of data unacceptably

[*] A portion of this paper appeared at the *8th IEEE Symp. on Parallel and Distributed Processing* (1996) 470–473.

[§] Dept. of Computer Science, Univ. of Massachusetts, Amherst, MA 01003, USA
Email: {`miranda, rsnbrg, cohen`}`@cs.umass.edu`
[†] School of Cognitive Science, Hampshire College, Amherst, MA 01002, USA
Email: `dgregory@hampshire.edu`
[‡] Dept. of Computer Science, Smith College, Northampton, MA 01060, USA
Email: `gao@cs.smith.edu`

time consuming; such transfers can lead to good schedules [5, 9].

• The tight interrelationships among our tasks—as reflected in the dependency structures of our computations—impose (load-balancing and scheduling) constraints that do not appear in activities such as job scheduling (cf. [1]).

The computations we study have the structure of dynamically evolving binary trees in which each executed task-node either halts (thereby becoming a leaf) or spawns exactly two child-tasks. Algorithms that lead to such structure include branch-and-bound and game-tree algorithms (cf. [6, 7, 8, 10]) and multigrid algorithms. To illustrate the latter, consider numerical integration algorithms that use Simpson's Rule or the Trapezoid Rule: The initial task—the root of the task-tree—represents the interval over which the integration takes place. A task spawns when the approximation over the current interval is too coarse; its two child-tasks each represent half of the parent-task's interval. A task halts when either the approximation is sufficiently accurate or the resolution of the host machine renders further subdivision of its interval useless.

We report here on an empirical evaluation and comparison of two low-overhead, deterministic scheduling policies. Policy KOSO, for "Keep One-Send One," has each PE keep one child-task of a spawning task and send the other to its clockwise neighbor. Policy KOSO* tries to balance loads better than KOSO, by having a PE send a spawned child-task to its neighbor only when the neighbor has a lighter computational load; otherwise, the PE keeps both child-tasks.

Our main contribution is the design and analysis of two experiments that supply strong evidence for the following conjectures—at least for large classes of dynamic tree-structured computations.

Conjectures. **1.** *Policies* KOSO *and* KOSO* *both yield close to optimal parallel speedup on large, significant classes of computations.*
2. *Policy* KOSO* *produces significantly better schedules than policy* KOSO*, except on very small processor rings.*

Section 2 describes our computational setting in detail. In Section 3, we present two analytical results which lend evidence for our conjectures. Section 4 is devoted to the experiments which validate important cases of our conjectures: on an artificial computational load and on a simulated real load. A combination of exploratory data analysis and hypothesis testing with variance-reducing transformations leaves little doubt that KOSO* is superior to KOSO in a wide range of experimental conditions. As importantly, these empirical results tell us *why* KOSO*'s schedules enjoy better parallel speedup. We view this explanatory component of our methodology as an essential part of our study. Indeed, one lesson of this paper is that empirical studies can augment formal analyses with rich detail about why algorithms perform as they do—and may provide evidence about algorithms in conditions that are difficult or impossible to analyze.

2 The Formal Setting of Our Study

2.1 The Computational Model

The Architecture. We focus on rings of PEs. The p-PE version $\mathbf{R}_p$ of this architecture has p identical PEs, denoted $\mathcal{P}_0, \mathcal{P}_1, \ldots, \mathcal{P}_{p-1}$, with each PE $\mathcal{P}_i$ connected to its *clockwise* neighbor

$\mathcal{P}_{i+1 \bmod p}$ and its *counterclockwise* neighbor $\mathcal{P}_{i-1 \bmod p}$.

The Computational Load. A *binary tree-dag* (*BT*, for short) $\mathcal{T}$ is a directed binary tree with root-toward-leaf arcs. The root of $\mathcal{T}$ resides at level 0; each nonleaf node x of $\mathcal{T}$ has a left child $L(x)$ and a right child $R(x)$ which both reside at level $\text{LEVEL}(x) + 1$. The (dynamic) computation that generates $\mathcal{T}$ proceeds as follows, until no active leaves remain.

1. Initially, $\mathcal{T}$ contains only its root, which is its (current) *active* leaf.
2. Inductively, the tasks corresponding to some of the then-current active leaves (the particular subset depending on the scheduling policy) get *executed*. An executed task/leaf may:
   (a) *halt*, thereby becoming a *permanent* leaf,
   (b) *spawn* two new active leaves, thereby becoming a nonleaf.

2.2 Policies KOSO and KOSO*

Policies KOSO and KOSO* differ in their load-balancing regimens but share the same scheduling strategy. Both have PEs retain tasks awaiting execution in local priority queues, ordered by the tasks' height in the BT being scheduled. Initially, the root of the evolving BT $\mathcal{T}$ is the sole occupant of PE $\mathcal{P}_0$'s task-queue, and all other PEs' task-queues are empty. At each step, the task-queue of each PE contains some subset of the then-active leaves of $\mathcal{T}$. Each $\mathcal{P}_i$ having a nonempty task-queue performs the following actions.

1. $\mathcal{P}_i$ executes that active leaf x in its task-queue which is first in the mandated order.
2. If leaf x spawns two children, then $\mathcal{P}_i$ adds $L(x)$ to its task-queue. Under KOSO, it simultaneously sends $R(x)$ to the task-queue of PE $\mathcal{P}_{i+1 \bmod p}$. Under KOSO*, $\mathcal{P}_i$ sends $R(x)$ to $\mathcal{P}_{i+1 \bmod p}$ only if the latter has a lighter load than $\mathcal{P}_i$; otherwise, $\mathcal{P}_i$ adds this task also to its own task-queue.

Notably, neither policy makes any assumption about the eventual shape of the evolving BT.

We assess one time unit for the entire process of executing a task and performing the balancing actions just described. Thus, we ignore the fact that a step of KOSO* consumes a bit more real time than does a step of KOSO, because of the required transmissions and comparisons of loads.

3 Analytical Evidence for the Conjectures

While we have not been able to establish our conjectures about KOSO and KOSO* analytically, we have established two results that support the conjectures. The first result concerns the *load disparity* under the two policies—the difference in the numbers of unexecuted tasks residing in the heaviest and lightest loaded PEs of $\mathbf{R}_p$.

Theorem 1. *Focus on a dynamic BT in which every executed task spawns two new tasks.*
(a) *Under policy* KOSO*, after $N \geq p-1$ steps, the load disparity is exactly $p-2$.*
(b) *Under policy* KOSO**, after $N \geq (p-1)^2$ steps, the load disparity is exactly* 1.

Proof. Since every executed task spawns two new tasks, and since an idle PE always accepts a task offered by its counterclockwise neighbor, it should be obvious that after $p-1$ steps of either policy, every PE contains at least one task that is eligible for execution. Less obviously, the load disparity at step p is exactly $p-2$. To wit, when $\mathcal{P}_{p-1}$ first receives a task to execute, the work profile within $\mathbf{R}_p$ is as follows: $\mathcal{P}_0$ contains a single task, while each other $\mathcal{P}_i$ contains $p-i$ tasks. Under KOSO, at each subsequent step, each PE receives one additional task, so the load disparity never changes, whence part (a).

We establish part (b) via three observations about KOSO*.

1. The first time a PE has nonzero load, its load is 1.
2. The load of $\mathcal{P}_0$ remains 1 during the first $p-1$ steps of the computation.
3. The load of the heaviest-loaded PE never increases by more than 1.

Focus on an arbitrary step $t > p-1$; let $\mathcal{P}_i$ be a heaviest loaded PE and $\mathcal{P}_j$ a lightest loaded PE at step t. During step t: $\mathcal{P}_i$'s load either stays the same (if $\mathcal{P}_{i+1 \bmod p}$ has a lighter load than $\mathcal{P}_i$) or increases by 1 (if $\mathcal{P}_{i+1 \bmod p}$ is not lighter); similarly, $\mathcal{P}_j$'s load either increases by 1 (if $\mathcal{P}_{j-1 \bmod P}$ has the same load as $\mathcal{P}_j$) or increases by 2 (if $\mathcal{P}_{j-1 \bmod p}$ has a heavier load). Therefore:

1. If each lightest-loaded PE has a more heavily loaded counterclockwise neighbor, then the load disparity decreases by either 1 or 2 at step t.
2. If some lightest loaded PE does not have a more heavily loaded counterclockwise neighbor, then, although the load disparity may not decrease at step t it cannot increase; and, the number of lightest loaded PEs definitely decreases.

Since there are at most $p-1$ lightest loaded PEs, alternative #1 must hold at least every $p-1$ steps. Since the disparity at step p is at most $p-2$, it follows that after $\leq (p-1)^2$ steps, the load disparity can be no greater than 1. In fact, the disparity is exactly 1, since any unit-disparity configuration having a single lightest loaded PE produces another such configuration. ■

Theorem 1 does not really address our conjectures, for two reasons. (1) Policies that balance computational loads well need not schedule parallel computations efficiently: pathological situations could arise wherein PEs do equally much work, but with little concurrency. (2) Steady-state balanced loads do not ensure good speedup, because once nodes start failing to spawn, the load disparity can increase arbitrarily (since more than half the nodes of a BT are leaves.)

The following result from [2] does address Conjecture 1, by showing that KOSO achieves asymptotically optimal parallel speedup, at least on a narrow class of BT-computations.

Theorem 2 [2] *Under policy* KOSO*, $\mathbf{R}_p$ executes each evolving BT that evolves into the height-n complete binary tree in time $(1+o(1))(2^n-1)/p$.*

While an analytical verification of our conjectures has eluded us, we have been able to verify them via simulation experiments—at least for classes of trees that model the structure of computations such as those enumerated in Section 1. The next section describes our experiments and their results.

4 Our Experimental Study

4.1 Experimenting with an Artificial Computational Load

In our first experiment, we abstract the computations of a large class of algorithms—such as multigrid algorithms—by generating BT's via a combinatorial model. We strive for realistic "computations" by generating BT's that are "bushy"—very likely to branch—near the root but quickly get "scrawny" at deeper levels. For inspiration, consider numerically integrating via the Trapezoid Rule. The BT-structured computations will start out bushy since most nontrivial functions are nonlinear, hence will incur several levels of subdivision; however, task-spawning will quickly become sparse because of the relative smoothness of most functions over most of their domains and of the limited resolution of floating point representations. We expect our policies to yield good parallel speedup for algorithms that generate such trees.

In this experiment, input trees were generated according to a branching parameter $\alpha \in (0,\ 1)$. Each level-l tree-node spawns with probability α^l and halts with probability $(1-\alpha^l)$. Values of α close to 1 yielded trees with the desired shapes.

Our simulator maintains a collection of queues representing the PEs of $\mathbf{R}_p$. Each task is represented within its queue by an integer, its level in the evolving BT. At each iteration, the simulator executes the task at the head of each nonempty queue: it determines the probability of the task's spawning—from α and the task's level—and generates a pseudo-random number to decide whether or not the task spawns. If the task spawns, two child-tasks are created and sent to the appropriate queues according to the scheduling policy being simulated. The simulator continues this iteration until all queues are empty. Due to implementation costs—mainly storage considerations—we generated input trees on-line, rather than generating a batch of trees off-line and testing the same input for all values of the policy and the parameters α and p.

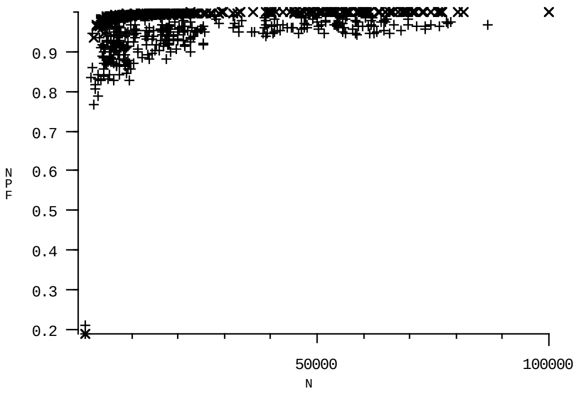
We ran the experiment with $\alpha = 0.96, 0.965, 0.97$ and $p = 8, 10, 12, 14, 16$. We chose the latter values because with fewer than 8 PEs, buses outperform rings, and with more than 16 PEs the communication latency of a ring is too great, so one would likely use a more densely connected network. For each α and p, we ran 100 trials with each policy, for a total of 3000 trials. In each trial, we measured the size N (total number of nodes) and height h (longest root-to-leaf path) of the BT, and the total execution time T.

We evaluated the observed computation times for KOSO and KOSO* in the light of the following factors which influence the optimal computation time T_p for $\mathbf{R}_p$.

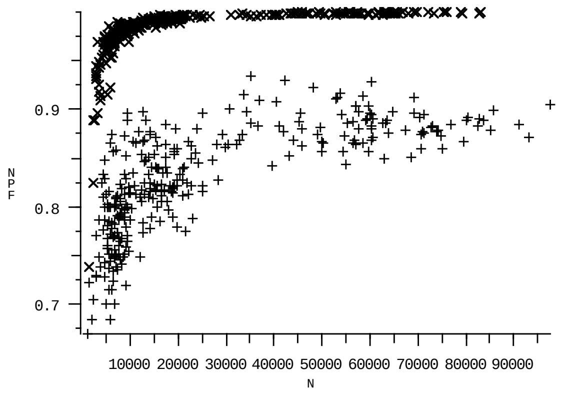
1. Easily, $T_p \geq \max(N/p,\ h)$, those quantities representing, respectively, p-fold parallel speedup and inherent sequentiality.

2. The bounds in item #1 are moderated by the inevitable "startup and cooldown" periods during which some PEs must have empty queues. (E.g., in $\mathbf{R}_p$, one cannot have all PEs busy until step p.)

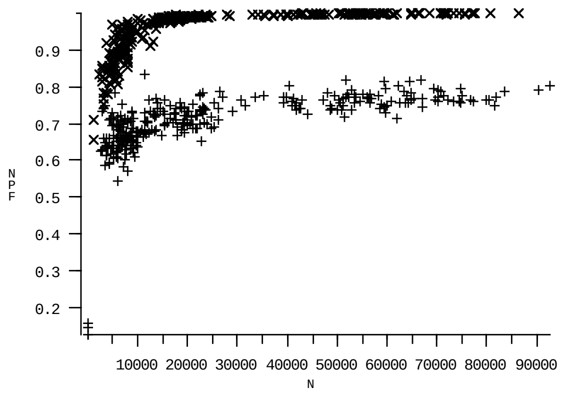
With these factors in mind, we calculated the *normalized parallelization factor*, $NPF \overset{\text{def}}{=} N/pT$, for each trial. NPF indicates what percentage of optimal speedup was attained, or equivalently, what percentage of $\mathbf{R}_p$'s capacity was utilized during the trial. Fig. 1 shows the relationship between NPF and N for each trial.



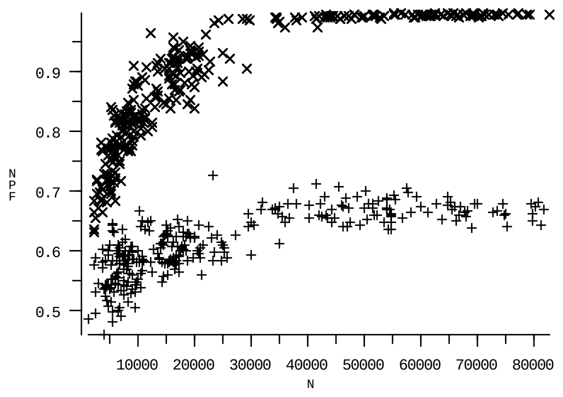
(a) $p = 8$ PEs



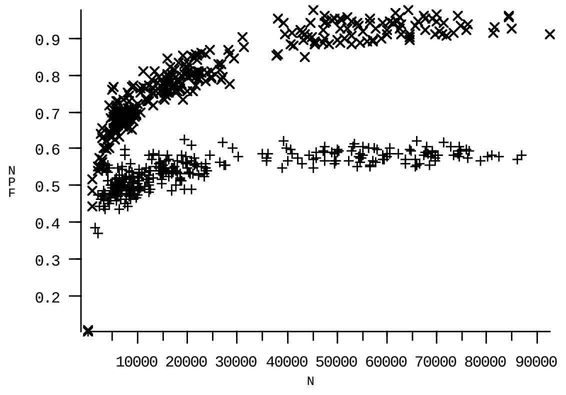
(b) $p = 10$ PEs



(c) $p = 12$ PEs



(d) $p = 14$ PEs



(e) $p = 16$ PEs

Figure 1: Percentage of optimal parallel speedup vs. tree-size ($\times$ = KOSO; $+$ = KOSO*)

Fig. 1 illustrates the importance of load-balancing, since for sufficiently large N, the NPF for KOSO* appears to climb very close to 1, while KOSO's NPF peaks notably lower. In the KOSO* trials, each of the four smaller rings appears to have a threshold tree size after which the NPF values plateau; the 16-PE ring probably has such a threshold as well, but our trees were too small to expose it. Note that NPF values are smaller for larger rings, because greater communication delay results in poorer PE utilization. Higher values of N tend to increase NPF, because the larger problem sizes offset the communication delay. Larger ring sizes also serve to increase the difference between the NPF values for each policy; the amount by which KOSO* outperforms KOSO increases with p because of the greater need for load balancing on a larger ring.

We tested the observation that KOSO* had higher NPF values than KOSO via two-sample two-tailed t-tests which indicate the probability that we are incorrectly rejecting the null hypothesis that KOSO and KOSO* perform equally well. A t-test for the entire sample, not broken down by processor size, yields a t-value of 2.89, which is significant at the $p < .001$ level. Hence, KOSO* had higher NPF values than KOSO averaging across all conditions of the experiment. Moreover, the t-values shown in Table 1 indicate that KOSO* is superior to KOSO for each individual ring size. Even after a Bonferonni adjustment for multiple testing, all results are significant at the $p < .001$ level.

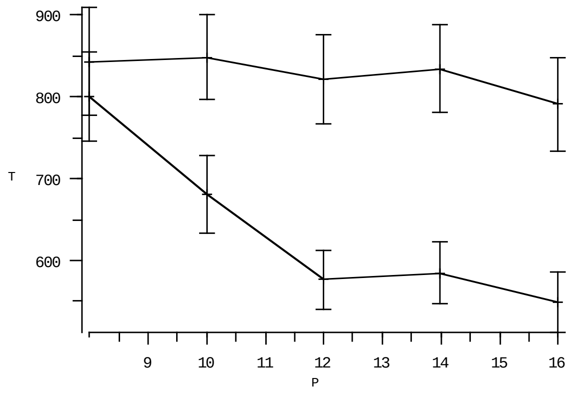
p	8	10	12	14	16
t-statistic	12.46	44.81	43.28	40.92	31.77

Table 1: t-test results for the combinatorial experiment

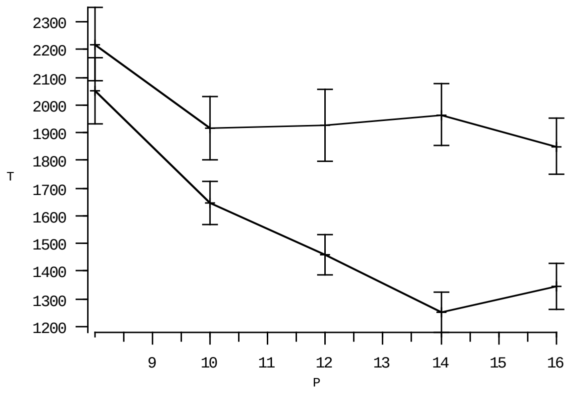
Fig. 2 shows the execution times under each policy, with trials segregated according to the value of α. (Recall that smaller values of α lead to smaller trees.) Each data point shows the mean execution time for the 100 trials in that condition, with 95% confidence bars to indicate the variability of the data. KOSO shows generally smaller execution times on larger rings, especially for larger trees. However, the overlap of confidence intervals from one value of p to the next suggests that increasing p is no guarantee of better performance; the specific shape of the input tree seems to have a large influence on execution time, especially for smaller trees. For KOSO* however, the decrease in execution times on larger rings is much more apparent, although some of the confidence bars do overlap from one value of p to the next, especially for smaller trees. Even more apparent, however, is the superiority of KOSO* over KOSO in nearly every condition—especially for larger rings. *Thus,* KOSO* *allows one to exploit parallelism better than* KOSO.

In order to investigate *why* KOSO* is more efficient than KOSO, we recorded traces of both the numbers of busy PEs and the queue sizes over the course of a trial. Figs. 3 and 4 show time-series data for six trees of approximately equal size.

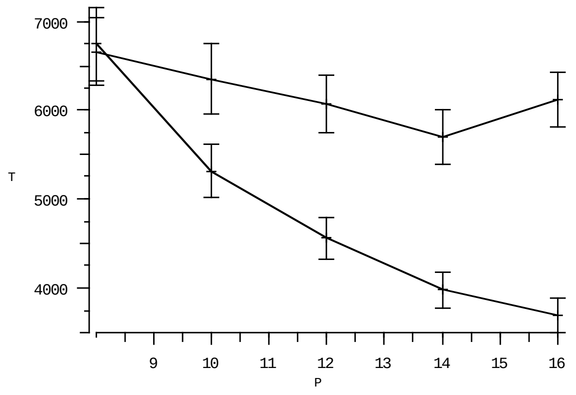
Summarizing Fig. 3. $\mathbf{R}_8$ saturates quickly under both policies, and most PEs remain busy throughout the trial; at the end of the trial, however, the number of busy PEs drops off sooner and less steeply for KOSO than for KOSO*. When $p = 12$, KOSO* is still able to keep all PEs busy for most of the trial, while under KOSO, the number of busy PEs begins to drop off less than halfway



(*a*) $\alpha = .96$

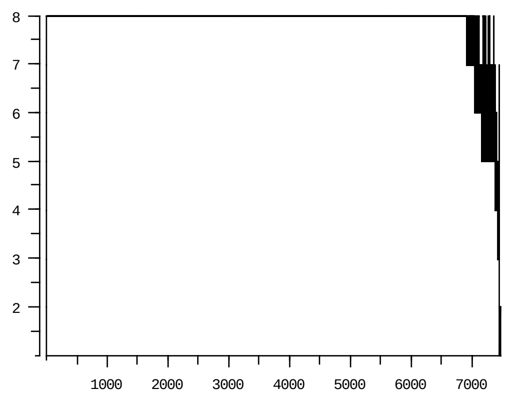


(*b*) $\alpha = .965$

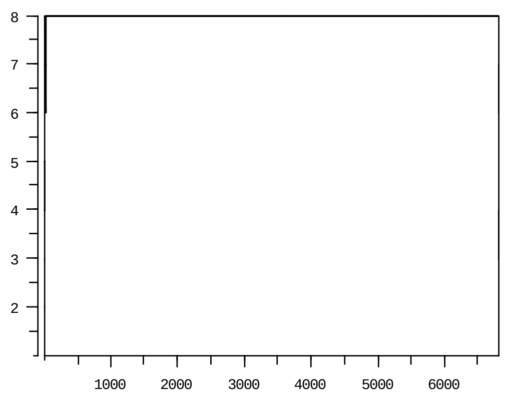


(*c*) $\alpha = .97$

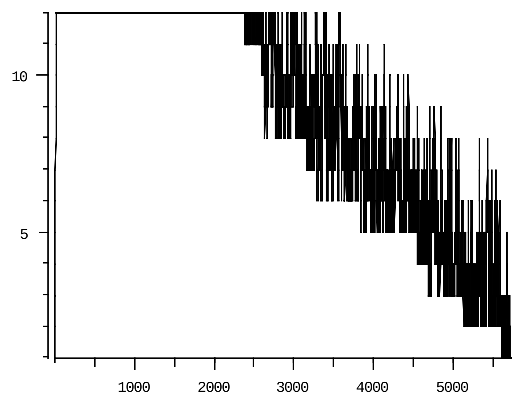
Figure 2: Execution times vs. ring size (top line KOSO; bottom line KOSO*)



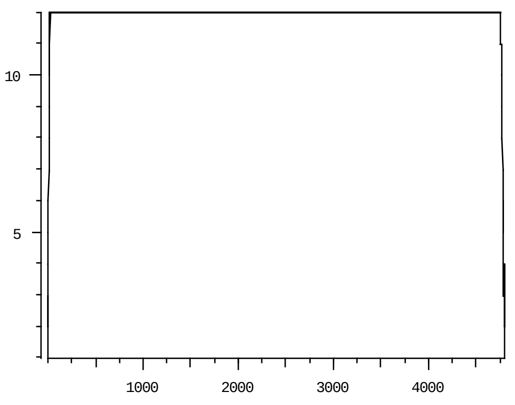
(a) KOSO with $p = 8$ PEs



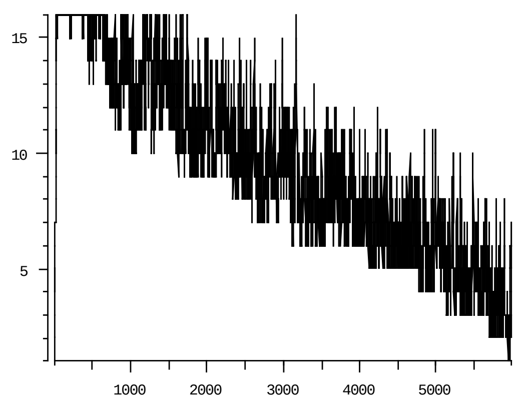
(b) KOSO* with $p = 8$ PEs



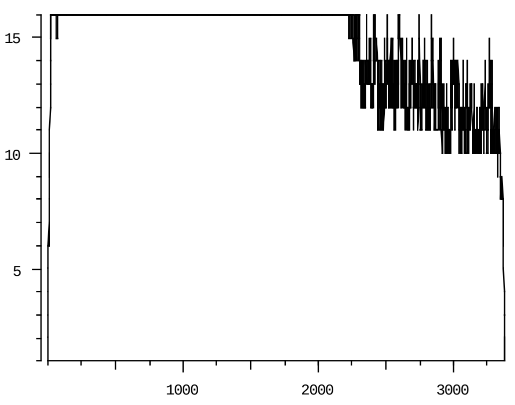
(c) KOSO with $p = 12$ PEs



(d) KOSO* with $p = 12$ PEs

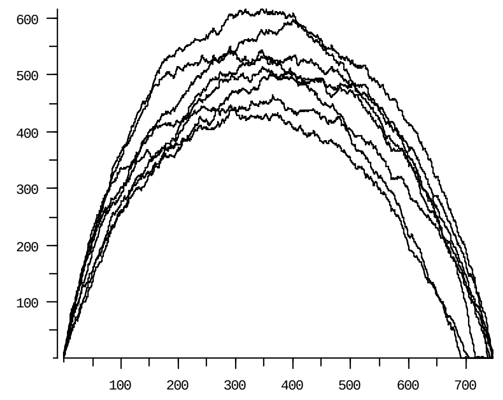


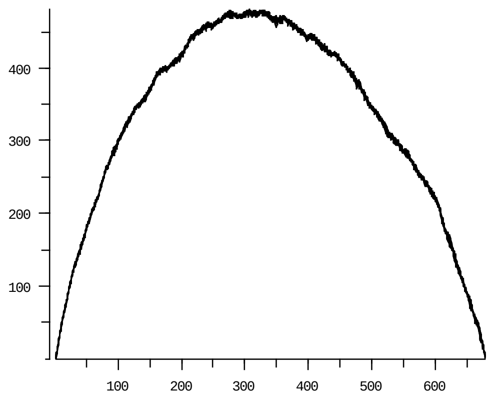
(e) KOSO with $p = 16$ PEs

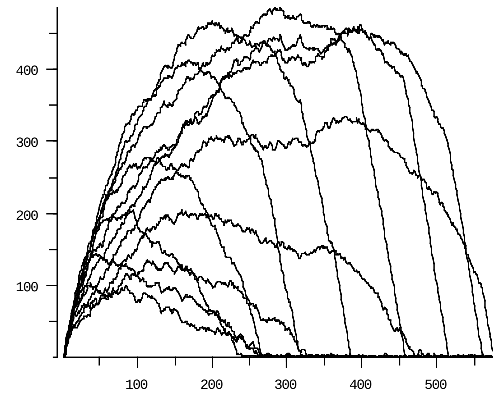


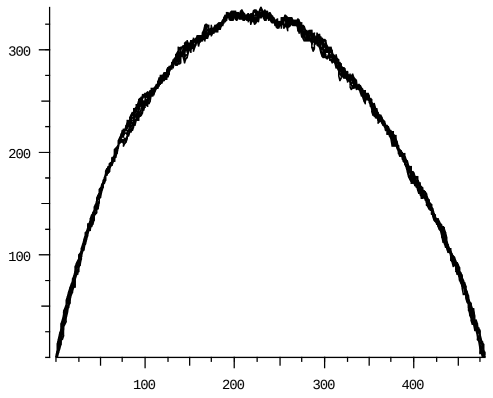
(f) KOSO* with $p = 16$ PEs

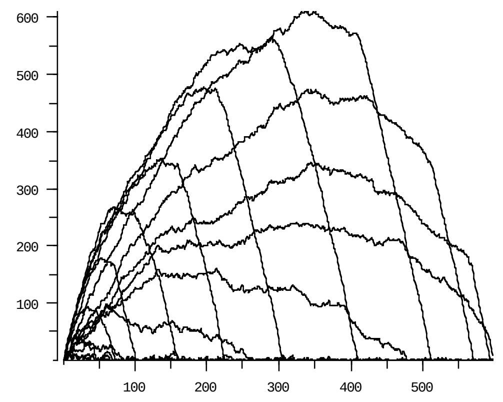
Figure 3: Numbers of busy PEs over the course of a trial


(a) KOSO with $p = 8$ PEs


(b) KOSO* with $p = 8$ PEs


(c) KOSO with $p = 12$ PEs


(d) KOSO* with $p = 12$ PEs


(e) KOSO with $p = 16$ PEs

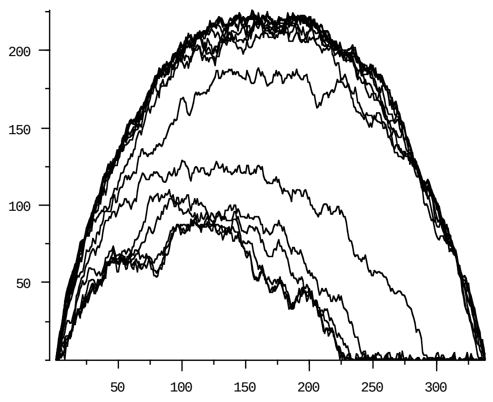

(f) KOSO* with $p = 16$ PEs

Figure 4: Queue sizes over the course of a trial

through the trial. When $p = 16$, the ring *never* saturates under KOSO; under KOSO*, the number of busy PEs drops off much more gradually than for the two smaller rings.

Summarizing Fig. 4. When $p = 8$, KOSO produces large variations in queue size, while KOSO* balances loads almost perfectly, allowing almost no variation in queue sizes throughout the trial. When $p = 12$, KOSO* performs almost as well when $p = 8$, while KOSO shows substantially degraded balance in loads. When the load is too light to saturate the ring, as when $p = 16$, neither policy balances loads well.

4.2 Experimenting with a "Real" Computational Load

Our second experiment was designed to test our policies on trees generated by instances of an actual problem, in order to demonstrate that the combinatorial model of the first experiment effectively captures the structure of trees generated by (at least some) actual algorithms. We chose the problem of integrating polynomial functions on the interval [0,1] using the Trapezoid Rule.

The Trapezoid Rule. Given a function f and a real interval [a,b]:

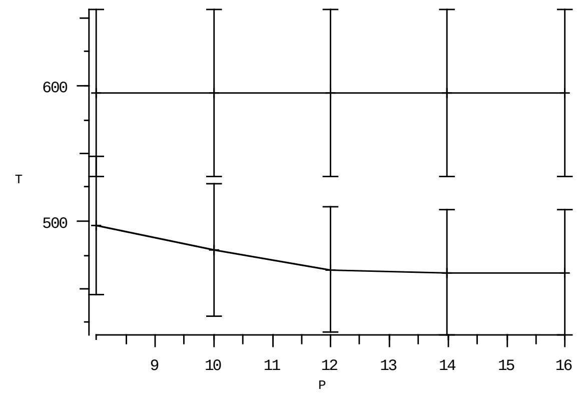
1. Calculate the area $A(a,\ b)$ of the trapezoid with corners $(a, 0)$, $(a, f(a))$, $(b,(f(b))$, $(b, 0)$.
2. If $\frac{1}{2}(b - a)$ is less than the *resolution threshold* θ_r, then return $A(a, b)$ and halt.
3. Evaluate $A(a,\ \frac{1}{2}(a + b))$ and $A(\frac{1}{2}(a + b),\ b)$.
4. If $A(a,\ \frac{1}{2}(a+b)) + A(\frac{1}{2}(a+b),\ b)$ differs from $A(a,\ b)$ by less than the *accuracy threshold* θ_a, then return $A(a, b)$ and halt; otherwise, apply the Trapezoid Rule recursively to the intervals $[a,\ \frac{1}{2}(a + b)]$ and $[\frac{1}{2}(a + b),\ b]$.

Note that θ_a plays somewhat the same role here as α did in our combinatorial experiment.

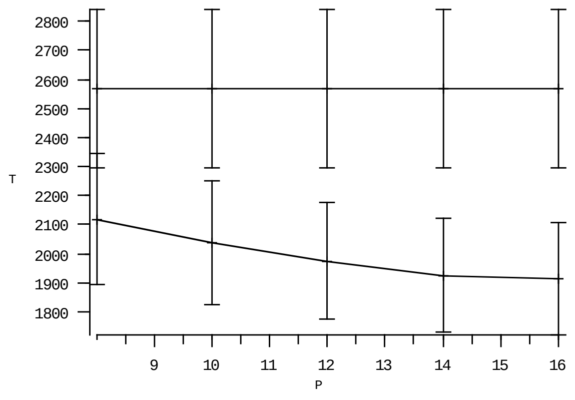
One difficulty in designing this experiment is how to generate "random" functions. (When is a function "random"?) We chose to use polynomial functions of varying degrees, because they are: easily parameterized by a small set of integers and simple to evaluate yet still provide enough "wiggles" to generate interesting trees. We generated a random polynomial π by first generating a pseudo-random degree $d \in \{0, 1, \ldots, 100\}$ and then generating d pseudo-random roots in the interval $[0, 1]$. Since this approach generates polynomials with very small coefficients, we multiplied π by a random factor $a \in \{1, 2, \ldots, 500\}$, to increase the amplitude of its wiggles. Finally, we squared $a\pi$, because the Trapezoid Rule as presented earlier expects positive functions.

Because the tree structure of a computation is determined entirely by the d roots and the amplifier a, it was computationally feasible to run multiple iterations of the experiment with the same input tree. We ran the experiment applying both KOSO and KOSO* to 100 polynomials, with ring-sizes $p = 8, 10, 12, 14, 16$ and $\theta_a = 10^{-6}, 10^{-8}, 10^{-10}$, for a total of 3000 trials. (θ_r was fixed at 10^{-10}).

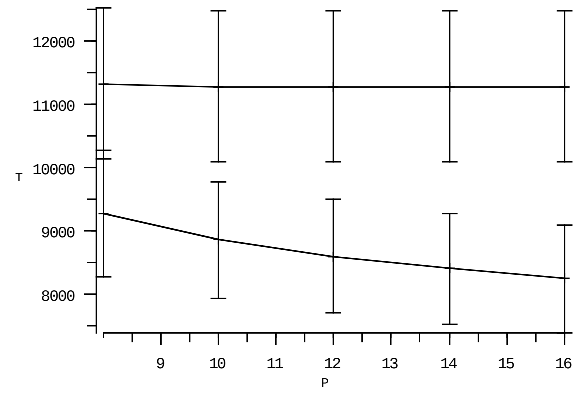
Fig. 5 shows the mean execution times for this experiment with 95% confidence intervals. These results illustrate dramatically the advantages of load-balancing. KOSO* is a clear winner over



(*a*) $\theta_a = 10^{-6}$



(*b*) $\theta_a = 10^{-8}$



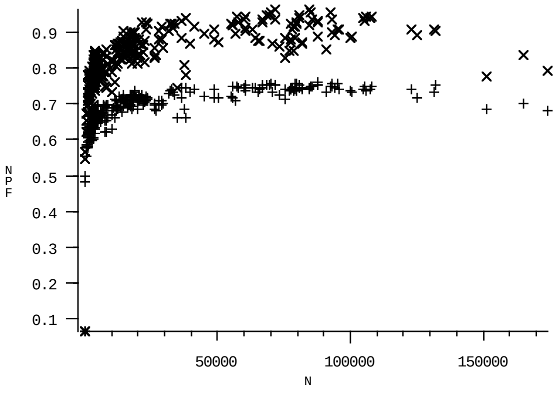
(*c*) $\theta_a = 10^{-10}$

Figure 5: Execution times vs. ring size (top line KOSO; bottom line KOSO*)

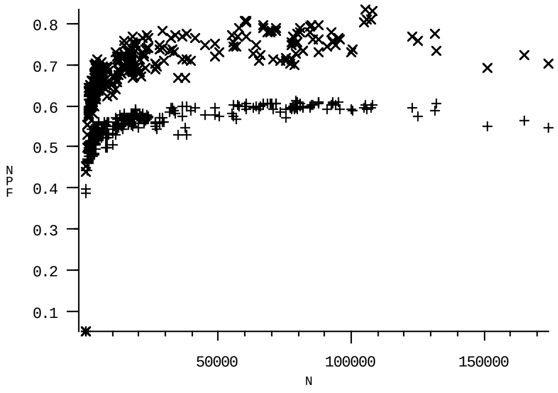
KOSO: the confidence intervals for the two policies overlap only when $p = 8$, and then only slightly. There is hardly any change in the execution times for KOSO as p increases, while KOSO*'s times show a definite downward trend. However, this trend, disappointingly, is not as steep as in the combinatorial experiment. The NPF vs. N data in Fig. 6, too, are not as encouraging as with the combinatorial experiment. Even for KOSO*, the NPF values decrease steadily as p increases and do not tend to 1 with increasing N. Once again however, the t-statistics, shown in Table 2, reveal that the NPF values are significantly higher for KOSO* than for KOSO.

p	8	10	12	14	16
t-statistic	18.82	21.97	24.16	25.10	25.21

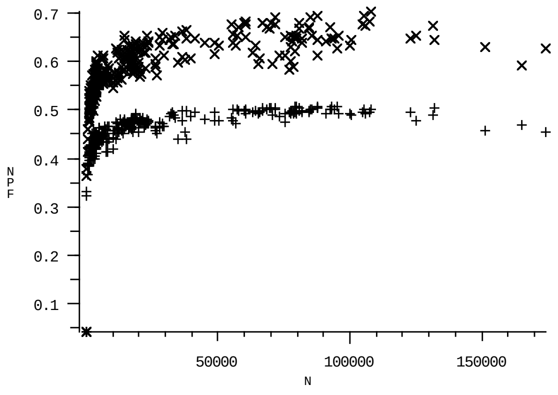
Table 2: t-test results for the second experiment



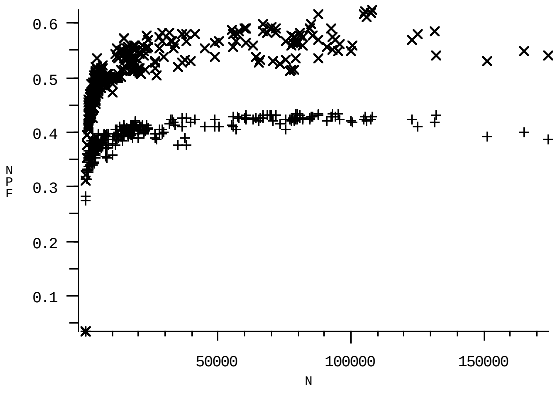
(*a*) $p = 8$ PEs



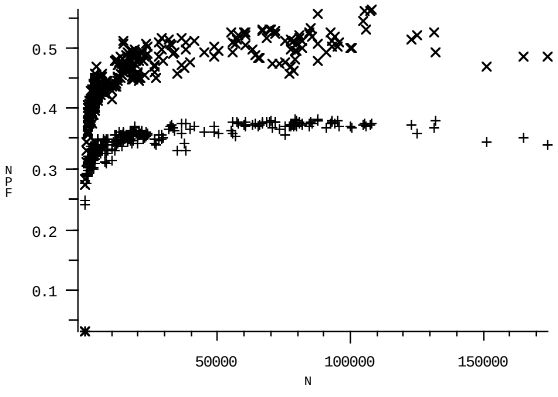
(*b*) $p = 10$ PEs



(*c*) $p = 12$ PEs



(*d*) $p = 14$ PEs



(*e*) $p = 16$ PEs

Figure 6: Percentage of optimal parallel speedup vs. tree size ($\times$ = KOSO; $+$ = KOSO*)

As in the combinatorial experiment, we attempted to explain our findings via a time-series analysis. The results, for a single polynomial under both policies and all ring sizes, with threshold $\theta_a = 10^{-10}$, are shown in Figs. 7 and 8.

The plots of busy PEs show that the ring saturates when $p = 8$, but not when $p = 12, 16$. As expected, KOSO* balances loads better than KOSO. The plots of queue size tell the same story: when $p = 8$, KOSO* keeps all queues approximately even throughout the trial, but when $p = 12, 16$, there is not enough work to saturate the ring. Under KOSO, there is little difference in performance between $\mathbf{R}_8$ and $\mathbf{R}_{16}$; in the latter, half the PEs are hardly doing any work at all.

The queue sizes from the KOSO trials do not look the same as those from the combinatorial experiment: the queues appear to grow and shrink in pairs. This may be because the random distribution of the roots of the polynomials causes most leaf nodes to occur at the same level of the tree (the deepest level). In other words, most nodes spawned until they reached a certain level, at which all of the remaining nodes halted. A histogram of the sizes and heights of the trees from the two experiments in Figs. 9 and 10 supports this explanation. To wit, while the distribution of N is comparable for the two experiments, the values of h from the polynomial experiment appear to reach an upper bound, while those from the combinatorial experiment are concentrated in one range, with outliers on either side—and, they are much larger.

4.3 Summary

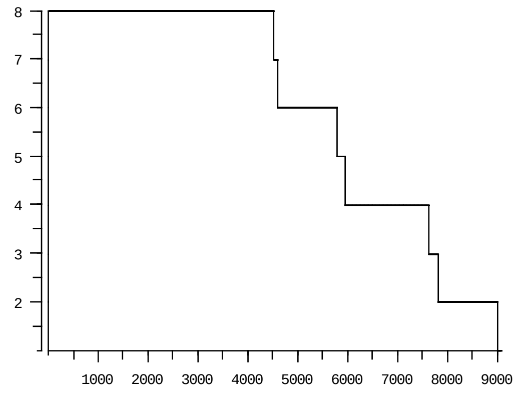
While the results from both experiments largely support our two conjectures, the polynomial experiment was disappointing, showing smaller speedup factors and less inherent parallelism. We believe this disparity is due to unanticipated different growth patterns of our combinatorial and polynomial model trees: the former trees were large and died out gradually; the latter were smaller and tended to have all leaves at the lowest level. We are not certain why this occurs—most importantly whether it is an artifact of our generation scheme or an inherent property of (random) polynomials. Further work is needed to settle this question, thereby sharpening our understanding of policies KOSO and KOSO*.

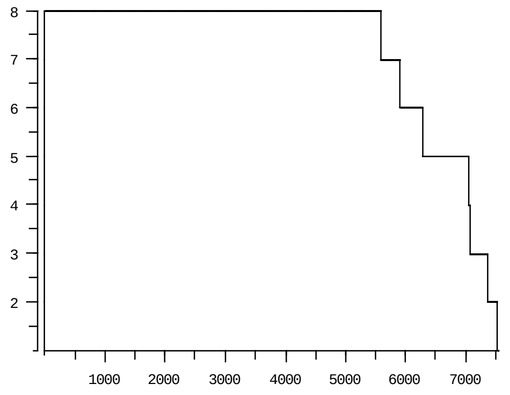
Acknowledgments. The authors wish to thank Nadia Perez of the University of Catania for programming assistance. The work of Miranda Barrows and Arnold Rosenberg was supported in part by NSF Grants CCR-92-21785 and CCR-97-10367. The work of Dawn Gregory was supported in part by a NSF Graduate Fellowship Award and by DARPA/RL Contract F30602-93-C-0100. The work of Lixin Gao was supported in part by NSF Grant NCR-9729084.

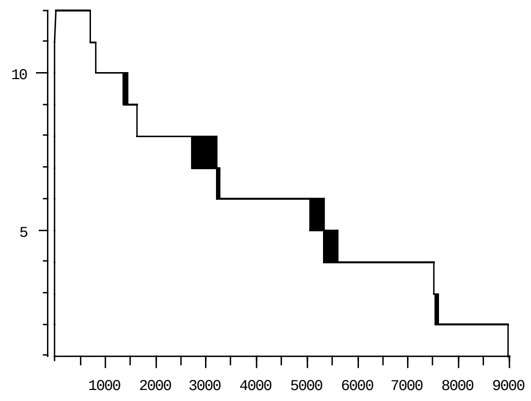
References

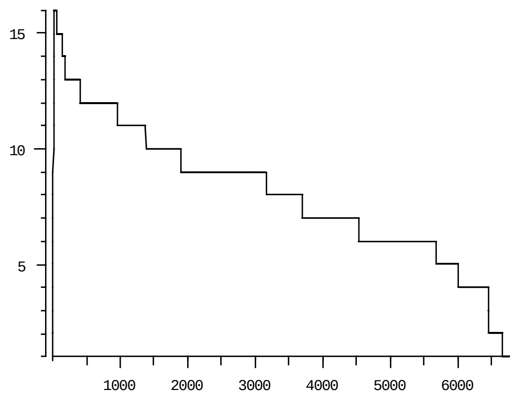
[1] P. Fizzano, D. Karger, C. Stein, J. Wein (1994): Job scheduling in rings. *6th ACM Symp. on Parallel Algorithms and Architectures*, 210–219.

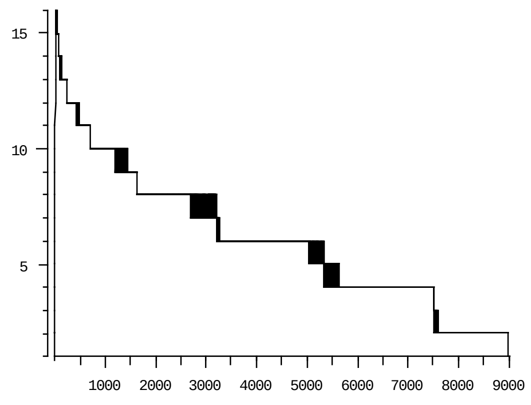
[2] L.-X. Gao and A.L. Rosenberg (1996): Toward efficient scheduling of evolving computations on rings of processors. *J. Parallel Distr. Comput. 38*, 92–100.


(*a*) KOSO with $p = 8$ PEs


(*b*) KOSO* with $p = 8$ PEs


(*c*) KOSO with $p = 12$ PEs


(*d*) KOSO* with $p = 12$ PEs


(*e*) KOSO with $p = 16$ PEs

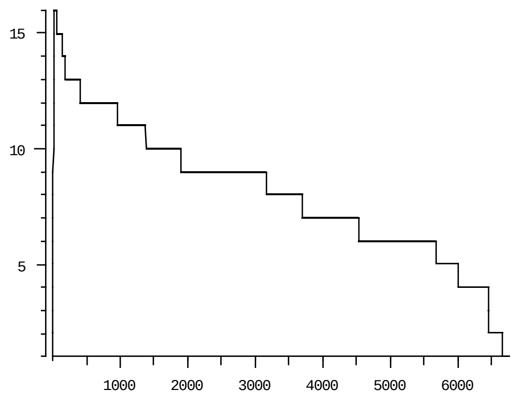
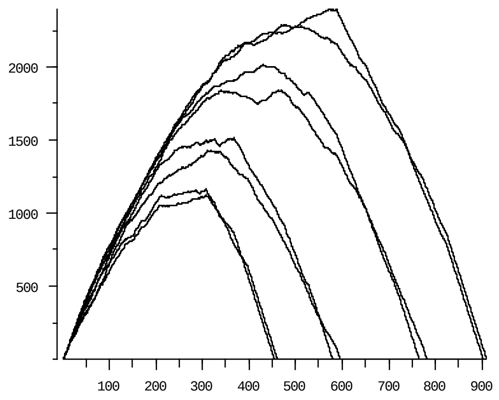
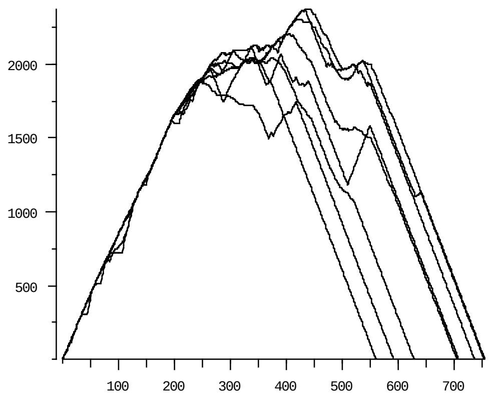

(*f*) KOSO* with $p = 16$ PEs

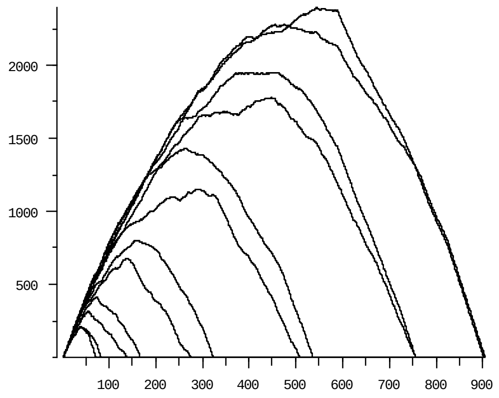
Figure 7: Numbers of busy PEs over the course of a trial



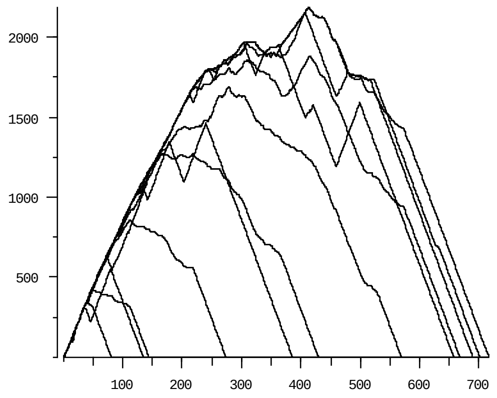
(a) KOSO with $p = 8$ PEs



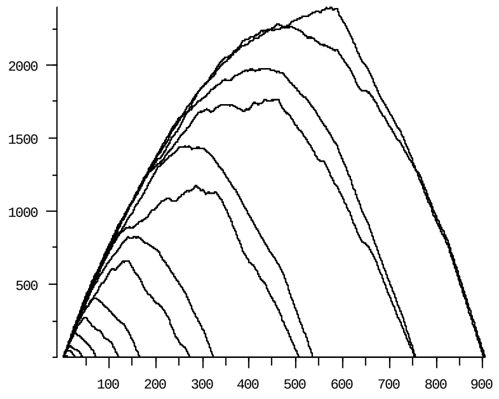
(b) KOSO* with $p = 8$ PEs



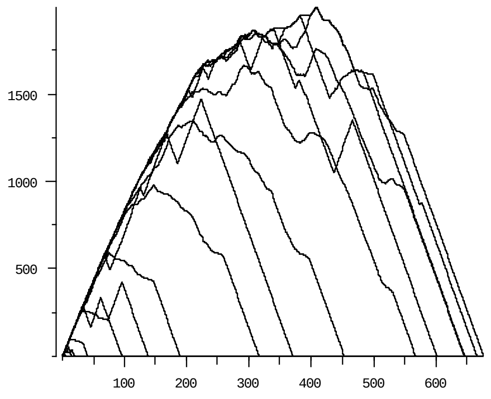
(c) KOSO with $p = 12$ PEs



(d) KOSO* with $p = 12$ PEs

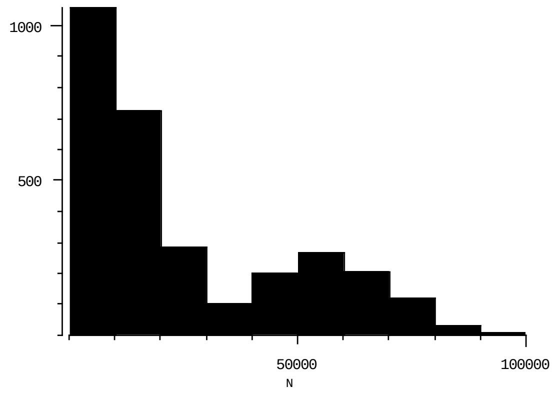


(e) KOSO with $p = 16$ PEs

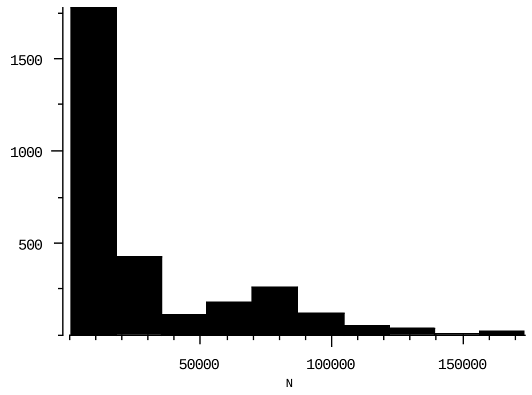


(f) KOSO* with $p = 16$ PEs

Figure 8: Queue sizes over the course of a trial

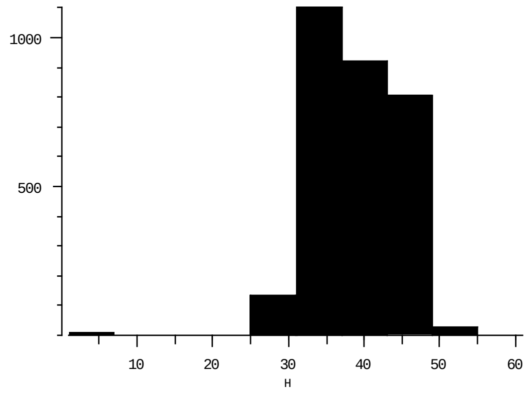


a) Combinatorial Experiment

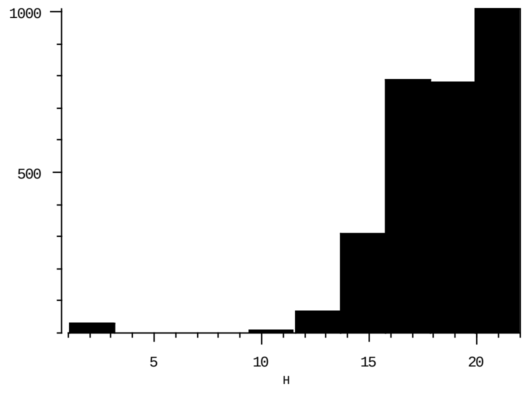


b) Polynomial Experiment

Figure 9: Distribution of N



a) Combinatorial Experiment



b) Polynomial Experiment

Figure 10: Distribution of h

[3] L.-X. Gao, A.L. Rosenberg, R.K. Sitaraman (1997): Optimal clustering of tree-sweep computations for high-latency parallel environments. See also, Optimal architecture-independent scheduling of fine-grain tree-sweep computations. *7th IEEE Symp. on Parallel and Distr. Processing*, 620–629.

[4] A. Gerasoulis and T. Yang (1992): Static scheduling of parallel programs for message passing architectures. *Parallel Processing: CONPAR 92 — VAPP V. Lecture Notes in Computer Science 634*, Springer-Verlag, Berlin, 601–612.

[5] S.L. Johnsson (1987): Communication efficient basic linear algebra computations on hypercube architectures. *J. Parallel Distr. Comput. 4*, 133–172.

[6] C. Kaklamanis and G. Persiano (1994): Branch-and-bound and backtrack search on mesh-connected arrays of processors. *Math. Syst. Th. 27*, 471–489.

[7] R.M. Karp and Y. Zhang (1993): Randomized parallel algorithms for backtrack search and branch-and-bound computation. *J. ACM 40*, 765–789.

[8] R.M. Karp and Y. Zhang (1999): On parallel evaluation of game trees. *J. ACM*, to appear.

[9] R. Lüling and B. Monien (1993): A dynamic, distributed load-balancing algorithm with provable good performance. *5th ACM Symp. on Parallel Algorithms and Architectures*, 164–172.

[10] A.G. Ranade (1994): Optimal speedup for backtrack search on a butterfly network. *Math. Syst. Th. 27*, 85–101.