

# **Maximizing Speedup Through Self-Tuning of Processor Allocation**

Thu D. Nguyen, Raj Vaswani, and John Zahorjan

Department of Computer Science and Engineering, Box 352350  
University of Washington  
Seattle, WA 98195-2350 USA

Technical Report UW-CSE-95-09-02  
October 3, 1995

# Maximizing Speedup Through Self-Tuning of Processor Allocation

Thu D. Nguyen, Raj Vaswani, and John Zahorjan

Department of Computer Science and Engineering, Box 352350  
University of Washington, Seattle, WA 98195-2350 USA

## Abstract

We address the problem of maximizing the speedup of an individual parallel job through the selection of an appropriate number of processors on which to run it. If a parallel job exhibits speedup that increases monotonically in the number of processors, the solution is clearly to make use of all available processors. However, many applications do not have this characteristic: they reach a point beyond which the use of additional processors degrades performance. For these applications, it is important to choose a processor allocation carefully.

Our approach to this problem is to provide a runtime system that adjusts the number of processors used by the application based on dynamic measurements of performance gathered during its execution. Our runtime system has a number of advantages over user specified fixed allocations, the currently most common approach to this problem: (1) we are resilient to changes in an application's speedup behavior due to the input data; and (2) we are able to change the allocation in response to speedup behavior that varies during the lifetime of the application.

This work has been done in the context of shared memory multiprocessors, such as the KSR-2 on which we ran our experiments, and for iterative parallel applications, those for which the bulk of the execution is driven by an outer sequential loop. Using both hand-tuned applications from the SPLASH benchmarks and compiler-parallelized applications from the PERFECT Club benchmarks, we show that automatic, dynamic selection of processor allocations can reliably determine allocations that match the best possible static allocation, and has the potential to find allocations that outperform any static allocation.

## 1 Introduction

We consider the problem of maximizing the speedup of an individual parallel job through selection of the number of processors on which it runs. We imagine that the operating system has made a partition of  $P$  processors available for use by the job. The application's goal is to determine the number of processors, no greater than  $P$ , that it should actually use to maximize its realized speedup on this run; unused processors are then released back to the system.

We address this goal by proposing and evaluating experimentally a runtime system that: (a) dynamically measures job efficiencies at different allocations, (b) uses these measures to calculate speedups, and (c) automatically adjusts a job's processor allocation to maximize its speedup. The technique we develop is meant to replace the currently common practice of leaving processor allocation decisions to the user of a parallel program; rather than requiring the user to specify the number of processors on which a job is to be run as part of its submission, the runtime system determines that number automatically. Besides the advantages of this approach in terms of convenience, it enjoys a number of potential advantages in terms of performance:

- *It may not be possible a priori to determine the best allocation.* Since application performance may vary significantly as a function of input data, both historical performance data and performance data obtained from test runs may be unreliable.

- *No static allocation may be optimal during the full execution of the job.* An application may exhibit different speedup characteristics as its execution evolves over time. For such applications, dynamic adjustments of processor allocations can lead to better performance.

Our runtime system employs a simple search procedure on application speedup curves as the basis of its actions. The search technique is an adaptation of *the method of golden sections* [11], which is an efficient technique to find the maximum of a unimodal function<sup>1</sup> within a finite interval. The primary modifications to this technique are to account for the fact that speedup functions, of course, are not necessarily unimodal. However, measurements of a large variety of parallel benchmarks, including both finely tuned hand-coded programs and compiler-parallelized sequential programs, show that, except for very local variations, speedup is typically unimodal over fairly substantial ranges of processors [13]. Therefore, we rely on the golden sections technique to find the optimum allocation over a local subinterval, and on our heuristic extensions to locate the local subinterval in which the global maximum lies.

Because one of our goals is to *not* rely on *a priori* information, we instead rely on runtime measurements of application performance at different allocations. Thus, for our search procedure to be successful, it is essential that comparisons of the speedup measurements made at one processor allocation with those taken at another be meaningful. Since instantaneous speedup measures reflect the characteristics of only a small section of the full application code, performing such comparisons can be problematic. This difficulty could be resolved by measuring speedup over long intervals of time (although it would be difficult to determine what constituted a sufficiently long time for an arbitrary application). However, this would have the disadvantage that each step of the search would take a long time, thus limiting its effectiveness in reducing job run time.

Instead of using this generic approach, we exploit a characteristic shared by a large variety of scientific parallel applications. In particular, we consider only *iterative* parallel applications<sup>2</sup>. An iterative application is one in which the majority of the execution is driven by a sequential loop (whose body may be entirely general, involving the execution of many parallel phases, subroutine calls, etc.). For such applications, we estimate speedups for individual iteration executions. Thus, the execution of a single iteration provides a basis by which to reasonably compare the performance of the application as processor allocations are varied. Further, empirical evidence shows that successive iterations tend to behave similarly, so that the measurements taken for a particular iteration are good predictors of future behavior.

We call the process by which an application determines for itself the best number of processors to use *self-tuning*. In what follows, we develop four approaches of increasing sophistication to self-tuning. In the first, we simply use our search procedure to identify a good processor allocation at the beginning of execution, and then use that allocation for the remaining lifetime of the execution. In the second and third, we perform the search procedure repeatedly, either at fixed intervals or in response to a dramatic change in measured efficiency (indicating that a phase change has occurred). Finally, we show how processor allocation decisions could be made for each parallel phase occurring within a single iteration.

As will become clear below, our approach requires that the application be able to adjust to changes in its processor allocation at runtime. While in the abstract this is possible in both shared memory and message passing systems (as long as the frequency of changes is not too high), in practice most existing applications have been written using a simple, static form of data and work partitioning, since they had no reason to anticipate dynamic adjustment of the number of allocated processors. As it was easier to modify shared memory applications to include the kind of dynamic scheduling required, our work has been done in that domain, and more specifically on a fifty processor KSR-2.

The remainder of the paper is organized as follows. Section 2 discusses the applications that we use and documents our experimental platform. Section 3 discusses the runtime measures used by our search procedure.

---

<sup>1</sup> For our purposes, a function  $f(x)$  is unimodal over an interval  $[a, b]$  if there is some  $x^* \in [a, b]$  such that  $f(x)$  is monotone non-decreasing in  $[a, x^*]$  and monotone non-increasing in  $[x^*, b]$

<sup>2</sup> This is not as severe a constraint as it may seem, as a large number of scientific applications are iterative. In [13], we found that five of the ten SPLASH applications and all seven of the Perfect Club applications we could compile are iterative. Furthermore, we believe that, with some modifications, our algorithm could also be applicable to non-iterative applications that use general runtime work organization paradigms, such as work heap [1, 12, 19].

Section 4 defines the basic self-tuning schemes we employ, and Section 5 examines their performance experimentally. Section 6 extends the basic schemes to allow more fine grained scheduling decisions, and examines the implications to performance. Section 7 discusses related work. Section 8 concludes the paper.

## 2 The Experimental Environment

In this section, we document our hardware and software infrastructure, and describe the set of applications used in our work.

### 2.1 Hardware Platform and System Software

All measurements were done on a Kendall Square Research KSR-2 COMA shared memory multiprocessor. Our machine consists of fifty 40MHz dual-issue proprietary processors, partitioned into two clusters of 26 and 24, respectively. Each processor is connected to a 256-KByte data cache, 256-KByte instruction cache, and 32-MByte attraction memory. Processors in each cluster, and the clusters themselves, are connected by separate 1-GByte/second slotted ring networks. The attraction memories cooperate to implement a sequentially consistent, globally-shared address space. A processor stalls when attempting to access a memory location that either is not present in its caches, or is present but not in the required state (e.g., exclusive ownership is required for a write operation).

Each node in the KSR-2 contains a hardware monitoring unit, called the event monitor, that tracks information such as cache misses and processor stall time. This information is made available to the system and user jobs through a set of read-only registers.

The KSR-2 runs a variant of the OSF/1 UNIX operating system. We use the KSR KAP preprocessor to parallelize sequential applications, and the KSR PRESTO runtime system and CThreads [4], an efficient user-level threads package, as the vehicles of parallelism. We instrument CThreads using the KSR-2's event monitor to perform runtime efficiency measurements.

### 2.2 Applications

Table 1 describes the applications used in our experiments, including the type, origin, a characterization of the run length, and a brief description of each. As mentioned in Section 1, we restrict ourselves to considering only *iterative* applications. The SPLASH applications of this class exhibit sufficiently similar behavior that we were able to simplify (and shorten) our experiments by selecting only those listed in Table 1. Although we were unable successfully to compile all of the Perfect Club codes using the KSR KAP compiler, all of those that did compile produced an iterative structure, and all such are listed in Table 1.

Our application suite, then, consists of three hand-coded parallel and seven compiler-parallelized applications. All applications are iterative and, except for Barnes, have slowdown points that are less than the maximum number of processors on our experimental system. (Basic speedups for these applications are shown in Figures 2 and 3, in Section 5, where we examine the performance of our approach.) In our experiments, Barnes and ARC2D were run for only fifty iterations because otherwise runs would have taken too long to complete. The Perfect Club applications and USAero were run on their default data sets.

To avoid loss of efficiency due to dynamic adjustment of processor allocations, we modified all hand-coded applications to dynamically match the number of threads to the number of processors at the beginning of each iteration [18]. For compiler-parallelized applications, we were able to insert this change into the KSR PRESTO runtime system, and so required no changes to the applications nor the compiler.

We also rely on the application to call appropriate runtime routines at the beginning and end of each iteration. For this study, we have inserted these calls into each application in our application suite by hand. In many instances, however, we believe that it would be possible for compilers to detect iterative behavior in both

Name	Type	Source	#Iterations	Runtime	Description
Grav	hc	Academic	200	343	N-body simulation.
Barnes	hc	SPLASH2	50	5500	Barnes-Hut N-body simulation.
MP3D	hc	SPLASH	200	276	Simulation of rarefied hypersonic flow.
ADM	cps	Perfect Club	720	432	Hydrodynamic simulation with mesoscale hydrodynamic model.
ARC2D	cps	Perfect Club	50	8583	Analysis of fluid flow problems using Euler equations.
DYFESM	cps	Perfect Club	1000	192	Analysis of symmetric anisotropic structures.
FLO52	cps	Perfect Club	150	378	Analysis of transonic inciscid flow past an airfoil using unsteady Euler equations.
QCD	cps	Perfect Club	2000	172	Simulation of gauge theory using a Monte Carlo-based algorithm.
TRACK	cps	Perfect Club	100	1404	Tracking of moving targets based on sensor inputs.
USAero	cps	Industry	1620	3350	CFD computation.

Table 1: *Benchmark Applications* (*hc* = *hand-coded*, *cps* = *compiler-parallelized sequential*). *#Iterations* is the number of iterations for which the program executes; *Runtime* is the execution time in seconds on a single processor.

hand-coded and compiler-parallelized applications and automatically insert such calls. Jeremiassen [6] has shown that it is possible to automatically detect phase behavior for many hand-coded applications. For our application suite, only FLO52 had a non-obvious iterative structure: its useful iterative loop is hidden in an outer loop that drives the program to solve multiple data sets. In cases like these, compilers may have to rely on programmer hints to identify the central iterative structure.

### 3 Runtime Measurements

A number of different runtime metrics can be appropriate for self-tuning, including execution time and efficiency. In this study, we have chosen to use efficiency because it is directly related to speedup (which itself is not directly measurable). For example, changes in efficiency from one iteration to another signify corresponding changes in speedup, whereas differences in the time to execute two iterations do not necessarily mean that speedup has changed, since the base execution times of the iterations may be different. To calculate efficiency, we directly measure overheads that cause loss of efficiency, and subtract these from 1.0.

It is well known that loss of efficiency in shared memory systems arises from:

- *Parallelization overhead.* Parallel programs may incur computational overheads that are not present in sequential programs such as per-processor initialization, work partitioning, synchronization, etc.
- *System overhead.* Parallel programs may incur additional system overhead when running on multiple processors because of per-processor system events, such as page faults, clock interrupts, etc.
- *Idleness.* Parallel programs may periodically allow assigned processors to idle because of load imbalance, synchronization constraints, etc.
- *Communication.* Parallel programs are constructed of threads of execution that may need to communicate with each other. If data required by an executing thread is not immediately available, it must be acquired. On shared memory systems, such as our KSR-2, such communication is required, for example, when data does not currently reside in the local cache, or is not in an appropriate state.

Processors in many shared memory systems stall in this situation, that is, they execute no instructions until the remote data becomes available.

We have shown previously [13] that we can accurately predict application efficiency by measuring only system overhead, idleness, and processor stall; parallelization overhead is typically small. Thus, we require only estimates of the latter three components to accurately assess efficiency.

On the KSR-2, we rely on a combination of hardware and software support to measure system overhead, idleness, and processor stall. The per-node event monitors available on the KSR-2 maintain three critical hardware counters: elapsed wallclock time, elapsed user mode execution time, and accumulated processor stall. Thus, measuring system overhead and processor stall is simply a matter of reading these three registers at the beginning and end of each iteration. Measuring idleness is only slightly more involved: we instrument all PRESTO and CThreads synchronization code to keep elapsed idle time using the wallclock hardware counter. This approach is relatively overhead free because idleness accounting is performed when the processor would otherwise not be doing any useful work. Note, however, that we implicitly assume that all application synchronization takes place through calls to the CThreads library, rather than through direct manipulation of shared variables. We did not have to modify our applications to meet this assumption; none of our hand-coded programs violated this assumption while all synchronization in compiler-parallelized applications takes place in the PRESTO runtime system by definition.

If we let  $WT(p)$  be the elapsed execution time at  $p$  processors,  $UT(p)$  be the accumulated user-mode execution time,  $IT(p)$  be the accumulated idle time, and  $PST(p)$  be the accumulated processor stall time, efficiency  $E(p)$  can then be calculated as:

$$E(p) = 1 - \frac{WT(p) - UT(p)}{WT(p)} - \frac{IT(p)}{WT(p)} - \frac{PST(p)}{WT(p)} \quad (1)$$

and speedup  $S(p)$  as:

$$S(p) = p \times E(p) \quad (2)$$

## 4 Algorithms for Self-tuning

In this section, we describe a family of self-tuning algorithms that assumes no knowledge of application structure other than the fact that the application is iterative.

### 4.1 A Basic Self-Tuning Algorithm

We start with two assumptions:

- Speedup is a function of a single variable;  $S(p) : I \rightarrow R$ , where the domain is  $[1, P]$ .
- $S(p)$  can be calculated for any  $p$ ,  $1 \leq p \leq P$ , by measuring the efficiency  $E(p)$  of any one iteration and solving equation 2.

Under these assumptions, self-tuning reduces to a single variable optimization problem, albeit with one unusual characteristic: evaluations of  $S(p)$  have different costs for different values of  $p$ . Thus, multiple evaluations of  $S(p)$  at  $p$ 's far away from the optimal value can lengthen the very execution that we are trying to minimize.

As previously mentioned, we base our current implementation of self-tuning on a simple optimization technique, the *method of golden sections* (MGS) [11], which searches for the maximum of a function over

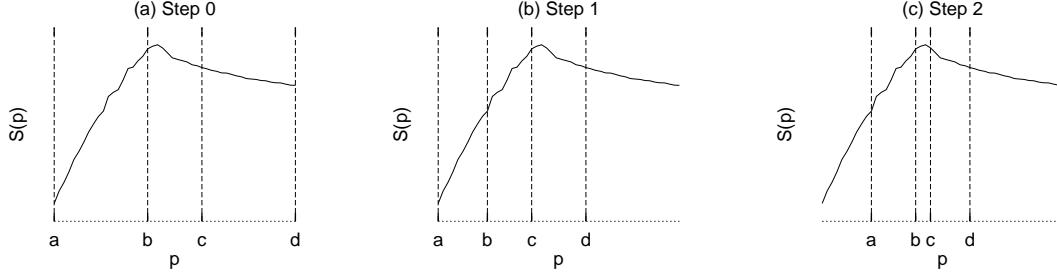


Figure 1: *Two iterations of the method of golden sections: (a) Initial state  $a_0, b_0, c_0, d_0$ ; (b)  $S(b_0) > S(c_0) \Rightarrow a_1 \leftarrow a_0, b_1 \leftarrow d_1 - 0.618(d_1 - a_1), c_1 \leftarrow b_0, d_1 \leftarrow c_0$ ; (c)  $S(b_1) < S(c_1) \Rightarrow a_2 \leftarrow b_1, b_2 \leftarrow c_1, c_2 \leftarrow a_2 + 0.618(d_2 - a_2), d_2 \leftarrow d_1$ .*

a finite interval by iteratively computing and comparing function values and narrowing the interval in which the maximum may occur; MGS is applicable to intervals over which the function is unimodal. Figure 1 illustrates two typical steps in the iteration, which involves the successive narrowing of an interval in which the maximum must fall. At each step, the function’s values at four points are known. By comparing the values of the four points, and using the fact that there is a single local maximum, the interval of interest can be narrowed by eliminating either the leftmost ( $[a, b]$ ) or rightmost ( $[c, d]$ ) subintervals. The efficiency of the technique comes from choosing the locations of the interior sample points so that they can be reused in going from one iteration to the next, thus requiring only a single new sample function value at each step.

Our adaptation of this technique begins by executing one iteration using all  $P$  available processors. This allows us to estimate  $S(P)$ . Since speedups can never be superlinear<sup>3</sup>, we know that the globally best number of processors must fall in  $[S(P), P]$ . Our search therefore starts in this interval. We compute the probe points within this interval given by the golden ratio, and execute an iteration under each. Since we know that  $S(S(P))$  is no greater than  $S(P)$ , we need not run an iteration with  $S(P)$  processors: the bound on the speedup at that number of processors is sufficient to carry out the golden sections search.

From this point, our search is identical to golden sections, with the exception that we have to handle function evaluations that indicate that the speedup function over the interval of current interest is not unimodal. When this occurs, we greedily choose the largest subinterval of the current interval for which the estimated speedups are conformal with a unimodal function, and which contains the largest speedup found so far. While this procedure is not guaranteed to choose the correct interval, it has worked well in practice.

This search procedure converges quickly. At the start, three evaluations are required to compute the four required values; the procedure terminates when these four values are adjacent. The maximum number of evaluations required is therefore  $3 + \lceil \log_{GR} \frac{4}{P} \rceil$ , where  $GR$  is the golden ratio ( $\approx 0.618$ ). For reference, at most 9 evaluations are required at  $P = 50$ , and at most 10 evaluations at  $P = 100$ .

## 4.2 Refining the Basic Self-Tuning Approach

The self-tuning algorithm just described makes three assumptions, the validity of which affect its ability to find good allocations:

The self-tuning algorithm just described makes three assumptions, the validity of which affect its ability to find good allocations:

1. *For non-unimodal speedup functions, our heuristic-based extended MGS search procedure will correctly locate the global maximum.* Especially at the beginning of the search, probe points can be widely separated. This can cause the search to exclude the interval in which the global maximum occurs.

<sup>3</sup>This is a direct consequence of our method of calculating efficiency, which is to subtract measured inefficiencies from 1.0.

2. *Speedup is not a function of time.* The basic self-tuning procedure assumes that the speedup values it sees at the beginning of a job’s execution are good representations of the job’s behavior for the indefinite future. In practice, this is not always true, possibly causing the allocation found by self-tuning at the beginning of a job’s execution to be inappropriate later in the execution.
3. *Speedup of successive iterations are directly comparable.* Even if speedup does not change substantially over time, speedup is likely to vary to some degree from iteration to iteration. Such variations, which we call *jitter*, can cause basic self-tuning to converge to a sub-optimal number of processors.

We found that the first assumption was never violated in our experimental environment. In theory, though, self-tuning could be extended in several ways to deal with erroneous exclusion of the correct interval. Possible extensions include randomly evaluating  $S(p)$  outside the current search interval, employing a covering technique [17] by evaluating  $S(p)$  at regular intervals within  $[S(P), P]$ , etc. We did not implement any of these extensions because our basic algorithm did not encounter any difficulties of this sort in our experimental environment. Therefore, it would have been impossible to evaluate the cost-effectiveness of the proposed modifications.

On the other hand, we found that applications can indeed violate our latter two assumptions. Thus, we extend self-tuning to possibly rerun the search procedure a number of times during a job’s execution lifetime. Such re-evaluation of the search would elicit adaptation to slowly evolving speedup behavior, and would allow us to escape from unfortunate search results obtained on any one attempt.

Our first extended algorithm, which we call *change-driven self-tuning*, continuously monitors job efficiency and re-initiates our search procedure whenever it notices a significant change (according to a predefined threshold value) in efficiency. This approach is intended to minimize the overhead of re-executions of the search procedure by triggering them only when it appears likely that there has been a genuine change in the application’s behavior.

Unfortunately, if job efficiency changes in the middle of a self-tuning search but stabilizes before the search completes, change-driven self-tuning can still converge to a stable but incorrect allocation. Thus, we consider a second extended algorithm, which we call *time-driven self-tuning*, that includes change-driven self-tuning but will also rerun the search procedure periodically, even when there has been no change in job efficiency. This approach is intended to allow us to limit the damage caused by any single search that converges to a grossly incorrect allocation.

Both these approaches, as well as the basic self-tuning procedure, are examined in the performance results presented in the next section.

## 5 Performance

In this section, we evaluate the performance of three variants of our self-tuning algorithm. The three implementations are:

- **Basic:** Run self-tuning once at the beginning of the execution.
- **Change-driven:** The basic algorithm as above, but reinitiate self-tuning whenever a significant change in speedup is detected.
- **Time-driven:** The change-driven algorithm as above, but in addition, restart self-tuning every 100 iterations. If a restarted search settles on the same number of processors as before, double the wait until the next restart.

Figure 2 depicts the speedup curves of hand-coded applications when run under each of the four possible processor allocation disciplines: no tuning (use all available processors), basic self-tuning, change-driven self-tuning, and time-driven self-tuning. Figure 3 depicts similar curves for compiler-parallelized applications.

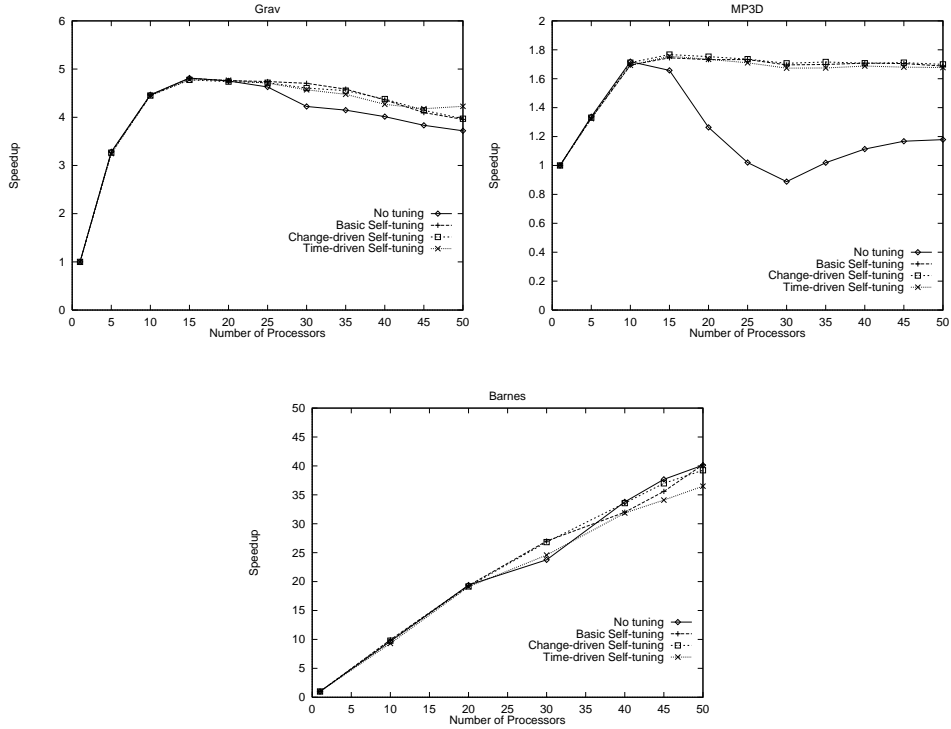


Figure 2: Speedup of hand-coded applications.

These figures provide the context for the following set of observations regarding the behavior of runtime self-tuning processor allocation policies.

*Self-tuning imposes very little overhead.* This can be seen by examining the curves for applications with good speedup: Grav and Barnes in Figure 2, and ARC2D in Figure 3. These applications continue to exhibit good speedup even when using (all forms of) self-tuning, with the curves for the self-tuning policies falling close to or on top of the original (no tuning) speedup curves.

*Basic self-tuning often significantly improves performance over no tuning.* While self-tuning — basic or otherwise — can do little to aid the three aforementioned well-performing applications, basic self-tuning significantly improves speedup for all of the remaining applications (with the exception of USAero, discussed next). This improvement can be seen by comparing the no tuning and basic self-tuning curves for MP3D in Figure 2, and for all applications except USAero in Figure 3.

*Change-driven self-tuning can significantly improve performance over basic self-tuning.* As previously explained, basic self-tuning can result in poor performance if either (a) a job’s speedup changes significantly over time, causing the allocation chosen at the beginning of its execution to be a poor choice later in its execution, or (b) jitter in measured efficiencies of consecutive iterations causes self-tuning to choose incorrectly. For the applications available to us, jitter appears to be small enough such that this basic instability in the search procedure does not cause problems. On the other hand, long term changes in speedup can lead to noticeable performance degradation. As can be seen in Figure 3, the performance of USAero, an application whose speedup does change with time [13], is indeed enhanced by change-driven self-tuning.

*Time-driven self-tuning is not useful for the applications we studied.* Time-driven self-tuning is meant to address the situation in which a job’s speedup changes in the middle of self-tuning but stabilizes before self-tuning completes, possibly trapping change-driven self-tuning into a poor allocation choice. For the applications available to us, this situation never arises. Given this, time-driven self-tuning’s additional overhead of periodically rerunning self-tuning can lead to poorer application performance as compared to change-driven self-tuning (e.g. ADM and DYFESM).

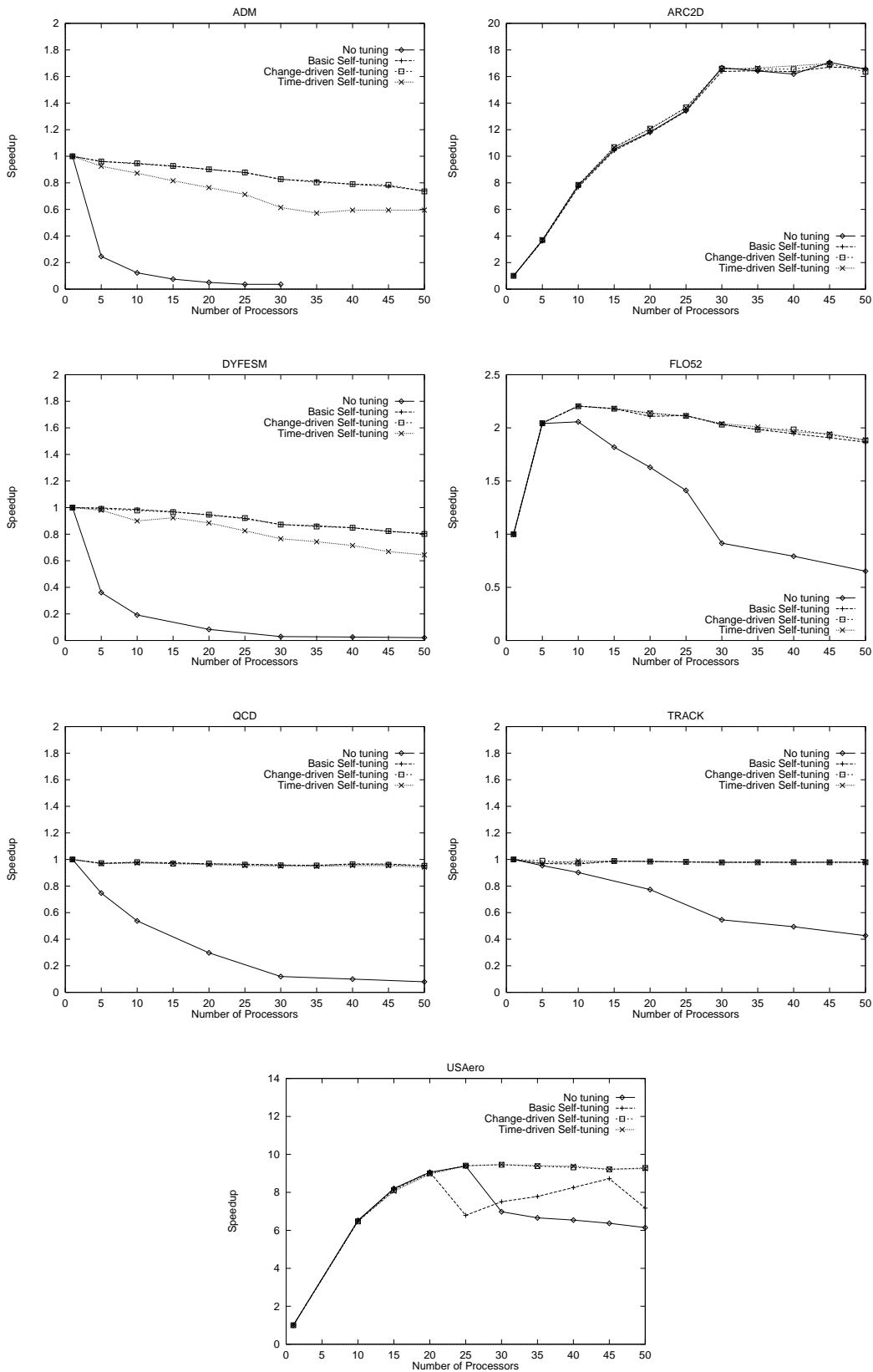


Figure 3: Speedup of compiler-parallelized applications.

*The performance benefit of self-tuning can be limited by the cost of probes.* As stated previously, self-tuning imposes little overhead on applications that achieve good speedup. For such applications,  $S(P)$  is reasonably close to  $P$ , implying that the interval searched by the self-tuning procedure ( $[S(P), P]$ ) is fairly small. Furthermore, the cost of probing (evaluating speedup) at various points within the interval is quite small, because speedups at those points are quite good. Applications that achieve poor speedup, on the other hand, suffer because  $S(P)$  is not close to  $P$ , implying that  $[S(P), P]$  is large. Furthermore, the cost of probing points within the interval is large because speedups at those points are poor. For this reason, although applications with poor speedup (most of those in Figure 3) are helped in the large by self-tuning, speedup does degrade somewhat due to the cost of the self-tuning procedure. That this degradation is due mainly to the cost of searching can be seen by comparing the curves of QCD and TRACK to those of ADM and DYFESM in Figure 3. QCD runs many more iterations than do the other applications (see Table 1), so the cost of the search procedure consumes a smaller fraction of execution time. In the case of TRACK, the slope of its slowdown is not as steep as that of the other applications. Therefore, probing at the slow points has a less detrimental impact on TRACK’s runtime than it does on that of the other applications.

We have seen that a fairly simple procedure can be used to achieve performance that is in all cases very near that achievable by statically choosing, with oracular knowledge, the perfect number of processors on which the application should be run. The schemes described so far are, however, fairly coarse-grained in that they settle for long periods on a single number of processors. In the next section, we investigate whether performance can be further improved by using more information about program structure to choose processor allocations at finer granularity.

## 6 Multi-phase Self-tuning

The self-tuning methods described above choose a single number of processors to be used during each iteration. The iterations of some applications, however, are composed of multiple parallel phases, each of which may have a distinct ideal number of processors to use when executing. In this section we explore the possible advantages of such *multi-phase self-tuning*.

In the discussion that follows, a phase corresponds to a specific piece of code, e.g., a parallel loop in a compiler-parallelized code, or a subroutine in a hand-parallelized program. Each phase may be executed zero or more times per iteration, depending on the outcome of conditional expressions and sequential loops contained within a single iteration. We assume that on each entry to and exit from a phase, the runtime system is provided with the unique ID of the phase.

Let there be  $N$  phases in an iteration. The phase allocation problem is to find a vector  $(p_1, p_2, \dots, p_N)$  that maximizes performance, where phase  $n$  executes on  $p_n$  processors. Clearly, the techniques presented in the previous section are special cases where  $p_i = p_j$  for  $1 \leq i, j \leq N$ .

There is an obvious approach to the solution of the phase allocation problem that simply extrapolates from the basic self-tuning technique: we merely apply self-tuning to each phase independently (but in parallel), using exactly the same search technique as before. We call this approach *independent multi-phase self-tuning*. The advantage of this technique is its simplicity, and the assurance that the procedure converges quickly. The disadvantage is that the final allocation vector may give performance that is far from optimal because it is based on the premise that the performance of each phase depends only on its own allocation and not on the allocations for any other phases. In practice, this is not the case because the allocation in phase  $n$  dictates a problem partitioning, and that partitioning affects the amount of communication required in phase  $n + 1$ , even for fixed  $p_{n+1}$ .

To account for the relationship of the performance in one phase to the allocation decisions made for other phases, a more complicated search procedure is required. In particular, we propose a randomized search technique, called *inter-dependent multi-phase self-tuning*, based on simulated annealing [7].

## 6.1 Inter-dependent Multi-Phase Self-Tuning

The basic operation of this technique is to choose an initial candidate allocation vector, to evaluate the performance at this vector, to select a modified vector to consider, to evaluate performance at the modified vector, and then to accept or reject the modified vector as the new candidate vector based on the outcome.

In somewhat more detail:

- *Choosing an initial candidate allocation vector.* In practice, it is important to avoid candidate vectors that yield poor performance, as these can greatly lengthen the execution time we are attempting to minimize. Our approach to finding a good initial candidate is to begin by performing the basic self-tuning search described earlier. This yields an allocation vector (obtained by using the single fixed allocation found by that procedure for all phases) that should be nearly optimal among such fixed vectors. We run the application for one iteration using this fixed allocation vector, and measure the efficiencies of each phase during that execution, using these to estimate phase speedups. We use the ratio of the speedup of each phase to the overall iteration speedup as a multiplier for that phase's initial processor allocation; that is, the initial allocation for phase  $n$ ,  $p_n^1$ , is the fixed number of processors found by the initial search procedure times that ratio. The initial candidate vector for our randomized search is then  $(p_1^1, p_2^1, \dots, p_N^1)$ .
- *Selecting a new candidate vector.* After running for one iteration at the currently chosen candidate vector, a new candidate vector must be selected. This is done by first randomly selecting some fraction of the  $N$  elements in the original vector. (In our experiments, we selected 40% of the  $N$  elements.) These  $k$  elements are then modified by randomly choosing a value near their values in the original vector. That is, for the next iteration  $i$ ,  $p_n^i$  is selected by randomly choosing a value near  $p_n^1$ .
- *Evaluating and accepting new candidate vectors.* A new candidate vector is evaluated by running under it for one iteration, measuring the total iteration efficiency, and using this to estimate iteration speedup. If that speedup is higher than that for the previous candidate vector, the new vector becomes the current candidate; if speedup is instead lower, we randomly decide whether or not to accept the new vector as the current candidate. This randomized acceptance of seemingly unpromising candidates is a key aspect of simulated annealing approaches, as it allows them to escape from local optima in order to find a better, global optimum.
- *Terminating the search.* Presently, we terminate the search after a fixed number of iterations. Once the search has terminated, we adopt for further iterations the iteration vector that resulted in the highest iteration speedup. (Note that this is not necessarily the final candidate vector, because of the randomized acceptance procedure.)

We now compare the performance of the two multi-phase self-tuning algorithms to that given by no self-tuning and basic self-tuning.

## 6.2 Performance

Multi-phase self-tuning obviously requires that applications' gross iterations exhibit internal phase structure. While all of the compiler-parallelized applications exhibit this, of the hand-coded applications, only Grav has multiple significant phases of execution within a single iteration. To illustrate the potential advantages of the multi-phase schemes, Figure 4 compares the speedup of Grav under the multi-phase schemes to using no self-tuning and basic self-tuning; Grav's behavior using both small and large data sets is shown.

We see that the multi-phase techniques are able to achieve performance not realizable by any fixed allocation. This occurs for two reasons. First, three of Grav's five phases have performance that falls off sharply once a relatively modest number of processors is reached, while the others are able to use all available processors.

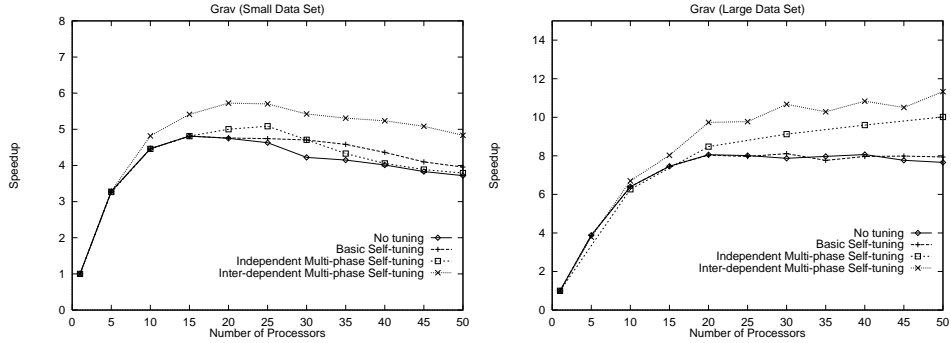


Figure 4: *Multi-phase Speedup.*

By allowing each phase to choose a distinct allocation, we are able to obtain near optimal performance in each phase, whereas any fixed allocation would have to make a number of compromises.

Second, successive phases of Grav do not exhibit a very tight data coupling. This is the result of Grav’s current implementation, which uses a very simple work-queue paradigm, such that the assignment of work to processors in each phase is relatively random. We speculate that if a more careful work assignment were performed, some of the performance gains of multi-phase self-tuning over basic self-tuning may disappear.

This speculation is borne out by preliminary results for the compiler-parallelized codes, not given here. Multi-phase self-tuning failed to outperform basic self-tuning in any of these codes. Examination of these applications, which use block scheduling of parallelized loops to processors, shows that successive parallel phases reference overlapping sets of data. This overlap, in conjunction with the block scheduling, means that there is a disadvantage to scheduling successive phases on differing number of processors, because of the communication penalty this induces.

Interestingly, we observe that inter-dependent self-tuning yields better performance than independent self-tuning. This shows that, even for an application where the data flow relationships of one phase to the next are quite weak, accounting for inter-phase dependencies can be beneficial.

Our current work includes further examination of this question for compiler-parallelized sequential programs. Measurements show that all 7 compiler-parallelized applications exhibit multiple significant phases of execution within each iteration. Unfortunately, these applications seem to have much tighter data coupling between phases. Preliminary multi-phase self-tuning results based on runtime phase identification have failed to show improved performance. We speculate that in order for multi-phase self-tuning to be effective, we need access to program structure at compile-time to identify those phase boundaries across which data coupling is relatively weak. A compiler-parallelized job’s processor allocation should then be changed only at these phase boundaries.

## 7 Related Work

Many researchers have studied the use of application characteristics by processor schedulers of multiprogrammed systems. This work differs from ours in that its goal is to determine an appropriate allocation to each of several simultaneously scheduled jobs, typically with the goal of minimizing average response time. Majumdar et. al. [9], Chiang et. al [3], Leutenegger and Vernon [8], Sevcik [15, 16], Ghosal et. al. [5], Rosti et. al. [14] and others have proposed using application characteristics such as efficiency, speedup, and average parallelism to improve the performance of static processor schedulers. More recently, Brecht [2] has proposed that application characteristics such as efficiency and execution time can profitably be used by dynamic processor schedulers as well. All of these studies, however, assume the availability of accurate historical performance data, provided to the scheduler simultaneously with job submission. Furthermore,

they focus on overall system performance, as opposed to the performance of individual applications.

McCann et. al. [10] have proposed a dynamic scheduler that uses application provided runtime idleness information to dynamically adjust processor allocations to improve processor utilization. Although the performance of individual applications can improve because each is likely to receive more processor time under this dynamic allocation policy, this occurs only in the context of a multiprogrammed system. Furthermore, lack of parallelism is the only source of inefficiency they consider.

## 8 Conclusions

We have proposed a technique to automatically regulate the number of processors used in the execution of a parallel program so as to maximize its speedup for that run. Our approach relies on the ability to measure program inefficiencies resulting from load imbalance, contention for shared critical sections, communication, etc. We have shown how such measurements can conveniently be performed by a runtime system for both hand-coded and compiler-parallelized programs.

We have demonstrated that simple search procedures, guided by the runtime measurements, can automatically select appropriate numbers of processors on which to run applications. Our approach thus relieves the user of the burden of determining the precise number of processors to use for each input data set. Additionally, because our runtime system can dynamically adjust to changes in the optimal processor allocation during the execution of a job, it has the potential to outperform any statically specified allocation.

Finally, we believe that self-tuning is especially promising for compiler-parallelized applications. For these applications, the compiler must determine which loops contain sufficient work for it to be worthwhile executing them in parallel. Due to the difficulty of making this determination statically, the compiler must either be conservative or else be subject to a high possibility of a mistake. Our system allows this decision to be delayed until runtime, and gathers enough information to make a correct decision.

## References

- [1] G. A. Alverson and D. Notkin. Program Structuring for Effective Parallel Portability. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):1041–59, Sept. 1993.
- [2] T. B. Brecht and K. Guha. Using Parallel Program Characteristics in Dynamic Multiprocessor Allocation Policies. [ftp://www.cs.yorku.ca/pub/brecht/Brecht\\_Guha.ps](ftp://www.cs.yorku.ca/pub/brecht/Brecht_Guha.ps), May 1995.
- [3] S.-H. Chiang, R. K. Mansharamani, and M. K. Vernon. Use of Application Characteristics and Limited Preemption for Run-to-Completion Parallel Processor Scheduling Policies. In *Proceedings of the ACM SIGMETRICS Conference*, pages 33–44, May 1994.
- [4] E. C. Cooper and R. P. Draves. C Threads. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie-Mellon University, June 1988.
- [5] D. Ghosal, G. Serazzi, and S. K. Tripathi. The Processor Working Set and its Use in Scheduling Multiprocessor Systems. *IEEE Transactions on Software Engineering*, 17(5):443–53, May 1991.
- [6] T. E. Jeremiassen. *Using Compile-Time Analysis and Transformation to Reduce False Sharing on Shared-Memory Multiprocessors*. PhD thesis, University of Washington, 1995.
- [7] S. Kirkpatrick, J. C. D. Gellat, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.
- [8] S. T. Leutenegger and M. K. Vernon. The Performance of Multiprogrammed Multiprocessor Scheduling Policies. In *Proceedings of the ACM SIGMETRICS Conference*, pages 226–236, May 1990.

- [9] S. Majumdar, D. L. Eager, and R. B. Bunt. Scheduling in Multiprogrammed Parallel Systems. In *Proceedings of the ACM SIGMETRICS Conference*, pages 104–113, May 1988.
- [10] C. McCann, R. Vaswani, and J. Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.
- [11] G. P. McCormick. *Nonlinear Programming*. John Wiley & Sons, Inc., 1983.
- [12] P. Moller-Nielsen and J. Staunstrup. Problem-Heap: a Paradigm for Multiprocessor Algorithms. *Parallel Computing*, 4(1):63–74, Feb. 1987.
- [13] T. D. Nguyen, R. Vaswani, and J. Zahorjan. On Scheduling Implications of Application Characteristics. Technical report, Department of Computer Science and Engineering, University of Washington, in preparation.
- [14] E. Rosti, E. Smirni, L. W. Dowdy, G. Serazzi, and B. M. Carlson. Robust Partitioning Policies of Multiprocessor Systems. *Performance Evaluation*, 19(2-3):141–65, Mar. 1994.
- [15] K. C. Sevcik. Characterizations of Parallelism in Applications and their Use in Scheduling. In *Proceedings of the ACM SIGMETRICS Conference*, pages 171–180, May 1989.
- [16] K. C. Sevcik. Application Scheduling and Processor Allocation in Multiprogrammed Parallel Processing Systems. Technical Report CSRI-282, Computer Systems Research Institute, University of Toronto, Mar. 1993.
- [17] A. Torn and A. Zilinskas. *Global Optimization*. Springer-Verlag, 1987.
- [18] A. Tucker and A. Gupta. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 159–166, Dec. 1989.
- [19] M. Vandevoorde and E. Roberts. WorkCrews: an Abstraction for Controlling Parallelism. *International Journal of Parallel Programming*, 17(4):347–66, Aug. 1988.