U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology

NISTIR 5139

Computer Systems Laboratory

Using Synthetic-Perturbation Techniques for Tuning Shared Memory Programs

Robert Snelick
Joseph Ja'Ja'
Raghu Kacker
Gordon Lyon

U. S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Gaithersburg, MD 20899

March 1993

CMRF

COMPUTER MEASUREMENT
RESEARCH FACILITY
FOR HIGH PERFORMANCE
PARALLEL COMPUTATION

Partially sponsored by the
Defense Advanced Research Projects Agency

QC
100
.U56
#5139
1993

NISTIR 5139

Using Synthetic-Perturbation Techniques for Tuning Shared Memory Programs

Robert Snelick, Div. 875
Joseph Ja'Ja', Div 875*
Raghu Kacker, Div 882
Gordon Lyon, Div 875

* University of Maryland,
College Park, MD 20742

U. S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Gaithersburg, MD 20899

Partially sponsored by the Defense Advanced Research Projects Agency

March 1993

Contents

Using Synthetic-Perturbation Techniques for Tuning Shared Memory Programs [1]

Robert Snelick
Joseph JáJá[2]
Raghu Kacker
Gordon Lyon

National Institute of Standards and Technology[3]
Gaithersburg, Maryland 20899

Abstract

The Synthetic-Perturbation Tuning (SPT) methodology is based on an empirical approach that introduces artificial delays into the MIMD program and captures the effects of such delays by using the modern branch of statistics called design of experiments. SPT provides the basis of a powerful tool for tuning MIMD programs that is portable across machines and architectures. The purpose of this paper is to explain the general approach and to extend it to address specific features that are the main source of poor performance on the shared memory programming model. These include performance degradation due to load imbalance and insufficient parallelism, overhead introduced by synchronizations and by accessing shared data structures, and compute time bottlenecks. We illustrate the practicality of SPT by demonstrating its use on two very different case studies: a large image processing benchmark and a parallel quicksort.

Key words: - design of experiments, parallel programs, performance, shared memory programming model, synthetic-perturbation, tuning

---

[1]Partially sponsored by the Defense Advanced Research Projects Agency.

[2]Also, University of Maryland, College Park, MD 20742.

[3]No recommendation or endorsement, express or otherwise, is given by the National Institute of Standards and Technology or any sponsor for any illustrative commercial items in the text.

1 Introduction

Today's multiprocessors provide unprecedented performance potential, yet all too often the actual performance obtained is far less impressive. Since their inception, a deficiency of multiprocessor computers has been the lack of adequate performance measurement and debugging tools. The inherent complexity of parallel programs makes it far more difficult to capture *true* performance measurements on multiple-instruction stream, multiple-data stream (MIMD) architectures. In the absence of MIMD performance tools, obtaining reasonable parallel program performance is no small undertaking. Our objective here is to explain, extend, and apply a technique that gives the programmer useful performance information and is portable across machines as well as architectures. The technique works equally well in both shared memory and message passing environments. This work emphasizes the SPT techniques for shared memory programs.

Many existing tools [6, 8, 9, 10, 11, 12] focus on capturing performance metrics via monitoring. Performance metrics for parallel programs can provide an overwhelming amount of internal detail that is difficult to relate to performance bottlenecks. Our approach identifies sources of performance degradation via a sensitivity analysis which links program bottlenecks directly to the source code. Synthetic-Perturbation Tuning (SPT)[1] introduces the notion of inserting user-induced artificial delays into the source code and capturing the effect of such delays by employing design of experiments techniques.

In the rest of this section we describe the problems associated with conventional profiling techniques when applied to MIMD architectures. We also report on existing tools for tuning parallel program performance. Finally, we give an argument for program sensitivity analysis without conventional profiles. A step-by-step methodology of SPT is presented in Section 2. In Section 3 we extend the technique to address specific features that are the source of poor performance on the shared memory programming model. Sources of performance degradation include load imbalance, insufficient parallelism, synchronization, critical sections, and compute time bottlenecks. Section 4 illustrates the practicality of SPT by demonstrating its use on two case studies (an *image processing benchmark* and a parallel *quicksort*). The last section (Section 5) draws conclusions and describes future plans.

1.1 Motivation

Performance statistics have long been used to improve program execution efficiencies [18, 19, 7]. The most common statistics are frequency counts and timings for segments of code. Segments can be procedures or smaller entities, such as pieces of straight line code. Simple and intuitive to use, execution profiles reveal program bottlenecks that impede execution.

The advent of the MIMD parallel system raises two challenges to conventional profiling. The first problem is an exploding state space. Program profiles on serial machines implicitly define a set of disjoint execution states whose occupancies sum to a total response time. Each execution thread of an MIMD program defines a similar set of (sub)states. Unfortunately, the set of states for the whole MIMD program is enormous. To see this, imagine first a serial program with a main procedure and four callable procedures; there are five states at the procedural level of profiling. Now consider a parallel version of this program on a small, eight-processor system. Eight active threads, each with five substates, lead to a state set whose size is $5^8 = 390,625$ states. Processor inactivity will further increase this number. Choosing not to distinguish among functionally identical processors collapses some states into what can be termed macrostates, but the fundamental problem remains: The program state space becomes enormous as the scalable parallel system grows larger.

A second MIMD profiling challenge is the coupling among profile states caused by parallel execution. Conventional profile statistics require a much deeper interpretation in MIMD. This problem is to be expected. With separate threads of execution working on a joint computation, it is natural that communication and constraints must exist among threads. Interdependencies are manifest as latencies–a wait for a message, a pause prior to writing some shared variable. Because latencies are generated by circumstances of the system and program, they are not easily estimated. Latencies can range from negligible to devastatingly large.

1.2 Related Work

Existing techniques collect performance statistics in a number of ways. In the taxonomy shown in Table 1, the first row shows two common methods of defining events to be recorded. The first method is **periodic sampling** (I), which is tied to a clock and is based on collecting statistics. For example, at regular time intervals, an interrupt may be generated and the program counter at that point looked up in an allocation table. This gives the name of the procedure that was active at the clock tick. Periodic sampling is very popular for instrumenting systems that run an anonymous collection of programs. No changes are necessary to any user program. By adjusting the sampling frequency, the overhead can be adjusted to some convenient level. One big disadvantage is in testing coverage; if a piece of code is not recorded at having been run, it may in fact not have run, or the sampling may have been unlucky. The system also has problems with interpretive language systems, since locations within the interpreter mean little to a user.

I. **Periodic Sampling**	II. **Fixed Triggering**
a. **Traces**	b. **Histograms**

Table 1: Simple Taxonomy of Performance Techniques.

The second method of **fixed triggering** (II) uses identifiable locations or patterns, which when reached or matched, define an event. For instance, a special procedure call upon entry to a segment of executable statements will record information about the program at that point. Fixed triggering is bound to features of software or hardware. Hence, even if a few instructions of an instrumented procedure execute, this will be indicated accurately. Testing coverage for software is quite clear. A major drawback is setup. Each software or hardware event of interest must have corresponding triggers defined within the monitoring system. The technique is not generally satisfactory for a constantly changing population.

The bottom row of the table gives common types of recorded information. A **trace** (a) often comprises a record of a location in code or a configuration of a subsystem plus a time-stamp. A constant stream of traces is generated as system execution proceeds, and from this data much important behavior can be reconstructed. Unfortunately, the stream is often hard to manage because of its magnitude. Special collection hardware may be needed to handle the volume of data[13, 14].

Histograms (b) are an accumulative approach that demands little extra bandwidth. The number of invocations of a procedure, the overall time spent in a loop– these are of type (b), histogram or profile statistics. Because histogram information accumulates, they demand far less storage or bandwidth than do traces. The cost is a loss of detail, since time is not generally recorded except as an accumulated amount. No detailed times are kept of individual events.

Tools *gprof* and *quartz* are of type I-b. The VLSI instrumentation chip *MultiKron*[14] supports either II-a or II-b. *MTOOL*, triggered by basic program blocks, builds histograms and is therefore of type II-b. A type I-a is uncommon, since periodic random sampling yields an erratic set of data. It is not clear what detailed I-a traces could contribute when the actual information lies more in the aggregate statistical distribution of samplings than in any one sample.

1.3 Program Sensitivities without Conventional Profiles

A practical code improvement scheme depends upon identifying the most sensitive sections within a program, so that worst bottlenecks can be corrected. Fortunately, the conventional execution profile is not the only avenue. An alternate approach treats program and system together as an entity of essentially unfathomable complexity. Here, program segments $\{s_i\}$ suspected of being bottlenecks are explored via systematic perturbations of their code. This generates different versions of the program. Overall program responses are measured and recorded for each variant. The responses are then used to solve mathematically for sensitivities of the segments $\{s_i\}$. The question of state in this approach has been shifted from the executing program to simpler, source code defined settings. This new state space is smaller, clearer and static. Furthermore, there exists a whole body of mathematics that simplifies its handling and interpretation. This is the statistics of design of experiments (DEX)[2]. Experimental designs especially address interactions. Focusing upon perturbation settings and measured responses, the DEX analysis is designed to catch likely interactions that might impede good performance. Each segment in $\{s_i\}$ in and of itself might not impede parallel execution, but together, some combinations may cause disastrous slowdowns (see example in [1]). The DEX approach can indicate interactions easily and clearly.

One major problem in the past with applying DEX to software has been in finding suitable ways to perturb program code. Natural program parameters work fine, but they are not commonly available for arbitrary segments. An alternative is to recode a segment from $\{s_i\}$ in a new faster or slower version. The problem is the recoding, which must be made and checked very carefully for algorithmic correctness. The perturbed version must compute *exactly* the same internal and external results. Recoding is slow and checking is tedious. Furthermore, each segment must be treated in this *ad hoc* manner. The efficient solution is to make all perturbations artificial. By doing this, each synthetic perturbation is easily introduced or removed, and yet it does not interfere with the computation of the original code. Since synthetic code does *simulate* changes in coding to a segment, DEX analysis proceeds in its normal fashion.

2 Description of Technique

Synthetic-Perturbation Tuning (SPT) is an empirical approach that treats an MIMD program as a *black box* with input parameters and outputs. The SPT approach introduces synthetic perturbations (i.e., *artificial delays*) into source code segments and relies on (for design and analysis) a modern branch of statistical theory called *design of experiments* (DEX) [2, 3, 5, 20, 21]. DEX provides an efficient methodology for determining the relative sensitivity of the MIMD program to synthetic perturbations. SPT focuses the programmer's attention on the potential problem areas in the program. An important step in this methodology is to identify which segments of code are candidates for improvements. The identified code segments are termed *bottlenecks*. Each bottleneck is ranked quantitatively according to its sensitivity to synthetic perturbation. Such a list is called an **SPT Rank**. An SPT rank is a guide that can be used to improve (tune) the corresponding code segments.

The SPT premise is that if the program is highly sensitive to source code perturbations in a code segment (i.e., delay has a clearly detrimental effect on performance), then source code improvements to that segment will have an opposite (positive) effect. This premise is easy to justify for serial code since the SPT ranking can be done so that it corresponds to a combination of how often a section of code is executed and its execution time. In the next section, a justification of this premise as it applies to shared memory programs is provided.

In what follows, we describe the generic SPT methodology for tuning an MIMD program. Extensions of SPT for capturing specific features for the shared memory programming model are given in the next section.

1. **Determine objective and define test conditions.** The first step in SPT is to determine the goal of the tuning effort. A common objective of SPT is to make a rank-ordered list of the source code segments based on the relative sensitivity of the MIMD program to synthetic delays associated with the code segments. The segments that rank high on this list are potential bottlenecks during the execution of the program.

   To perform a set of SPT experiments, the user must define a set of test conditions. Test conditions include the source code implementation, data set, and machine. SPT's analysis applies to the defined test conditions. If these conditions change, a new set of SPT experiments and analysis may need to be performed. Based on our experience, given a source code and a machine, results for similar data sets are usually consistent.

2. **Choose candidate code segments.** A candidate code segment can be any section of code. Typically it is a function declaration, function call (usually for synchronization, e.g., a send protocol or a locking mechanism), critical section, or a loop construct. Selection of candidate code segments can involve a number of techniques. Important factors that help in narrowing the field of all possible code segments include the users knowledge of the program and code inspection. A conventional profiling tool can aid in this process as well. Automatic selection is also possible. The user can perform preliminary experiments on the set of all possible user defined code segments (e.g., the user may choose to examine all loop constructs or all critical sections). Brief experiments and analysis quickly screen out unlikely bottlenecks. This preparatory process reduces the field to a manageable size for which more exhaustive testing can be performed.

3. **Insert Perturbations.** Each candidate code segment is instrumented with a delay option (*delay* or *no delay*). *No delay* leaves the code unperturbed. *Delay* takes the form of a function call that performs a

specified number of instructions. The call does not alter the natural path of the program or the values of its variables. It merely attaches a specified number of instructions to that code segment. The delays could be of different lengths [1]. However, for simplicity, we opted to implement constant delays at the source code level. Thus in the example described, delay has two possible values, zero or a fixed number irrespective of the code segment. An example of how a delay might be implemented is given in the following pseudo C source code block:

```
while(v--) { /* factor 12 */          /* begin original code */
#if F12                               /* begin spt code      */
   spt_delay (delay_value);
#endif                                /* end spt code        */
              :
            Code
              :
}                                     /* end original code   */
```

spt_delay() is a function that performs a specified number of synthetic instructions corresponding to *delay_value*. The implementation of the delay function must yield a consistent delay while not altering the natural path of the program. The looping block **while**(v--) { ...} is a designated code segment and referred to as, for example, *factor 12* (*F12*). The statistical term **factor** is used to represent a candidate code segment. Conditional compilation creates multiple versions of the program corresponding to pattern of delays indicated by the experimental plan (next step).

The duration of the artificial delay is an important aspect. Ideally, the delay should be long enough so that it can easily be distinguished from noise and short enough so as not to produce unnecessarily long program execution times. The magnitude of the delay is often determined through trial and error. A discussion of important aspects for choosing the delay magnitude can be found in the extended version of [1]. In the next three steps we describe how SPT experimental design plans are developed and used.

Treatment	Factors			Response
	F1	*F2*	*F3*	
1	−	−	−	17.05
2	+	−	−	17.08
3	−	+	−	23.19
4	+	+	−	23.34
5	−	−	+	19.62
6	+	−	+	19.71
7	−	+	+	25.61
8	+	+	+	25.71

Table 2: 2^3**Complete Factorial Design for** *Xprog.*

4. **Design experimental plan.** Once the candidate code segments are determined, an experimental plan can be developed. There is no theoretical limit on how many distinct factors (source code segments) can be investigated on a given SPT iteration. A variety of schemes can be used for designing an experimental plan[2, 5]. A small 2^3 factor complete factorial example is given to illustrate the ideas of experimental designs. A 2^n plan indicates that the experiment has n factors each at 2 levels. Here we have $n = 3$ factors (called, for example *F1, F2*, and *F3*) and 2 levels (*no delay* (−) and *delay* (+)) for each factor.

   Suppose we have a MIMD program (call it *Xprog*) with three suspected bottleneck locations, *F1, F2*, and *F3* that correspond to certain code segments within *Xprog. F1* represents a *for loop* in the function *func_Y*, *F2* represents a *critical section* in the function *func_Z*, and *F3* represents a *while loop* in the function *func_Z.*

   With a three factor complete factorial plan, there are $2^3 = 8$ possible delay patterns each indicated by a row in Table 2. A plus sign (+) in a given row denotes that the corresponding code segment is perturbed, and a minus sign (−) indicates that the corresponding code segment is unperturbed (i.e., it retains its original code). In DEX terminology each delay pattern is a **treatment**. Table 2 lists the eight treatments and corresponding response which is the total execution time of the MIMD program. Note that the first treatment (all minuses) represents the original unperturbed program code.

The basis of an SPT rank associated with a source code segment is a quantitative measure called *main effect* associated with that code segment. In our 2^3 example the ranking of the three factors is based on their main effects (computation of the main effects will be illustrated). A main effect of a factor (code segment) is a measure of the sensitivity of the MIMD program to the artificial delay in that code segment. Depending on the experiment plan that is used, this measure can be affected in unknown ways by the interactions amongst the code segments. In this paper we propose the use of experimental plans called *resolution IV* plans that ensure that the main effects are not affected by the 2nd-order interactions amongst the code segments[2]. The total number of test runs required by a resolution IV plan with k factors is approximately $2k$.

5. **Run experiments according to plan and record a response.** Each treatment or version of the program is run and the corresponding response is recorded. The response can be any useful measurement; typically the response is the total program execution time. The treatments are usually run in a random order.

   In our example (Table 2), all eight versions of the program are compiled, run in random order and measured for execution time . The response time for each treatment of the program is given in the *Response* column.

6. **Analyze Results.** The object of data analysis is to evaluate the main effects associated with each factor. The computed values of the main effects are subsequently used to produce an SPT ranking of the factors. In addition to the main effects, a resolution IV plan provides a measure of the standard error (a measure of uncertainty) associated with the computed values.

   The **main effect** of a factor is the difference between two average responses, one corresponding to the treatments which have the (+) level of the factor and the other corresponding to the treatments which have the (−) level of the factor. For example, in the 2^3 plan (Table 2), the main effect of factor *F3* is the average response for treatments 5, 6, 7, and 8 (i.e., $[19.62 + 19.71 + 25.61 + 25.71]/4 = 22.66$), minus the average response for treatments 1, 2, 3, and 4 (i.e., $[17.05 + 17.08 + 23.19 + 23.34]/4 = 20.17$). Thus the main effect for *F3* is 2.49. The

Rank	Factor	Main Effect †	Routine	Construct
1	F2	6.10	func_Z()	*while* loop
2	F3	2.49	func_Z()	*critical section*
3	F1	0.09	func_Y()	*for* loop

† Standard Error of Main Effects: ± 0.06

Table 3: **SPT Rank of Main Effects for** *Xprog*.

main effects are organized into an ordered list to form an SPT ranking of the code segments. An example of an ordered list (from Table 2) that can be produced by SPT is shown in Table 3. This SPT rank is the format we use throughout the rest of the paper.

The first column of the SPT rank gives the standing of the corresponding code segment. A higher rank indicates a higher sensitivity to artificial delays (e.g., *F2* is most sensitive to the delay). Column 2 gives the factor number which provides a reference back to the source code location represented by the factor. The main effects column gives the sensitivity levels of the corresponding code segments as well as an estimate of the standard error[4]. The actual numbers are not as important as their relative magnitudes. Column 4 describes which function the section of code resides in. The last column indicates what type of construct the code segment is. By surveying Table 3 we can conclude that factor *F2* is the most significant. This code segment should be given first priority in the tuning effort.

7. **Improve bottlenecks and determine performance.** An SPT rank gives a list of potential bottlenecks. The bottlenecks so identified may or may not be improvable. Investigation begins with the higher ranked bottlenecks since they possess the greatest potential for improvement. These bottlenecks can be pursued further with SPT to gain more information about the bottlenecks or an attempt can be made to improve them.

   After improvements to the code are attempted the user must make a determination of performance. If the desired performance is obtained the process is complete. Otherwise the user can continue to investigate

[4]The standard error of the main effect is evaluated by treating high order interactions as errors from noise (see[2], page-327).

the program by using SPT to probe the code further.

This methodology provides the basic SPT framework. Within this framework, expanded issues relevant to a particular programming model and architecture can be handled. For example, on the shared memory programming model, programming concerns such as degree of parallelism, load balancing, critical sections, and synchronization can easily be investigated. The next section addresses these issues.

3 SPT Applied to Shared Memory Programs

The emergence of shared memory multiprocessors in the past decade has given rise to a substantial effort in designing and analyzing software for these machines. According to Bell [17], "the mainline, general-purpose computer is almost certain to be the shared memory, multiprocessor after 1995." Hence it is important to develop tuning tools for shared memory programs. The SPMD (Single Program, Multiple Data) model using a single address space is the natural programming model for shared memory multiprocessors. This programming model can be viewed as an evolution of the traditional programming model used for von Neumann architectures. The performance of a shared memory program depends crucially on several interrelated factors such as the amount of parallelism used, the degree to which the work load is balanced among the processors, the contention over shared resources (interconnection network, bus, memory), and the overhead incurred by synchronization. Unless a balance taking into consideration the relative importance of these factors is maintained, the actual performance of shared memory programs will be disappointing. In fact, experimental work thus far bears this out. In the rest of this section, we describe our approach for finding bottlenecks related to each of these aspects as they arise in a shared memory program (some strategies apply to message passing environments as well). The SPT approach described in the previous section is expanded to determine the sources of potential bottlenecks. In the next section, we illustrate the use of these techniques on two case studies, the *Image Understanding Benchmark*, and a parallel version of the *quicksort* algorithm.

Degree of Parallelism: A typical MIMD program contains a mix of scalar, serial, vector, and parallel operations. A section of code with insufficient parallelism is a bottleneck if its execution time is significant compared to the overall execution time of the program. Such a bottleneck can be detected only if the performance of the program is analyzed as a function of the number of processors involved. In fact, by Amdhal's law, for a given program, it is the execution time of the serial portions that will ultimately determine the speed of the program as the number of processors increases (and the input size is held constant). Our method is based on an extension of this observation.

We insert artificial delays into the sections of code under investigation. We then perform the design of experiments on successively scaled-up versions of the system. As the number of processors increases, the effects of the parallel code will become less important while the effects of the serial code will become more significant.

Consider for example a section of code that multiplies an $n \times n$ matrix A by a vector x to generate the vector $y = Ax$. Partition A as follows

$$A = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_p \end{bmatrix}$$

where each A_i is of size $(n/p) \times n$, n/p is assumed to be an integer, and p is the number of processors available. The following section of the code corresponds to the computation $A_i x$ performed by the ith processor

```
**for** $j = (i-1)\frac{n}{p} + 1$ **to** $i\frac{n}{p}$ **do**
    $y(j) := 0$
    **for** $k = 1$ **to** $n$ **do**
        SPT-delay
        $y(j) := y(j) + A(j,k)x(k)$
    **end**
**end**
```

The execution time of this section of code is proportional to $\frac{n^2}{p}(\Delta + 2t_{fp})$, where Δ is the SPT delay time, and t_{fp} is the time it takes to execute a floating-point add or multiply (assumed to be equal for simplicity). Hence the effect of the SPT delay is a net increase of $\frac{n^2}{p}\Delta$ in the total execution time; thus, it represents a factor whose effect is a decreasing function of

p. Therefore, the effect of the parallel code becomes less important as the system is scaled-up.

Load Balancing: The speedup achieved by a parallel program is primarily due to the development of threads of execution that can be run concurrently. This can be done either by using functional or data decomposition present (explicitly or implicitly) in an existing algorithm, or by developing a new algorithm that has a higher degree of (functional or data) parallelism. With functional decomposition, each processor is responsible for executing a different function, and hence the distribution of the loads among the processors is completely dependent on the computational requirements of these functions. Similarly, data decomposition can result in some processors having to handle much larger amounts of data than the rest of the processors.

A load balancing problem can be viewed as insufficient parallelism that, in general, arises dynamically. The insertion of artificial delays followed by an SPT analysis allows us to determine each section of the code that generates a significant load imbalance. Notice that an SPT delay will cause the processor with the heaviest load to run even slower and hence its SPT effect will be significant. Consider for example the case when there are p processors that have to be assigned to process (say, search for a specific item) n lists L_j of different sizes, for $0 \leq j < n$. We can use data decomposition by assigning the ith processor to process the lists $L_i, L_{i+p}, \cdots$, for $0 \leq i < p$. The following program segment illustrates such a decomposition.

```
for (i = id; i < n; i+ = p)do
    { for(j =head(i); j! =NULL; j = j- >next)do
          SPT-Delay
          { Process Node j}
    }
```

The impact of the SPT delay Δ is proportional to $\Delta \max_{0 \leq i < p}\{|L_i| + |L_{i+p}| + \cdots\}$. It follows that the larger the total size of the lists to be processed by a single processor, the more significant the SPT contribution of the corresponding factor.

In the Image Understanding benchmark that we study in the next section, we use this technique to predict the load imbalance that is caused by the procedure to determine the connected components of an image. In this case, the processor assigned to handle the background pixels has much more work to do than the remaining processors. Without analyzing the procedure, our SPT analysis was able to determine the load imbalance resulting from this procedure and to predict its importance as the number of processors increases.

Critical Sections and Synchronization: Processors executing a shared memory program may waste a substantial amount of time trying to enter a critical section ("busy wait") or trying to synchronize their activities. SPT can be used to provide information concerning any significant overhead incurred in a critical section or at a synchronization point. We start by handling critical sections.

The insertion of an artificial delay into a critical section allows us to perform an SPT analysis similar to the previous two cases. We claim that, for a critical section that represents a significant bottleneck in the program, its SPT effect will become more important as we scale-up the system. In fact, the overall contribution of the delays tends to be cumulative with respect to the number of processors that are trying to access the critical section.

As for synchronization, we cannot use the technique in a straightforward way. However we can extend it as follows. For each synchronization barrier, we insert two types of perturbations, one immediately before the barrier and the other immediately after the barrier. The perturbation *FB1* inserted before the barrier consists of an artificial critical section, while the perturbation *FB2* inserted after the barrier consists of an artificial critical section followed by an artificial barrier. The justification of the perturbation *FB2* is as follows. The critical section delay in *FB2* is an obvious bottleneck since all the released threads try to execute it at once whereas the artificial barrier ensures that the new program is functionally identical to the original one. We then run our experiments and compare the effects of *FB1* and *FB2*. If their effects are about the same, we can conclude that the synchronization cost is marginal. The argument is that in this case *FB1* is also being pressed for execution by many threads, which is indicative of how threads arrive at the barrier– all together – a good parallel execution. As the difference in the two effects increases, the synchronization cost increases. Threads that arrive one-by-one at *FB1* will not find it much of a bottleneck and hence its effect will be lower than that of *FB2*. It follows that by comparing the effects of

FB1 and *FB2*, we will be able to diagnose a barrier being used efficiently. This method is applied in the next section to a quicksort program that contains several synchronization points and is shown to identify properly the costly synchronizations.

Summary: SPT can be used to detect bottlenecks due to lack of parallelism, load imbalance, and critical sections, by simply inserting artificial delays into appropriate sections of the code and conducting a design of experiments and an SPT analysis as described in Section 2. As for synchronization, we can insert two types of delays, one immediately before and the other immediately after each synchronization barrier, and conduct the design of experiments and an SPT analysis as before. Therefore SPT can be used to detect the main sources of inefficiency in a shared memory program. In the next section, we illustrate our techniques on two detailed case studies.

4 Case Studies

4.1 Image Processing Benchmark

In this section we present a practical shared memory tuning example based upon a large image processing benchmark. The test code is the *Image Understanding Benchmark* for parallel computers developed at the University of Massachusetts at Amherst[15]. The benchmark was described as a "complex benchmark that would be almost impossible to tune" [15]. Using SPT, we demonstrate how important bottlenecks were identified and subsequently analyzed and improved.

The benchmark was designed to test common vision tasks on parallel architectures. It consists of a model-based object recognition problem, given two sources of sensory input, intensity and range data, and a collection of candidate models. The intensity image is a 512 × 512 array of 8-bit pixels, while the depth image consists of a 512 × 512 array of 32-bit floating point numbers. The models contain rectangular surfaces, floating in space, viewed under orthographic projection. Added to the configuration is both noise and spurious nonmodel surfaces. The benchmark's task is to recognize an approximately specified 2 1/2-dimensional "mobile" sculpture in a cluttered environment. The sculpture is a collection of 2-dimensional rectangles of various sizes, brightnesses, orientations, and depths.

The experiments are performed on both a ten processor and twenty-six processor Sequent Symmetry. The Image Understanding Benchmark package comes with a number of data sets and their corresponding outputs. The example presented here uses test set number two. The benchmark consists of more than 50 procedures and has approximately 3500 lines of C code.

Our objective for performing an SPT analysis on this example is to screen the code for potential bottlenecks at different levels of parallelism. We selected 31 factors (loops, function declarations, and critical sections) as potential candidates for bottlenecks based on code inspection. An experimental plan is selected to handle the large number of code segments that need to be investigated. The image benchmark is instrumented with an SPT delay for each factor. The treatments are run in a random order and the overall execution time of the program is recorded as the response.

Table 4 lists the main effects of the 31 factors of the image processing benchmark running on 8 processors. This initial set of experiments indicates that the three top ranked procedures (Gradient Magnitude, Median Filtering, and Connected Components) represent major bottlenecks. Hence tuning the corresponding code segments should be given first priority. Notice that none of the top ranked factors involves a critical section or a synchronization barrier. Therefore the emphasis of the tuning effort should concentrate on increasing the efficiency of the serial sections within the loops (corresponding to factors $F17$, $F26$ and $F2$), or better balancing the load among the processors, or increasing the degree of parallelism. Since factor $F17$ was ranked highest, we concentrated initially on the corresponding code segment.

The Gradient Magnitude procedure performs a standard 3×3 Sobel operation on the depth image. The section of code within the loop corresponding to factor $F17$ is quite inefficient. After removing multiplications by zeros, and reducing the total number of remaining multiplications, the execution time of the procedure improved 300%. At this point, the relative ranking of the procedure dropped to 8 with 8 processors(Table 9).

Our next task was to consider the Median Filtering procedure. While we were attempting to tune this procedure, we discovered that the procedure generated erroneous results. At this time we switched our efforts to tuning the third procedure, Connected Components. This procedure assigns a unique label to each contiguous collection of pixels having the same intensity level value. To gain a better understanding, we ran our experiments using 2, 4, 8, and 24 processors. Tables 5, 6, 7, and 8 show the resulting rankings of the

Rank	Factor	Main Effect †	Routine	Construct
1	17	6.03	Gradient Magnitude	*for* loop
2	26	5.46	Median Filtering	*while* loop
3	2	5.26	Connected Components	*for* loop
4	1	3.94	Connected Components	function
5	4	3.84	Connected Components	*while* loop
6	25	2.01	Median Filtering	*for* loop
7	20	1.67	Match	function
8	29	1.36	Probe	*for* loop
9	6	0.64	Extract Cues	*for* loop
10	21	0.53	Match	*for* loop
11	19	0.16	K-curvature	*for* loop
12	18	0.10	K-curvature	*for* loop
13	12	0.10	Complete Match	*critical section*
14	11	0.08	Complete Match	*while* loop
15	13	0.08	Complete Match	*while* loop
16	8	0.05	Complete Match	function
17	10	0.05	Complete Match	*critical section*
18	24	0.05	Median Filtering	*while* loop
19	5	0.05	Connected Components	*while* loop
20	15	0.04	Extract Cues	*critical section*
21	16	1.04	Complete Match	*critical section*
22	3	0.04	Connected Components	*while* loop
23	14	0.03	Complete Match	*critical section*
24	28	0.03	Probe	function
25	7	0.02	Complete Match	*while* loop
26	27	0.02	Probe	*for* loop
27	22	0.02	Median Filtering	*for* loop
28	23	0.01	Median Filtering	*for* loop
29	9	0.01	Complete Match	function
30	31	0.00	Trace Boundary	*while* loop
31	30	0.00	Graham Scan	*while* loop

† Standard Error of Main Effects: ±0.04

Table 4: **SPT Rank for** ***Image Benchmark*****, 8 Processors.**

major factors (on the original code) as a function of the number of processors.

It is immediately clear that there is a serious load balancing problem: the three factors (*F1, F2, F4*) corresponding to Connected Components have gradually moved to the very top of the table as the number of processors increased. A close examination of the procedure confirms our suspicion. One processor is assigned to handle the background pixels and hence ends up doing most of the work. A completely different scheduling policy or a completely new algorithm is required before a significant improvement can be made. Even by making slight modifications, we were able to improve the performance of this procedure.

We now show the results of the SPT analysis when performed on our improved version. We have modified the Gradient procedure as indicated earlier and have made some simple modifications to the Connected Components procedure. Tables 9 and 10 show a summary of the SPT analysis when performed on our improved version. Notice that the Gradient procedure (rank=8 with 8 processors, and rank=17 on 24 processors) is no longer a significant bottleneck and that the Median, Connected Components, and Probe contribute much more significantly to the overall running time when the number of processors increases beyond eight. Using eight-processors, our version runs 18.2% faster than the original version.

4.2 Parallel Quicksort

The image processing benchmark provided insights on how SPT can be used to handle large applications. It successfully detected code inefficiencies and a

Rank	Factor	Main Effect †	Routine	Construct
1	17	24.04	Gradient Magnitude	*for* loop
2	26	22.18	Median Filtering	*while* loop
3	25	8.27	Median Filtering	*for* loop
4	2	5.26	Connected Components	*for* loop
5	1	4.50	Connected Components	function
6	4	4.34	Connected Components	*while* loop

† Standard Error of Main Effects: ±0.04

Table 5: **SPT Rank for** *Image Benchmark*, **2 Processors.**

Rank	Factor	Main Effect †	Routine	Construct
1	17	12.06	Gradient Magnitude	*for* loop
2	26	11.09	Median Filtering	*while* loop
3	2	5.27	**Connected Components**	*for* loop
4	4	4.35	**Connected Components**	*while* loop
5	1	4.33	**Connected Components**	function
6	25	4.09	Median Filtering	*for* loop

† Standard Error of Main Effects: ±0.04

Table 6: **SPT Rank for** *Image Benchmark*, **4 Processors.**

Rank	Factor	Main Effect †	Routine	Construct
1	17	6.03	Gradient Magnitude	*for* loop
2	26	5.46	Median Filtering	*while* loop
3	2	5.26	**Connected Components**	*for* loop
4	1	3.94	**Connected Components**	function
5	4	3.84	**Connected Components**	*while* loop
6	25	2.01	Median Filtering	*for* loop

† Standard Error of Main Effects: ±0.04

Table 7: **SPT Rank for** *Image Benchmark*, **8 Processors.**

Rank	Factor	Main Effect †	Routine	Construct
1	2	5.43	**Connected Components**	*for* loop
2	1	3.95	**Connected Components**	function
3	4	3.93	**Connected Components**	*while* loop
4	17	2.14	Gradient Magnitude	*for* loop
5	26	2.03	Median Filtering	*while* loop
6	20	1.55	Match	function

† Standard Error of Main Effects: ±0.12

Table 8: **SPT Rank for** *Image Benchmark*, **24 Processors.**

Rank	Factor	Main Effect †	Routine	Construct
1	26	5.57	Median Filtering	*while* loop
2	2	5.30	Connected Components	*for* loop
3	1	4.06	Connected Components	function
4	4	3.93	Connected Components	*while* loop
5	25	2.04	Median Filtering	*for* loop
6	20	1.64	Match	function
7	29	1.23	Probe	*for* loop
8	17	0.59	Gradient Magnitude	*for* loop
9	6	0.59	Extract Cues	*for* loop

† Standard Error of Main Effects: ± 0.04

Table 9: **SPT Rank for** *Improved Image Benchmark*, **8 Processors.**

Rank	Factor	Main Effect †	Routine	Construct
2	2	5.85	Connected Components	*for* loop
4	4	4.66	Connected Components	*while* loop
3	1	3.80	Connected Components	function
1	26	2.31	Median Filtering	*while* loop
7	29	2.22	Probe	*for* loop
9	6	1.82	Extract Cues	*for* loop
.	.	....	....	....
17	17	0.60	Gradient Magnitude	*for* loop

† Standard Error of Main Effects: ± 0.76

Table 10: **SPT Rank for** *Improved Image Benchmark*, **24 Processors.**

load imbalance. However, synchronization and critical sections did not play a significant role. In this section, we discuss a parallel version of the *quicksort* algorithm and illustrate how SPT can be used to address bottlenecks due to synchronization and critical sections.

The test code is a parallel implementation of Hoare's *quicksort* algorithm[16]. *Quicksort* is a scheme that is based on partitioning a given list into two sublists relative to a selected member of the list, called the *pivot*. Elements of the list are rearranged such that all elements smaller than the pivot are to the *left* of the pivot and all elements greater than the pivot are to the *right* of the pivot. There are several ways of choosing the pivot to induce approximately equal partitions. We refer to a such partitioning step as a *pass*. Hence after a pass, the pivot value is positioned in its sorted order. This procedure is then applied recursively to each sublist. Once a sublist becomes small enough, it can be sorted by using a simple sorting routine, say selection sort or bubble sort.

A simple way to parallelize the quicksort procedure is to allocate newly-created sublists to available processors (see[4] for a more involved parallelization of quicksort). A sublist assigned to a processor is then partitioned into two sublists by that processor. The allocation of sublists to processors is controlled by a shared stack. An idle processor asks for a sublist from the shared stack. To insure that no two processors take possession of the same sublist, the stack access is controlled by a critical section.

The following is a skeleton of the program code for a simple implementation of *quicksort*.

```
Initializations;
Put list on stack;
barrier(); /* barrier #1 */
while(stack is not empty) {
        barrier();  /* barrier #2 */
        lock(stack_lock);
                                        if(stack is not empty)
                                          pop();
        unlock(stack_lock);
        Select a pivot and partition current list into sublists $L_1$ and $L_2$;
                                          if($|L_1| > |L_2|$) {
                                          lock(stack_lock);
                                          push($L_2$);
                                          push($L_1$);
                                          unlock(stack_lock); }
                                          else {
                                          lock(stack_lock);
                                          push($L_1$);
                                          push($L_2$);
                                          unlock(stack_lock); }
        barrier();  /* barrier #3 */
        }
```

Our tuning effort of *quicksort* begins by investigating the cost of synchronization. There are three synchronization points, denoted as *barrier()*. The first barrier insures that all initializations are complete before the processes begin executing the **while** loop. The two barriers within the main loop synchronize the processes before and after each pass. This implementation makes it easy to determine when the sort is completed. Our SPT objective is to find out if processes are arriving at widely dispersed times, and hence causing many processors to idle for a significantly long period of time. Our investigation follows the treatment method presented in Section 3. For each synchronization barrier, two types of perturbations are inserted, one immediately before (*FB1*) and the other immediately after (*FB2*). The method is illustrated by the following code segment:

Paired Factor	Main Effect	Difference †
barrier pair 1	*FB1:* 0.16	
	FB2: 0.22	0.06
barrier pair 2	*FB1:* 14.22	
	FB2: 15.22	1.00
barrier pair 3	*FB1:* 6.78	
	FB2: 15.34	8.56

† Standard Error of the Difference: ±0.21

Table 11: **Paired Effects for *Quicksort's* Barriers.**

```
lock(spt_lock_1);              /*               */
    spt_delay(spt_delay);      /*     FB1       */
unlock(spt_lock_1);            /*  treatment    */

barrier();                     /* original barrier */

lock(spt_lock_1);              /*               */
    spt_delay(spt_delay);      /*     FB2       */
unlock(spt_lock_1);            /*  treatment    */
barrier();                     /*               */
```

The three synchronization barriers are instrumented as shown above. This implementation demands six factors, two for each barrier tested. The experiments proceed as before; an experimental plan is created and tested. The resultant is an effect measure for all six factors. The interpretation of the results differ slightly in that we now want to compare the effects of the factors before and after each barrier. Table 11 shows the results. The left-most column of Table 11 identifies the barrier. The second column gives the calculated main effect for each factor. The individual main effects are meaningless in isolation and must be paired up and compared to obtain the proper information. The last column, which contains the difference of each of the paired factors, gives an indication of the cost associated with each synchronization. Remember that the treatment *FB2* shows the effect of an ideal barrier application, and is very sensitive to delay. If the paired effects are about the same, we conclude that the synchronization cost is marginal. As the difference in the two effects increases, the synchronization cost increases.

(Recall that *FB1* has less effect on straggling threads.) It follows that by comparing the effects of each pair of the delays introduced for each synchronization barrier, we will be able to determine those incurring large overheads. It should be noted that this type of experiment should be performed separately from a screening experiment. The effects have no relationship to the screening for important factors because the treatments are not comparable in any easy fashion.

In spite of its simplicity, this example illustrates the effectiveness and the generality of the SPT approach. The difference shown for the first synchronization barrier indicates that almost all processors arrive there at the same time. This is clearly the case since only one processor is responsible for the initialization phase and the rest crowd around the barrier. Used only once, the effects also show that this barrier is not very important to performance. The second synchronization barrier is not needed since the processors are already synchronized at the beginning of each pass. The test confirms what algorithm inspection tells us. The third row of the table indicates that the third synchronization barrier is costly compared to the other two synchronization barriers. This is because processors are working on different-length sublists (or no sublist at all) and hence arrive at the third synchronization point at widely different times. The barrier deserves some attention.

To alleviate the problem of synchronization at the end of the **while** loop, we rewrite the code following the skeleton shown next. The resulting improvement in performance is substantial (78%).

```
Initializations;
Put list on stack;
barrier();
for() {
    lock(stack_lock); /* CS1 */
        if(stack is not empty){
            pop();
        }
    unlock(stack_lock);
    if(!qsort_done) {
        Select a pivot and partition current list into sublists $L_1$ and $L_2$;
        if($|L_1| > |L_2|$) {
            lock(stack_lock); /* CS2 */
            push($L_2$);
            push($L_1$);
            unlock(stack_lock);
        } else {
            lock(stack_lock); /* CS3 */
            push($L_1$);
            push($L_2$);
            unlock(stack_lock);
            }
        }
    }
```

In the next experiment, SPT's objective is to obtain the relative importance (detrimental effect) of the three new critical sections (CS1, CS2, CS3). A delay is inserted in each critical section. An experimental plan is developed and run. Table 12 shows the SPT performance information for each critical section. By looking at the program, it is not clear which critical section presents the main bottleneck among the three critical sections. Our SPT analysis shows that the first critical section dominates. At this point, we remove the other two factors from further consideration, and perform a complete SPT analysis that includes the factor (labelled $F1$) of the

Rank	Factor	Main Effect †	Routine	Construct
1	1	13.82	main()	*critical section 1*
2	3	2.86	main()	*critical section 3*
3	2	2.62	main()	*critical section 2*

† Standard Error of Main Effects: ±2.15

Table 12: **SPT Rank for *Quicksort's* Critical Sections.**

Rank	Factor	Main Effect †	Routine	Construct
1	F3	29.01	partition_list()	*while loop*
2	F7	8.94	swap()	function
3	F4	1.26	push()	function
4	F6	0.69	bubble_sort()	*while loop*
5	F1	0.14	main()	*critical section 1*
6	F2	0.11	select_pivot()	function
7	F5	0.09	pop()	function

† Standard Error of Main Effects: ±0.39

Table 13: **SPT Rank for *Quicksort*.**

critical section $CS1$. Six additional code segments are selected to be tested along with this critical section. These are the procedures: partition_list(), bubble_sort(), swap(), push(), pop(), and select_pivot(). Since the delay for the critical section and regular code segments are equivalent, they can be compared. Table 13 shows the results obtained. Clearly factors *F3* and *F7* dominate the overall performance. Based upon this data, we examine the procedure partition-list() which calls the swap() procedure. Removing the calls to swap() and inserting its code into partition_list() resulted in an additional 23% improvement of the execution time of quicksort. This improvement has been reported earlier in [1] via SPT. As shown in the same paper, using the UNIX profiling tool *gprof* would have provided little information for improving the parallel quicksort routine.

5 Conclusion

We have described the tuning methodology of SPT, Synthetic-Perturbation Tuning, that is based on a branch of statistics called design of experiments. The main purpose of this methodology is to identify performance bottlenecks present in MIMD programs. SPT should provide the basis of a very powerful tuning tool that is portable across machines and architectures. We also considered in some detail the sources of poor performance on the shared memory model and showed how these issues can be adequately captured using SPT. Two detailed case studies were then discussed and their bottlenecks analyzed using our methodology. Significant improvements were made based on the results of the SPT analysis.

The work presented here should be viewed as a contribution towards developing a comprehensive methodology for tuning MIMD programs based on the techniques of the design of experiments. We are currently refining and extending our methodology in several directions. In particular, we are analyzing approaches to measure the performance of memory hierarchy in a shared memory environment, and the communication overhead present in a message passing environment. Additional large case studies are currently being examined using SPT. Our future plans include the development of automated tools for performing the SPT analysis and reporting the appropriate information to the user.

A minor disadvantage of our methodology is the amount of experimentation necessary to perform the analysis. However, we believe that tuning MIMD programs is a highly nontrivial task requiring the capture of many parameters and their interactions. Simpler schemes are likely to fail in one aspect or another. The mathematical basis of our method provides a solid foundation upon which we can build general tuning techniques that are applicable across machines and architectures.

References

[1] G. Lyon, R. Snelick, R. Kacker. "TPT: Time-Perturbation Tuning of MIMD Programs." Proceeding of the 6th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Edinburgh, Scotland, September 1992, 211-224. Edinburgh Uni-

versity Press Ltd. (an extended version of this paper exists as a Natl. Inst. of Standards and Technology (NIST) internal report, NISTIR 5131)

[2] G. Box, W. Hunter, J. Hunter, Statistics for Experimenters (1978), John Wiley and Sons Inc., New York.

[3] R. Jain. The Art of Computer Systems Performance Analysis. J. Wiley & Sons (New York, 1991), 720 pp.

[4] J. JáJá. An Introduction to Parallel Algorithms. Addison-Wesley, 1992, 556 pp.

[5] R. Kacker, E. Lagergren, J. Filliben. "Taguchi's Orthogonal Arrays are Classic Designs of Experiments." J.Res. Natl. Inst. Stand. Technol. 96, 5(Sept.-Oct. 1991), 577-591.

[6] T. Anderson and E. Lazowska. "Quartz: A Tool for Tuning Parallel Program Performance." Processings, SIGMETRICS 1990 Conference, May 1990, 115-125.

[7] S. Graham, P. Kessler, and M. McKusick. "Gprof: A Call Graph Execution Profiler." "Proceeding, ACM SIGPLAN Symposium on Compiler Construction, June, 1982.

[8] A. Goldberg and J. Hennessy. "Performance Debugging Shared Memory Multiprocessor Programs with MTOOL." In Proceedings Supercomputing, pp. 481-490, Nov. 1991.

[9] M. Martonosi, A. Gupta, T. Anderson. "MemSpy: Analyzing Memory System Bottlenecks in Programs." Performance Evaluation Review, Vol. 20, No. 1, June 1992.

[10] P. Newton and J. Browne. "The Code 2.0 Graphical Parallel Programming Language." In Processings ACM International Conference on Supercomputing, pp. 167-177, July 1992.

[11] H. Burkhart and R. Millen. "Performance-Measurement Tools in a Multiprocessor Environment." IEEE Transactions on Computers, Vol. 38, No. 5, May 1989.

[12] B. Miller, M. Clark, J. Hollingworth, S. Kierstead, S. Lee, T. Torzewski. "IPS-2: The Second Generation of a Parallel Program Measurement System." IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 2, April 1990.

[13] A. Mink, R. Carpenter, G. Nacht, J. Roberts. "Multiprocessor performance-measurement instrumentation." IEEE Computer: 63-74; September 1990.

[14] A. Mink and R. Carpenter. "Operating Principles of MULTIKRON Performance Instrumentation for MIMD Computers." Natl. Inst. of Standards and Technology, Gaithersburg, MD., NISTIR-4737; March 1992. 23 p.

[15] C. Weems, E. Riseman, A. Hanson, A. Rosenfeld. "The DARPA Image Understanding Benchmark for Parallel Computers." Journal of Parallel and Distributing Computing 11, 1-24, 1991.

[16] C. Hoare. "Quicksort." Computer Journal 5, 1(January 1962), 10-15.

[17] G. Bell. "Ultracomputers, A Teraflop before its Time." Communications of the ACM, Vol. 35, No. 8, August 1992.

[18] J. von Neumann and H. Goldstine. "Planning and Coding of Problems for an Electronic Computing Instrument", Institute for Adv. Study, Princeton, N.J. (3 vols.), 1947-1948. Reprinted in von Neumann's *Collected Works* (A. Taub, ed.), vol. 5, Pergamon Press, Oxford, 1963.

[19] D. Knuth. "An Empirical Study of FORTRAN Programs." Software–Practice and Experience 1, (1971), pp. 105-133.

[20] D. Montgomery, Design and Analysis of Experiments (1976), John Wiley and sons Inc., New York.

[21] C. Daniel, Applications of Statistics to Experiment Design (1976), John Wiley and Sons Inc., New York.