The Search for Lost Cycles: A New Approach to Parallel Program Performance Evaluation

Mark E. Crovella and Thomas J. LeBlanc
{crovella,leblanc}@cs.rochester.edu

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 479

December 1993

Abstract

Traditional performance debugging and tuning of parallel programs is based on the "measure-modify" approach, in which detailed measurements of program executions are used to guide incremental changes to the program that result in better performance. Unfortunately, the performance of a parallel algorithm is often related to its implementation, input data, and machine characteristics in surprising ways, and the "measure-modify" approach is unsuited to exploring these relationships fully: it is too heavily dependent on experimentation and measurement, which is impractical for studying the large number of variables that can affect parallel program performance. In this paper we argue that the problem of selecting the best implementation of a parallel algorithm requires a new approach to parallel program performance evaluation, one with a greater balance between measurement and modeling. We first present examples that demonstrate that different parallelizations of a program may be necessary to achieve the best possible performance as one varies the input data, machine architecture, or number of processors used. We then present an approach to performance evaluation based on *lost cycles analysis*, which involves measurement and modeling of **all** sources of overhead in a parallel program. We describe a measurement tool for lost cycles analysis that we have incorporated into the runtime environment for Fortran programs on the Kendall Square KSR1, and use this tool to analyze the performance tradeoffs among implementations of 2D FFT and parallel subgraph isomorphism. Using these examples, we show how lost cycles analysis can be used to solve the problems associated with selecting the best implementation in a variable environment. In addition, we show that this approach can capture large amounts of performance data using only a small number of measurements, and that it is flexible enough to allow conclusions to be drawn from empirical data in some cases, and analytic results in other cases.

This research was supported under NSF CISE Institutional Infrastructure Program Grant No. CDA-8822724, and ONR Contract No. N00014-92-J-1801 (in conjunction with the DARPA HPCC program, ARPA Order No. 8930). Mark Crovella is supported by an ARPA Research Assistantship in High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland.

1 Introduction

The performance of a parallel algorithm is often related to its implementation, input data, and machine characteristics in surprising ways. Most programmers do not experiment with a wide variety of alternative implementations or data inputs when tuning an application however. This lack of experimentation is due primarily to the difficulty of restructuring a parallel application, and the enormous number of possible implementations and inputs. Recent work on programming languages and environments is designed to address the first problem, by allowing the user to easily implement multiple parallelizations within the same program framework [Alverson and Notkin, 1992; Coffin, 1990; Crowl and LeBlanc, 1991; Halstead, 1985]. Nonetheless, the sheer size of the parameter space effectively limits the extent of experimentation.

In this paper we argue that understanding the behavior of parallel programs as a function of implementation and input represents a performance evaluation problem that is important and yet fundamentally different from the traditional view of parallel performance tuning. Given that multiple alternative implementations of a parallel program may be needed to achieve the best possible performance as one varies the input, the machine characteristics, the number of processors used, and other aspects of the execution environment, how does the programmer sort through the many possible implementations and determine the circumstances under which each performs best? More specifically, how does the programmer predict crossover points at which one implementation outperforms another?

In this paper we describe a methodology and associated tools for solving these problems. Our approach is based on metrics for evaluating sources of overhead in parallel programs, referred to as *lost cycles*. These metrics, and our entire approach, draw on existing work in both performance evaluation and scalability analysis.

We describe a measurement and evaluation tool set based on our metrics that combines the advantages of empirical performance measurement with the predictive power of analytic performance modeling. Although our focus is on demonstrating the utility of our approach and tools when solving the problems that arise in selecting among alternative implementations, we also show that the tools can be used to predict large amounts of performance data based on a small number of measurements.

2 Solving the Best Implementation Problem

2.1 An Example Problem

An example that demonstrates the difficulties posed by multiple implementations is parallelizing an algorithm for the *subgraph isomorphism* problem. Given two graphs, one small and one large, the subgraph isomorphism problem is to find one or more isomorphisms from the small graph to arbitrary subgraphs of the large graph. An isomorphism is a mapping from each vertex in the small graph to a unique vertex in the large graph, such that if two vertices are connected by an edge in the small graph, then their corresponding vertices in the large graph are also connected. The basic algorithm we use organizes possible solutions into a tree, and searches the tree for actual solutions. Subgraph isomorphism is NP-complete,

but by applying *filters* at each node of the search tree, large portions of the search space can often be pruned, allows solutions to be found in a reasonable amount of time.

This algorithm has a number of potential parallelizations including:

Tree parallelism searches subtrees of the root node in parallel;

Loop parallelism parallelizes the loops within each filter (since all the filters work by iterating over the nodes in each graph);

Instruction parallelism packs graph connectivity data as bitmaps into words, allowing set intersection operations to be implemented as Boolean operations within the filter loops.

We have created a program to solve subgraph isomorphism that can implement tree, loop, and instruction parallelism, in any combination. In this paper, we refer to these variations in the structure of a program, such as different algorithms, different parallelizations, different task schedules, and different synchronization methods as different *implementations.*[1] Thus our program incorporates 8 different implementations.

Our program runs on 7 shared-memory multiprocessors: the Sequent Balance, the Sequent Symmetry, the Silicon Graphics Iris, the BBN Butterfly, the BBN TC2000, the IBM 8CE, and the KSR1. All of these machines have at least 8 processors; on some machines we used as many as 32 processors.

Input to the program consists of two graphs, generated randomly. Based on the random process used to construct the two graphs, we can estimate the probability that any given leaf node in the search tree represents a valid isomorphism, which we call the *density* of the solution space. When the small graph has few edges and the large graph has many edges, the solution space is dense; when the small graph has many edges and the large graph has few edges, the solution space is sparse.

The program can search for any number of isomorphisms; in our experiments we vary the number of solutions requested from 1 to 256. We refer to this as varying the *problem,* since the implementation and input are fixed.

Different parallelizations have widely differing performance as a function of machine, number of processors, input, and problem. The performance of each parallelization is a function whose domain is this 4-dimensional space. Problems requiring selecting among the various parallelizations can come in many forms:

- for a fixed machine, number of processors, and problem, we may need the best parallelization as we vary the input density;
- for a fixed machine, number of processors, and input density, we may need the best parallelization as we vary the problem;
- for a fixed machine, input density, and problem, we may need the best parallelization as we vary the number of processors; and

[1] Different implementations could even include the use of different runtime libraries or various compiler optimizations.

Varying:	Machine				Density				Problem		
Fixed:	128 solns, sparse				Butterfly, 1 soln				Symmetry, dense		
	8CE	Iris	Symm.	KSR1	10^{-5}	10^{-21}	10^{-35}	empty	1	128	256
Loop	24.7	2.06	29.7	10.7	0.73	33.7	541.5	1.77	0.32	1.31	2.32
Tree	36.1	2.60	15.8	2.24	2.33	3.76	8.00	1.49	1.32	1.67	1.80

Table 1: Comparison of Loop and Tree Parallelism in Varying Environment

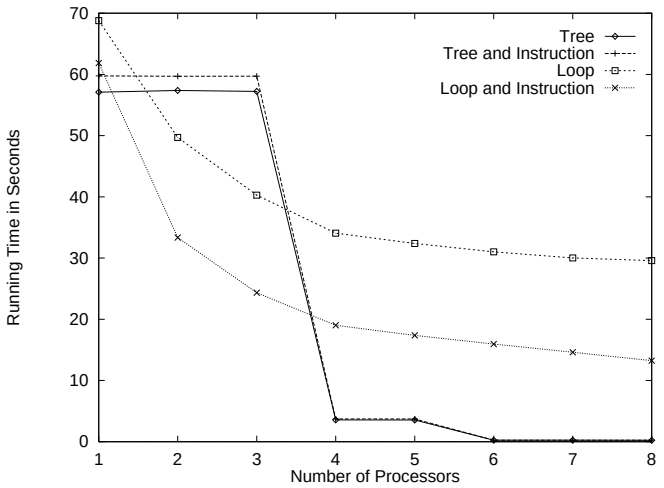

Figure 1: Comparison of Four Parallelizations, Varying Processors

- we may need the best parallelization for a fixed input density and problem as we port the program across machines.

One might assume that for some of these cases, the best parallelization does not vary, making the decision easy. In fact, we show in a detailed study of this application [Crowl *et al.*, 1993] that in *none* of these cases is the best parallelization fixed; the choice of which parallelization of subgraph isomorphism performs best varies in all cases by significant margins. Other researchers have also noted that the best parallelization for a given problem can vary depending on the input, machine, or problem [Eager and Zahorjan, 1993; Rao and Kumar, 1989; Subhlok *et al.*, 1993].

Examples of these effects are shown in Table 1. This table shows the best running time in seconds for loop- and tree-parallel implementations, while varying one component of the environment. The underlined entries in the table are the better-performing executions. The table shows that: 1) when we seek 128 solutions in a sparse solution space, some machines prefer loop parallelism, while others prefer tree parallelism; 2) when we seek 1 solution on the Butterfly, as the density of the solution space varies from 10^{-5} (dense) to 10^{-35} (sparse) and finally to an empty solution space, the best parallelization varies; and 3) in searching a dense solution space on the Symmetry, loop parallelism is preferable when seeking 1 or 128 solutions, but tree parallelism is preferable when seeking 256 solutions.

An example of how the best parallelization varies as we vary the number of processors used is shown in Figure 1. This figure shows the running time of four parallelizations (tree, tree combined with instruction, loop, and loop combined with instruction) on the Silicon Graphics Iris. It shows that for some numbers of processors (1, and 4-8), tree parallelism outperforms the others; for other ranges of processors (2-3) loop and instruction parallelism outperforms the others; and neither loop parallelism nor tree combined with instruction parallelism ever perform better than all the others. It also shows that adding instruction-level parallelism to loop parallelism improves it significantly, while adding instruction-level parallelism to tree parallelism has a slightly negative effect on performance.

These effects can be explained in terms of load imbalance, communication costs, processor speed, pure computation costs, and speculative computation (parallelism used to discover alternative, cheaper solutions rather than speeding the discovery of a particular solution). Detailed explanations are presented in [Crowl *et al.*, 1993]; here we note only that the insight necessary to explain the relative performance of these implementations can be derived almost exclusively from high-level notions such as load imbalance and communication costs, without making detailed measurements of each execution.

We will call the four dimensions of the domain of an implementation's performance function (input density, problem, number of processors, and machine) *environment variables.* An *execution* consists of running an implementation for a fixed set of environment variables. Additional environment variables exist for other programs; for example, in many problems the *size* of the input data, rather than its internal structure (e.g., solution space density), is the most important factor in application performance.

2.2 Performance Tuning By Selecting Implementations

Most often, parallel performance evaluation and tuning is concerned with the performance of a single execution of an application. In contrast to tuning an application by focusing on specific code segments to improve, our study of subgraph isomorphism indicates that there is a prior, more general performance tuning problem: under what circumstances is each implementation best? The previous section gave typical examples of these performance tuning problems: varying input density or size, problem, number of processors, or machine. Each of these problems corresponds to searching a subspace of the environment space. The previous section also showed that no dimension of the environment is trivial. To be able to solve all of these performance tuning problems by selecting the proper implementation in each case requires knowing the relative performance of each implementation over the entire, high-dimensional environment. We call this discovery of the relative performance of all implementations the *best implementation* problem. We view the best implementation problem as a necessary precursor to the traditional style of performance tuning that focuses on improving individual (often serial) code segments.

One way to solve the best implementation problem is to measure the performance of all available implementations over the entire environment space. Unfortunately this solution is exponential in the number of environmental dimensions. Thus, even for simple programs, it is a huge task. In our study of subgraph isomorphism we measured over 37,000 executions,

while solving only a subset of the best implementation problem. We need some way to reduce the complexity of the problem.

In many cases we can solve the best implementation problem using methods that are only linear in the number of dimensions. The key insight lies in the explanatory power of simple overhead categories like those used to explain the performance of subgraph isomorphism: load imbalance, communication costs, and wasted (speculative) computation. Measuring these overheads directly for the entire environment space is still impractical, but if the categories are chosen properly, modeling them as a separate function of each environment variable is feasible. A small number of measurements for each environment variable will then suffice to parameterize the models, leading to an aggregate model of performance prediction spanning the entire environment space.

Given performance models for each implementation that span the entire environment space, the selection of best implementation is straightforward. When an application is to be ported, or run on a different kind of data set, or run on a different number of processors, the performance models can be quickly reduced to functions of the environment variable(s) of interest. The crossover boundaries at which one implementation outperforms another are then obtained by directly solving the performance functions as simultaneous equations. In the context of a parallel programming environment these performance models would be associated with their implementations, for ready use as implementation-selection decisions arise.

This approach is in sharp contrast to traditional performance debugging, which is usually an "iterative task alternating between measuring and modifying the performance of successive computation prototypes." [Lehr *et al.*, 1989] Our approach is complementary to "measure-modify" performance debugging, but our problem is one that requires modeling to play a more prominent role. If programmers are to explore a variety of vastly different implementations before beginning to fine-tune the implementation with the best high-level structure for a given environment, we must find a way to minimize the role of measurement, and exploit the fact that many differences in high-level program structure are amenable to analysis.

2.3 Lost Cycles

In order for models based on simple overhead categories to be useful, all categories must be measured using the same metric. We call this metric *lost cycles.* Lost cycles are simply aggregate seconds of parallel overhead, attributed to various categories. Lost cycles is an important notion because it allows us to quantitatively study tradeoffs among effects such as serial fraction, synchronization, communication, and contention that are often measured and modeled in incompatible ways. The portion of the execution time not consumed in lost cycles we refer to as *pure computation.*

The core of our approach is the proper selection of categories. To be successful, lost cycles must be allocated to a set of categories that together meet three criteria:

1. **Completeness.** The categories must capture *all* sources of overhead.
2. **Orthogonality.** The categories must be mutually exclusive.

3. **Meaning.** The categories must correspond to states of the execution that are meaningful for analysis.

Although often overlooked, completeness is a crucial criterion. Completeness ensures that we do not ignore any overheads as we vary environment variables, *regardless of whether we expect them to be dominant.* Completeness is rarely achieved in performance measurement tools; many tools concentrate on specific performance metrics such as cache miss rates, message traffic, and execution profiles. Each of these tools is useful as long as the tool's metric corresponds to a dominant source of overhead. However, predicting which category of overhead is dominant in all cases is a very difficult task — it is not uncommon that performance is dominated by unexpected effects.

Completeness is also rarely achieved because it requires measurement of effects that occur at different levels — application (e.g., load imbalance) and hardware (e.g., resource contention). A system that attempts to measure lost cycles therefore must be able to instrument the application, as well as have access to hardware performance data.

Completeness and orthogonality together ensure that we can correctly measure lost cycles, and indirectly, pure computation. Completeness ensures that measurements of pure computation are accurate. Orthogonality ensures that we can subtract overheads from running time to calculate pure computation in the natural way.

Meaningfulness of categories serves a rather different purpose, and makes the choice of categories somewhat more difficult. Categories must be meaningful so that they are likely to be amenable to simple analysis. It is certainly possible to define a set of overhead categories that are complete and orthogonal, but without meaning for analytic purposes — the simplest such set would contain one category for all lost cycles. Thus the challenge in defining a category set is in dividing overheads finely enough that they can be analyzed simply, but not so finely as to present problems in measurement, or in verifying completeness and orthogonality.

Meaningfulness of categories also allows the programmer to relate the measurements made to the program being studied. Categories that are amenable to analysis tend to correspond in simple ways to the structure of the program. As a result, measurements of meaningful categories can provide significant performance tuning assistance even apart from their use in analytic models.

2.4 Related Work

The majority of the tools and metrics devised for performance evaluation and tuning reflect their orientation on the measure-modify paradigm [Callahan *et al.*, 1990; Cybenko *et al.*, 1991; Davis and Hennessy, 1988; Dongarra *et al.*, 1990; Goldberg and Hennessy, 1993; Heath and Etheridge, 1991; Kohn and Williams, 1993; So *et al.*, 1987]. Such tools are very useful in application fine-tuning, but usually do not provide completeness (i.e., they don't measure *all* sources of overhead in the execution). As a result, they are limited to cases in which the principal overheads are known in advance.

A number of researchers in the parallel performance evaluation and tuning community have focused on measurement of multiple parallel overheads. In particular the PEM system

has developed a taxonomy of parallel overheads similar to ours [Burkhart and Millen, 1989], and Quartz and MemSpy together can measure the overhead categories we use [Anderson and Lazowska, 1990; Martonosi *et al.*, 1992]. However, since these (and similar tool sets) are not oriented toward performance prediction of alternative implementations, the specific overhead categories they use are not always amenable to easy analysis. In addition, the completeness criterion has not been emphasized in most previous overhead measurement work [Burkhart and Millen, 1989; Møller-Nielsen and Staunstrup, 1987; Tsuei and Vernon, 1990; Vrsalovic *et al.*, 1988].

A common method of modeling of overheads in parallel programs is scalability analysis [Kumar and Gupta, 1991]. Scalability analysis develops analytic, asymptotic models of computation and selected overhead categories as a function of the size of the problem n and the number of processors p. These analyses provide insight into the inherent scalability of a particular application and machine combination. Such analyses are an important part of our approach, but they don't provide enough mechanism on their own to solve the best implementation problem. Most importantly, they are subject to the problem of deciding beforehand which overheads will dominate, which can be error-prone. In addition, they cannot be used directly for performance prediction or for a comparison of alternative implementations in general because of their reliance on asymptotic analysis, and on constants which must be determined experimentally.

In addition to scalability analysis, many other analytic frameworks are based on the notion of parallel overheads [Carmona and Rice, 1991; Eager *et al.*, 1989; Flatt and Kennedy, 1989; Nicol and Willard, 1988]. The analytic portion of our work is compatible with these other frameworks as well as with scalability analysis, so we will use that term to include all analytic frameworks that deal expressly with the modeling of parallel overheads.

Other methods model overheads in highly empirical ways that are difficult to generalize [Abrams *et al.*, 1992; Dimpsey and Iyer, 1991]. These methods develop categories of execution state based solely on program events; as a result it is difficult to predict how time spent in these categories would change with changes in the environment.

Finally, the notion of selecting alternative implementations based on the environment is present in the ISSOS system [Schwan *et al.*, 1988]. The emphasis in that work is on selecting alternative implementations dynamically based on ongoing system monitoring. As a result the dynamic adaptations do not include drastic restructuring of the application, and no guidelines for how the programmer should select among alternatives were developed.

3 Identifying and Measuring Lost Cycles

The lost cycles approach consists of the following steps:

1. programs under study are instrumented transparently;
2. at run time, the programs are measured to determine lost cycles, and those measurements are partitioned into a small number of categories;
3. lost cycles data is accumulated for a range of environment variations (varying the number of processors, size of input, structure of input, or machine characteristics);

4. data for each measurement category is fitted to an appropriate analytic model, a task made simple by the specific categories used; and

5. the resulting category-specific models are aggregated into an overall model of how the program's performance varies depending on the environment.

Lost cycles bridge the gap between scalability analysis and performance evaluation by combining the best of both approaches. By basing analysis on empirically measured overheads, lost cycles provides "hard numbers" for use in scalability models. In addition, because the overhead categories chosen have reasonably well-understood properties, choosing the proper model to fit the data is fairly straightforward. On the other hand, by adding an analytic basis to performance evaluation, we extend performance evaluation to allow its application to cross-execution problems such as the best implementation problem.

3.1 Categories of Lost Cycles

To predict performance, we must be able to predict two quantities: total lost cycles T_o, and pure computation T_c, as functions of all environment variables. Given T_o and T_c, we can predict running time as $T_p = (T_o + T_c)/p$. As long as the parallel algorithm is a parallelization of the best serial algorithm, pure computation is equal to serial running time. In that case, we can calculate efficiency and speedup as

$$E = \frac{1}{1 + \frac{T_o}{T_c}}$$

$$S = p - \frac{T_o}{T_p}.$$

We measure T_o by breaking it down into additive categories. Once we have an accurate measurement for T_o we can obtain T_c: $T_c = pT_p - T_o$. The category set we use in this paper is:

Load Imbalance: processor cycles spent idling, while unfinished parallel work exists.

Insufficient Parallelism: processor cycles spent idling, while no unfinished parallel work exists.

Synchronization Loss: processor cycles spent acquiring a lock, or waiting in a barrier.

Communication Loss: processor cycles spent waiting while data moves through the system.

Resource Contention: processor cycles spent waiting for access to a shared hardware resource.

This category set has proven to satisfy the three criteria (completeness, orthogonality, meaning) for the applications we have studied. Of course, it will need to be expanded to handle a wider range of overheads as it is used in more varied situations. In particular, it does not currently distinguish between synchronization types, measure contention for software resources, or measure operating system and runtime library effects. Each of these extensions appears to be straightforward within the existing framework however.

3.2 A Tool For Analyzing Lost Cycles

Although we performed model selection by hand for the examples in this paper, we anticipate the need for a tool that manages performance data and focuses the user on a selection of appropriate models for each category of lost cycles. Such a tool assists the user in two ways:

1. The analysis tool stores and presents lost cycles data, and provides the ability to quickly explore different models for each variable.

2. The analysis tool guides the user's selection of models for each category and environment variable, using preferences based on our experience in modeling the categories of lost cycles. For example, when varying data set size, the model for communication loss defaults to a linear function of data size. Likewise, when varying the number of processors, the model for insufficient parallelism loss defaults to a linear function of number of processors.

Analyzing lost cycles is done independently for each environment variable. For each variable, a small number of data points are taken. Generally, the programmer should select a number of data points that will provide enough data to get accurate models of all of the overheads. The simple models we use rarely require more than two data points to parameterize, but additional data points may be taken for two reasons: to give insight into the type of model to use, and to eliminate noise in the data and get a more accurate model fit (e.g., using a least squares fit).

An example we will present in Section 4 is somewhat extreme because it only uses two data points for the complete analysis. However, we don't expect that a large number of data points are necessary in any case, and the number needed doesn't grow faster than the number of environment variables being considered.

In many cases it will be necessary to perform simple analysis on the program to ascertain the proper models. The examples in Section 4 exhibit several cases where this occurs. We expect that in most cases this analysis will be straightforward.

3.3 A Tool For Measuring Lost Cycles

In previous work [Crovella and LeBlanc, 1993] we showed that basing measurement on logical expressions that recognize lost cycles is a particularly useful approach. We call these expressions *performance predicates*. The use of performance predicates to specify categories of lost cycles makes program instrumentation straightforward, and allows *predicate profiles* to be constructed based on user demands. For example, using predicate profiling, the user can ask for a breakdown of lost cycles by processor number, task, or procedure. The current implementation uses predicate profiling of event logs, rather than the runtime profiling used in our earlier work.

Our current implementation of the lost cycles measurement tool measures Fortran programs running on the Kendall Square KSR1, and takes the form of a library linked into the executable code. The KSR Fortran runtime system can log events such as the start and end

of individual loop iterations, which we use for calculating load imbalance. Additional calls to our library routines are inserted at the start and end of parallel loops, parallel tasks, and synchronization operations. The inserted library calls are quite simple and could easily be added by a source-to-source preprocessor.

The KSR1 [Research, 1991] is a two-level ring architecture in which all memory is managed as a cache, which is organized in two levels on each node. Thus, inter-node communication occurs only as the result of misses in the secondary cache. Dedicated hardware monitors the state of buses between the processor and the second-level cache. This performance monitor counts the number of secondary cache misses, the time taken to service secondary cache misses, and the number of cache lines that passed through the higher-level ring before arrival. Based on this data, we can calculate the amount of communication performed in an execution and the amount of resource contention that occurred.

Communication loss is measured as a simple product of the number of cache misses and the ideal time to perform the cache line transfers. Resource contention (contention for the ring interconnect and for remote memories) is measured as in [Tsuei and Vernon, 1990] — that is, the ideal time to perform the communication operations is compared to the actual elapsed time. Since the KSR1 hardware monitors both the number of cache lines transferred and the elapsed time waiting for cache lines, this calculation is straightforward. Although the performance monitoring hardware on the KSR is rather unique, something comparable may be required for other cache-coherent architectures. On simpler architectures, such as a message-passing system, the performance monitoring capabilities of the DEC Alpha [Corporation, 1992] should be sufficient to gather the same information.

4 Cross-Execution Performance Evaluation

This section presents three examples of the use of lost cycles analysis. First we show how to construct accurate analytic models of program scalability based on a small number of measurements. Then we present two examples that illustrate how lost cycles analysis can help solve the problem of selecting the best parallelization for a program.

4.1 Modelling the Performance of 2D FFT

The ability to capture the expected performance of a program based on a small number of measurements is critical to managing the problem of understanding and selecting among differing implementations. Measuring and debugging program performance without gathering large amounts of data is an important capability in its own right, and is the subject of much current effort [Ball and Larus, 1992; Hollingsworth and Miller, 1993; Miller and Choi, 1988; Netzer and Miller, 1992]. The results in this section show that in addition to its use in solving the best implementation problem, lost cycles modelling is a convenient way of capturing large amounts of performance data, requiring minimal measurement effort and little storage.

In this section we study the scalability of a two-dimensional discrete Fourier transform program (2D FFT). This program is simple enough to demonstrate our method, while exhibiting a wide variety of performance effects. Our implementation of the program consists

Category	Abbrev.	Model	
		Varying n	Varying p
Pure Computation	PC	$n^2 \log(n)$	1
Load Imbalance	LI	$n \log(n)$	$p\sqrt{p}$
Insufficient Parallelism	IP	1	p
Synchronization Loss	SL	0	0
Communication Loss	CL	n^2	p
Resource Contention	RC	n^2	$p, p > \theta$

Table 2: Models of Overhead, 2D FFT, as a Function of n and p

of a number of iterations in which all processors first participate in 1D FFTs on columns of the input matrix, then transpose the matrix in parallel and perform 1D FFTs on the rows of the matrix. Each iteration of the program consists of 5 parallel loops: one to initialize the matrix, one to perform the column-wise FFTs, two to transpose the matrix (using an intermediate matrix), and one to perform the row-wise FFTs. Our study will result in the ability to quickly characterize the program's execution time and efficiency over a range of dataset sizes (from 32×32 points to 1024×1024 points) and numbers of processors (2 to 26), while requiring that we measure only a small number of executions.

In this example we will:

1. Measure the program's lost cycles for two cases: the highest-overhead case, and the lowest-overhead case.

2. Select appropriate simple models for each category of lost cycles and for pure computation, as separate functions of varying data size and varying number of processors.

3. Use the measurements made in step 1 to parameterize the models selected in step 2, yielding predictions for running time over the entire range of data sizes and numbers of processors.

We selected two data points for measurement because by measuring an execution with high relative overhead we can get an accurate estimate of true overhead, and by measuring an execution with low relative overhead we can get an accurate estimate of pure computation. Rules of scalability analysis guide us in selecting the data points for measurement: in a parallel system, overheads tend to grow with increasing processors and decrease with increasing data size. These observations suggest that we should capture lost cycles for an execution with maximum processors and minimum data (highest overhead) and for an execution with minimum processors and maximum data (lowest overhead).

The simple models we chose to describe each overhead are listed in Table 2, as separate functions of n (the length of a side of the input matrix) and p (number of processors). Each model has an implicitly associated constant; the purpose of our lost cycles measurements in step 1 is to discover the constants. Each of these models is a simple, initial approximation to reality. Better models for each are possible, but not necessary in this context since

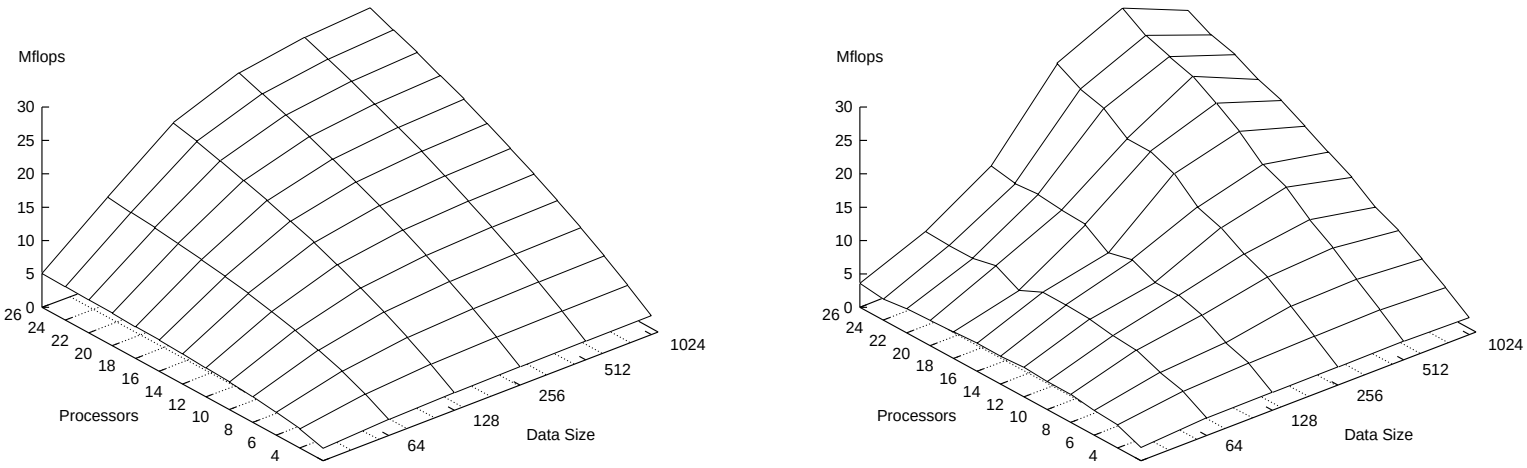

Figure 2: Predicted and Actual Performance of 2D FFT

they trade increasing accuracy for increasing measurement cost, and decreasing analytic tractability.

Considering first the models for varying n, the model for pure computation is based on simple algorithmic analysis of 2D FFT. The model for load imbalance is based on the length of an iteration of the program's parallel loops. There are no synchronization operations in the program, so we expect no synchronization loss. The model for insufficient parallelism is based on the the portion of the code that runs serially, which has no data size dependencies. The model for communication loss is based on the total amount of data used. Finally, the model for resource contention is based on the expectation that resource contention will be proportional to data size.

In choosing models of overhead as we vary the number of processors, we can rely on the large body of work in the literature to provide likely candidate models. Most of the models we use are straightforward: pure computation does not vary as we vary processors, insufficient parallelism obeys Amdahl's Law [Amdahl, 1967], and synchronization loss is zero.

Load imbalance can arise in two ways: variation in the running time of each loop iteration, and unequal numbers of loop iterations handled by different processors. If variation in running time of iterations is random, the time taken by the longest iteration can be modeled using order statistics (e.g., [Hummel *et al.*, 1992]) and predicted to grow proportionally to $\sqrt{p}$. Communication loss can be difficult to model, but for this simple application is proportional to p. Finally, resource contention can be expected to grow linearly once the number of processors passes a threshold value.

Using the lost cycles measurements from the two executions we then parameterize the six models (*PC, LI, IP, SL, CL,* and *RC*). Using the basic identity $T_p = (T_c + T_o)/p$, we

construct the performance model for the implementation as:

$$T_p(n,p) = \frac{PC(n,p) + LI(n,p) + IP(n,p) + SL(n,p) + CL(n,p) + RC(n,p)}{p}$$

The results for the 2D FFT program are shown in Figure 2. These plots show the performance of the application measured in Mflops, as a function of both number of processors and of dataset size. The left hand plot shows the predictions of our model for 78 data points, that is, all points within the range of processors and data set sizes we set out to model. The right hand plot shows the actual measured performance of the application on the KSR1 for those same 78 data points.

As can be seen, the model is an idealized but reasonably accurate approximation to actual performance. In fact, the average relative error of the model with respect to the actual performance, over all 78 points, is only 12.5%. For comparison, the average relative error of a simple linear interpolation based on a least squares fit of the four "corner" points is over 750%. Thus both the overall shape of the predicted performance curve and its actual values are sufficiently accurate to allow it to be used in studying tradeoffs against an alternative implementation, which we will do in the next section.

4.2 Task Parallel vs. Data Parallel 2D FFT

The 2D FFT program we used in the previous section has an alternative implementation that uses task parallelism as well as data parallelism. In this implementation, processors are segregated into two groups using tasking directives. One group initializes the matrix and performs data-parallel row-wise 1D FFTs, while the other group transposes the matrix and performs data-parallel column-wise 1D FFTs. The two tasks are pipelined so that each one is kept busy working on separate matrices.

The performance of these two implementations on the iWarp was studied in [Subhlok *et al.*, 1993]. On that machine, the authors discovered that as data set sizes are varied past a certain threshold, the choice of which implementation is best changes. For small data sets ($n \leq 128$) the parallel tasking implementation outperformed the pure data parallel implementation. For large data set sizes ($n \geq 256$), the purely data parallel implementation outperformed the parallel tasking implementation. The principal reason for this effect is that in the parallel task version, communication between tasks must pass through a single channel of the iWarp network, while purely data parallel communication can take place along multiple channels. For small data sizes, the larger problem granularity of parallel tasking leads to better performance, but as problem sizes increase, intertask communication becomes a bottleneck.

It is interesting to ask whether a similar effect would be observed when this application is run on the KSR1, a machine with a significantly different architecture. Unfortunately, the data from the iWarp cannot help us decide which executions to measure, since the machines are so different. Thus we immediately run into the best implementation problem: perhaps there is a crossover between implementations in some section of the environment space (here, n and p), but finding the crossover would require measurements over the entire space.

Category	Data Parallel	Task Parallel
Pure Computation	$\frac{n^2 \log(n)}{3550}$	$\frac{n^2 \log(n)}{3350}$
Load Imbalance	$\frac{n \log(n)}{63.0}$	$\frac{n \log(n)}{81.9}$
Insufficient Parallelism	3.36	$\frac{n^2}{992}$
Synchronization Loss	0	$\frac{n^2}{31600}$
Communication Loss	$\frac{n^2}{14900}$	$\frac{n^2}{12200}$
Resource Contention	$\frac{n^2}{20100}$	$\frac{n^2}{35600}$

Table 3: Performance Models for Data Parallel and Task Parallel 2D FFT

To answer this question using the lost cycles approach, we only need to construct a model for the application. The simplest approach in this case is to 1) decide whether the category models used for the pure data parallel implementation follow the same functions; and 2) determine new constants for the overhead functions. To do this we measured 6 points varying the data set size, (to explore the functions of n) and 6 points varying the number of processors (to explore the functions of p).

The results are shown in Table 3. The table shows the functions and the associated constants for the n variable, since the models did not differ significantly in the p dimension.[2] These functions immediately answer our questions about these two implementations. First of all, resource contention in the task parallel implementation is significantly less than in the data parallel implementation, indicating that the channel bottleneck effects observed on the iWarp will not be present on the KSR1. This conclusion is reasonable, since intra-ring communication costs are insensitive to source and destination on the KSR1. In fact, we see that resource contention is only about half as great in the task parallel version, since in the pure data parallel version, all processors are simultaneously requesting *and* providing data during the matrix transpose, while in the parallel task version, half the processors request data and the other half provide it.

The second observation is that on this machine, the task parallel implementation will always perform more poorly than the pure data parallel implementation. Synchronization loss and insufficient parallelism are functions of n^2 in the task parallel implementation. The reason for this change from constant values to functions of n^2 when the implementation is changed can be seen in observing that synchronization loss is now equal to about a third of the communication loss. In fact, in this implementation, the two tasks do *not* incur equal overhead. The task that transposes the matrix incurs more overhead because it must traverse the source matrix across cache lines, destroying locality. Thus each loop iteration for the transposing task takes slightly longer than an iteration of the initializing

[2] We hold p constant at its maximum value (26) in these formulae.

Category	Processors						
	1	2	3	4	5	6	7
Wasted Speculation	0.00	50.1	103	10.7	14.2	1.25	1.44
Pure Computation	51.4	50.1	51.7	3.56	3.52	0.227	0.214

Table 4: Seconds of Pure Computation and Wasted Speculation in Subgraph Isomorphism

task; a pair of spinlocks prevents either task from overtaking the other. As a result of this synchronization loss in the main thread of the initializing task, the other threads in its group must wait without work, incurring lost cycles due to insufficient parallelism. This insufficient parallelism has a particularly small constant in the denominator and hence dominates the small improvements in resource contention and load imbalance generated by task parallelism.[3]

Thus we have quickly answered the question of whether two implementations, known to have a performance tradeoff on at least one architecture, have a similar performance tradeoff on the architecture of interest to us. To do this, we only needed to measure a small number of data points in each of the 2 environmental dimensions, and compare the resulting lost cycles models.

4.3 Subgraph Isomorphism

We now return to the example of subgraph isomorphism. Referring to Figure 1, we would like to understand the differences among the four parallelizations of subgraph isomorphism (tree, tree plus instruction, loop, and loop plus instruction) as a function of p. Since we are only studying one dimension of the environment in this case, we can measure the implementations over the entire range and use the resulting models to explain their relative performance.

In order to achieve completeness for this application, we need to include measurement of cycles lost in wasted computation (due to fruitless speculation). We will define wasted computation in this case as the processor cycles spent searching a subtree under the root in which no solutions are found. Modeling this category is difficult, but since we are only searching a single dimension, we can simply measure lost cycles due to wasted computation for the points of interest.

First of all, we show the wasted speculation and pure computation data for the tree parallel case in Table 4. This table immediately explains why the tree parallel versions outperform the loop parallel versions when $p > 3$, yet do worse than loop parallel versions when $1 < p < 4$. The pure computation required to solve the problem changes drastically with increasing p because these executions of the program are searching for only one solution — the first processor to find a solution ends the computation. Clearly, the subtrees searched by processors 2 and 3 do not yield the solution, since wasted computation increases in steps of 50 seconds (the time spent by processor 1 in finding a solution). Processor 4 finds a

[3]Presumably these effects were not present on the iWarp because of message passing optimizations.

Category	Implementation	
	Loop	Loop + Instr.
Pure Computation	54.7	55.9
Load Imbalance	$\frac{p\sqrt{p}}{.750}$	$\frac{p\sqrt{p}}{1.46}$
Insufficient Parallelism	$\frac{p}{.496}$	$\frac{p}{2.08}$
Synchronization Loss	0	0
Communication Loss	$\frac{p}{.110}$	$\frac{p}{.233}$

Table 5: Lost Cycles Models for Two Implementations of Subgraph Isomorphism

solution in its subtree much sooner than the others however, and the effect is repeated again by processor 6. The data in this table shows that even when overhead categories are difficult to model, the raw lost cycles data can be informative in ways that are difficult for tools that do not provide completeness.

Next, we consider why loop and instruction parallelism together outperform loop parallelism alone. Since there is no speculative computation in the loop parallel executions, this is done most easily by considering the lost cycles models, which are shown in Table 5.[4]

We might expect that the benefit of adding instruction parallelism to the implementation would be in decreased pure computation (since instruction parallelism shows up as decreased pure computation). However, the lost cycles models show that in fact, pure computation is relatively unchanged between the two implementations. The model shows that the additional cost of packing and unpacking data counterbalances the pure computational decrease gained in parallel set operations, actual increasing pure computation slightly.

In fact, the gains from adding instruction parallelism come from a less expected direction. First of all, by packing datasets, the overall data being transferred decreases, decreasing communication loss by a factor of 2. Secondly, load imbalance within loops and insufficient parallelism are decreased since these overheads both tend to increase as communication increases.

The lost cycles models in Table 5 also explain why the loop parallel implementation benefits from the addition of instruction parallelism, while the tree parallel implementation actually suffers slightly from the same addition. Since the tree parallel version searches separate subtrees in parallel, it has essentially no communication loss (this effect can be observed from the lost cycles data as well). The absence of communication loss in the tree parallel implementation means that it cannot reap the benefits of instruction parallelism, whether directly in decreased communication, or indirectly through decreased load imbalance and insufficient parallelism. Instead the tree parallel version only pays the (small) price of instruction parallelism, a fact reflected in the performance data shown in Figure 1.

[4]These data are derived from an implementation of the LC tool on the Silicon Graphics Iris that did not allow the measurement of resource contention.

In this example we have gained a number of insights into the relationship between the particular implementation of the subgraph isomorphism program and its corresponding performance. These insights were gained from measurements of only 12 data points, showing the power of lost cycles analysis to provide tuning guidance — especially since these insights were not evident from our original collection of over 37,000 data points.

5 Conclusion

We've argued that an important kind of performance tuning consists of selecting among alternative implementations of a parallel program, and that this problem suffers from combinatorial explosion as one increases the number of environment variables considered during tuning. To address this problem, we have presented a technique that combines performance measurement with analytic modeling. Our technique is based on measuring overhead categories that meet the three criteria of completeness, orthogonality, and meaning; we have shown how each of these criteria is necessary in order to reliably solve the best implementation problem.

Our examples show that modeling lost cycles has significant flexibility. Our first example showed how modeling lost cycles can capture a great deal of performance data using only a small number of measurements. Our second example studied the best implementation problem for two implementations and used lost cycles modeling to expose differences in the asymptotic behavior of the two applications. Finally our last example showed that the raw data output from lost cycles measurements can be informative in its own right, and that comparison of lost cycles models can expose subtle interactions among program components.

We are continuing to develop the lost cycles approach in two directions. We intend to extend the set of categories measurable so that lost cycles analysis will be complete for a wider range of applications. We also expect to develop our data analysis capabilities to guide the user to the best choice of model for each category of lost cycles.

Acknowledgements

Jaspal Subhlok of CMU provided the iWarp Fortran code for the 2D FFT problem. Our thanks to the Cornell Theory Center and staff, for their help and the use of their KSR1.

References

[Abrams *et al.*, 1992] Marc Abrams, Naganand Doraswamy, and Anup Mather, "Chitra: Visual Analysis of Parallel and Distributed Programs in the Time, Event, and Frequency Domains," *Journal of Parallel and Distributed Computing*, 3(6):672–685, November 1992.

[Alverson and Notkin, 1992] Gail A. Alverson and David Notkin, "Abstracting Data-Representation and Partitioning-Scheduling in Parallel Programs," In N. Suzuki, editor, *Shared Memory Multiprocessing*, pages 315–338. MIT Press, 1992.

[Amdahl, 1967] G. M. Amdahl, "The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," In *AFIPS Conference Proceedings*, volume 20, pages 483–485. AFIPS Press, Reston, Va., April 1967.

[Anderson and Lazowska, 1990] Thomas E. Anderson and Edward D. Lazowska, "Quartz: A Tool for Tuning Parallel Program Performance," In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 115–125, May 1990.

[Ball and Larus, 1992] Thomas Ball and James R. Larus, "Optimally Profiling and Tracing Programs," In *Conference Record of the Nineteenth POPL*, Albuquerque, NM, 19-22 January 1992.

[Burkhart and Millen, 1989] Helmar Burkhart and Roland Millen, "Performance-Measurement Tools in a Multiprocessor Environment," *IEEE Transactions on Computers*, 38(5):725–737, May 1989.

[Callahan *et al.*, 1990] David Callahan, Ken Kennedy, and Allan Porterfield, "Analyzing and Visualizing Performance of Memory Hierarchies," In *Performance Instrumentation and Visualization*, pages 1–26. ACM Press, 1990.

[Carmona and Rice, 1991] Edward A. Carmona and Michael D. Rice, "Modeling the Serial and Parallel Fractions of a Parallel Algorithm," *Journal of Parallel and Distributed Computing*, 13:286–298, 1991.

[Coffin, 1990] Michael H. Coffin, *Par: An approach to architecture-independent parallel programming*, PhD thesis, University of Arizona, August 1990.

[Corporation, 1992] Digital Equipment Corporation, "DECChip 21064-AA RISC Microprocessor Preliminary Data Sheet," Digital Equipment Corporation, Maynard, MA, 1992.

[Crovella and LeBlanc, 1993] Mark E. Crovella and Thomas J. LeBlanc, "Performance Debugging using Parallel Performance Predicates," In *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 140–150, May 1993.

[Crowl and LeBlanc, 1991] L. A. Crowl and T. J. LeBlanc, "Architectural Adaptability in Parallel Programming via Control Abstraction," Technical Report 359, University of Rochester Computer Science Department, Rochester, New York, 14627, January 1991.

[Crowl *et al.*, 1993] Lawrence A. Crowl, Mark E. Crovella, Thomas J. LeBlanc, and Michael L. Scott, "Beyond Data Parallelism: The Advantages of Multiple Parallelizations in Combinatorial Search," Technical Report 451, Department of Computer Science, University of Rochester, April 1993.

[Cybenko *et al.*, 1991] G. Cybenko, J. Bruner, S. Ho, and S. Sharma, "Parallel Computing and the Perfect Benchmarks," In *Intl. Symposium on Supercomputing*, Fukwoka, Japan, November 1991.

[Davis and Hennessy, 1988] Helen Davis and John Hennessy, "Characterizing the Synchronization Behavior of Parallel Programs," In *Proceedings of the First PPEALS*, pages 198–211, July 1988.

[Dimpsey and Iyer, 1991] R. T. Dimpsey and R. K. Iyer, "Performance Prediction and Tuning on a Multiprocessor," In *Proceedings of the Eighteenth ISCA*, pages 190–199, Toronto, Canada, May 1991.

[Dongarra *et al.*, 1990] J. Dongarra, O. Brewer, J. A. Kohl, and S. Fineberg, "A Tool to Aid in the Design, Implementation, and Understanding of Matrix Algorithms for Parallel Processors," *Journal of Parallel and Distributed Computing*, 9(2):185–202, June 1990.

[Eager and Zahorjan, 1993] Derek L. Eager and John Zahorjan, "Chores: Enhanced Run-Time Support for Shared-Memory Parallel Computing," *ACM Transactions on Computer Systems*, 11:1–32, February 1993.

[Eager *et al.*, 1989] Derek L. Eager, John Zahorjan, and Edward D. Lazowska, "Speedup Versus Efficiency in Parallel Systems," *IEEE Transactions on Computers*, 38(3):408–423, 1989.

[Flatt and Kennedy, 1989] Horace P. Flatt and Ken Kennedy, "Performance of Parallel Processors," *Parallel Computing*, 12:1–20, 1989.

[Goldberg and Hennessy, 1993] Aaron J. Goldberg and John L. Hennessy, "Mtool: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications," *IEEE Transactions on Parallel and Distributed Systems*, 4(1):28–40, January 1993.

[Halstead, 1985] Robert H. Halstead, Jr., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[Heath and Etheridge, 1991] Michael T. Heath and Jennifer A. Etheridge, "Visualizing the Performance of Parallel Programs," *IEEE Software*, 8(5):29–39, September 1991.

[Hollingsworth and Miller, 1993] Jeffrey K. Hollingsworth and Barton P. Miller, "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems," In *7th ACM International Conference on Supercomputing*, July 1993.

[Hummel *et al.*, 1992] Susan Flynn Hummel, Edith Schonberg, and Lawrence E. Flynn, "Factoring: A Method for Scheduling Parallel Loops," *Communications of the ACM*, 35(8):90–101, August 1992.

[Kohn and Williams, 1993] James Kohn and Winifred Williams, "ATExpert," *Journal of Parallel and Distributed Computing*, 14, May 1993.

[Kumar and Gupta, 1991] Vipin Kumar and Anshul Gupta, "Analyzing Scalability of Parallel Algorithms and Architectures," Technical report, TR-91-18, Computer Science Department, University of Minnesota, June 1991, A short version of the paper appears in the Proceedings of the 1991 International Conference on Supercomputing, Germany, and as an invited paper in the Proc. of 29th Annual Allerton Conference on Communication, Control and Computing, Urbana, IL, October 1991.

[Lehr *et al.*, 1989] Ted Lehr, Zary Segall, Dalibor Vrsalovic, Eddie Caplan, Alan Chung, and Charles Fineman, "Visualizing Performance Debugging," *IEEE Computer*, pages 38–51, October 1989.

[Martonosi *et al.*, 1992] Margaret Martonosi, Anoop Gupta, and Thomas Anderson, "MemSpy: Analyzing Memory System Bottlenecks in Programs," In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 1–12, June 1992.

[Miller and Choi, 1988] Barton P. Miller and Jong-Deok Choi, "A Mechanism for Efficient Debugging of Parallel Programs," In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 135–144, June 1988.

[Møller-Nielsen and Staunstrup, 1987] Peter Møller-Nielsen and Jørgen Staunstrup, "Problem-heap: A Paradigm for Multiprocessor Algorithms," *Parallel Computing*, 4:64–74, 1987.

[Netzer and Miller, 1992] Robert H. B. Netzer and Barton P. Miller, "Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs," In *Proceedings Supercomputing '92*, pages 502–511, Minn., MN, November 1992. IEEE.

[Nicol and Willard, 1988] David M. Nicol and Frank H. Willard, "Problem Size, Parallel Architectures, and Optimal Speedup," *Journal of Parallel and Distributed Computing*, 5:404–420, 1988.

[Rao and Kumar, 1989] V. Nageshwara Rao and Vipin Kumar, "Parallel Depth-First Search," *International Journal of Parallel Processing*, 16(6), 1989.

[Research, 1991] Kendall Square Research, "KSR1 Principles of Operation," Kendall Square Research, 170 Tracer Lane, Waltham MA, 15 October 1991.

[Schwan *et al.*, 1988] Karsten Schwan, Rajiv Amnath, Sridhar Vasudevan, and David Ogle, "A Language and System for the Construction and Tuning of Parallel Programs," *IEEE Transactions on Software Engineering*, 14(4):455–471, April 1988.

[So *et al.*, 1987] K. So, A.S. Bolmarcich, F. Darema, and V.A. Norton, "A Speedup Analyzer for Parallel Programs," In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 653–662, August 1987.

[Subhlok *et al.*, 1993] J. Subhlok, J. M. Stichnoth, D. R. O'Hallaron, and T. Gross, "Programming Task and Data Parallelism on a Multicomputer," In *Proceedings of the Fourth PPOPP*, San Diego, CA, 20-22 May 1993.

[Tsuei and Vernon, 1990] Thin-Fong Tsuei and Mary K. Vernon, "Diagnosing Parallel Program Speedup Limitations Using Resource Contention Models," In *Proceedings of the 1990 International Conference on Parallel Processing*, pages I–185 – I–189. The Pennsylvania State University Press, August 1990.

[Vrsalovic *et al.*, 1988] Dalibor Vrsalovic, Daniel P. Siewiorek, Zary Z. Segal, and Edward F. Gehringer, "Performance Prediction and Calibration for a Class of Multiprocessor Systems," *IEEE Transactions on Computers*, 37(11):1353–1365, November 1988.