Two Computer Systems Paradoxes: Serialize-to-Parallelize, and Queuing Concurrent-Writes

Rimon Orni*and Uzi Vishkin†

September 17, 1995

Abstract

We present and examine the following *Serialize-to-Parallelize Paradox*: suppose a programmer has a parallel algorithm in mind; the programmer must serialize the algorithm, and is actually trained to suppress its parallelism, while writing code; later, however, compilation and runtime techniques are used to reverse the results of this serialization effort and extract as much parallelism as possible. This work actually provides examples where parallel or parallel-style code enables extracting more parallelism than standard serial code.

The "arbitrary concurrent-write" convention is useful in parallel algorithms and programs and appears to be not too difficult to implement in hardware for serial machines. Still, typically concurrent-writes to the same memory location in a program are implemented by queuing the write operations, thus requiring time linear in the number of writes. We call this the *Queuing Concurrent-Writes Paradox.*

Assuming that providing useful, easy-to-program programming paradigms to improve the overall effectiveness of computer systems is of interest, this work is a modest example for applying such software-driven considerations to computer architecture issues. This work may be the first to relate parallel algorithms and parallel programming with the technology of instruction level parallelism.

1 Introduction

Computers were first designed as serial machines, and accordingly, a serial programmer's model has been used for algorithms and programming languages. This programmer's model, as taught in standard computer science and engineering curricula and supported by commercial machines, has essentially remained unchanged over the last few decades.

---

*Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742;

†Institute for Advanced Computer Studies, and Department of Electrical Engineering, University of Maryland, College Park, MD 20742; and Department of Computer Science, Tel Aviv University, Tel Aviv, Israel; Partially supported by NSF grants CCR-9416890 and CCR-9111348.

Overlapping the execution of a growing number of instructions which are independent of one another, or Instruction Level Parallelism (ILP), has become a trend in uniprocessor design. Examples include pipelining and multiscalar execution. ILP has been based on the premise that some parallelism exists in serial codes and indeed a variety of compiler and hardware techniques have been proposed in order to extract this parallelism. Much effort has been devoted to measuring the amount of parallelism which can be automatically extracted from programs by these techniques [17, 5, 19, 26, 2]. These studies suggest that the amount of parallelism which can be thus extracted is limited, even when we invest much hardware and software in the effort, and is likely to be insufficient as the reliance on ILP increases.

The technologies of parallel algorithms and programming and of parallel computer systems have been viewed by this "main stream" activity as separate, having no direct relevance towards the commercial serial machines. This paper offers some modest insights that question this separation. We hope that it will stimulate debate on this aspect.

Before stating the two main points of the paper, we present a concise form of our lead example. In the introductory sections this example will be used for concreteness, while later a more detailed form of it will be used as a test algorithm.

Lead Example Breadth-first search (BFS) is an elementary graph algorithm. Beginning from some vertex s (which forms layer number 0) BFS visits all vertices of the graph adjacent to s (which form layer number 1), then all the yet unvisited vertices which are adjacent to vertices of layer number 1 (forming layer number 2), and so on. In most standard textbooks on algorithms and data structures such a verbal description is followed by the following code-like description.

```
Q := s
layer[s] := 0
while Q ≠ ∅
      u := dequeue( Q )        (the head of the queue is removed and put into u)
      for each v ∈ adjacent( u )
          if u is "unvisited" then
          begin
             mark u "visited"
             layer[u] := layer[v] + 1
             enqueue( u )
          end
```

The gist of the current paper is to question some aspects in the current practice of advancing from algorithm to code-like description. While the verbal description of BFS simply guides us to advance from the starting vertex s to vertices of the first layer, and then to vertices of the second layer and so on, the code-like description introduces Q, a first-in first-out queue data structure. Suppose that s had 5 adjacent vertices, then while the verbal description allows treating the 5 vertices in parallel, the code-like version dictates treating them one at a time. The serialize-to-parallelize paradox, stated below, confronts this practice with another common practice: attempts by compiler and run-time techniques to extract as much parallelism from the full code as possible.

Suppose now that a different code-like description is given, one allowing parallelism to be expressed. From vertices of the currently visited layer we would visit in parallel all their adjacent vertices. Marking a yet "unvisited" vertex v as "visited" becomes a bit complicated since it is possible that several different "threads" will try to do it simultaneously. A Concurrent-Read Concurrent-Write Parallel Random Access Machine (CRCW PRAM) programmer's model would allow a concurrent-write command. In the current context such a concurrent-write command would mean that an arbitrary one among the attempting threads should succeed in marking v. Still, the paradox of queuing concurrent-writes, presented later, is that current parallel systems typically implement a concurrent-write command in ways which lose the parallelism inherent in such a concurrent-write command.

We are ready to state the two main points of the paper.

The serialize-to-parallelize paradox A closer observation of current practice reveals the following. In a first stage the code writer makes an effort to enforce a serial order which might not be necessary for correctness. This is followed by an attempt to undo this serialization, using compiler and runtime techniques. The current paper actually demonstrates that it is much harder (sometimes maybe impossible) for these techniques to fully reverse the serialization of the code.

The queuing concurrent-writes paradox An important programming primitive used in parallel programming is known as *concurrent writes*. In a parallel program several instructions that can be executed in parallel may include an update of the same variable. Suppose the programming language allows the programmer to specify that only one of these updates, chosen arbitrarily, should occur (the so-called *Arbitrary Concurrent-Write* primitive in [22]). Most current parallel (and serial) systems implement concurrent writes by queuing all the updates and handling them in sequence, relying on run-time protocols resolving "Write After Write" data dependences. This is an overkill: correctness is indeed preserved but parallelism is lost. We see no inherent difficulty in preserving both parallelism and correctness for a parallel program being executed on a uniprocessor.

It is interesting to relate this last paradox to the work of Gibbons, Matias and Ramachandrans in [8]. They contributed a modeling of the queuing of concurrent writes (QRQW PRAM model). Their paper fits a hardware centric approach which seeks to have programmers support available machines, while our approach is software (or algorithm) centric: we try to influence hardware designers to support easy and useful programming paradigms, which should not be too difficult to implement.

The real subject of this paper is how to go about programming so that it will be relatively easy and lead to a high degree of execution parallelism. We suggest that a PRAM-like programmer's model in conjunction with a programming language which represents it, offer a competitive programming platform for the purpose of: (i) "thinking in parallel" as well as easy representation of parallel thinking, (ii) easy programming, and (iii) extracting a high degree of execution parallelism. Concretely, we envision a programming language, or languages, in which: (i) the programmer can direct the compiler to treat certain portions of code as parallelizable, and (ii) the parallel code can include concurrent-writes which will be efficiently implemented. As much as possible, we avoid in this paper the issue of how exactly the parallel programming language should look. We only state that we would like to see parallel programs which implement in code the PRAM programmer's model, much in the way of parallel programming languages such as Fortran90 [1], HPF [11], and C* [21], or NESL for functional programming [3]. Programming should be relatively easy, as suggested by the wealth of literature on this general approach to parallel algorithms and parallel programming.

We demonstrate the usefulness of our approach by measuring the added amount of parallelism which could be extracted by the system we envision. In order to maintain the generality of our results, we made an effort not to restrict ourselves to a specific system. Therefore, we analyzed the instruction trace of a program produced for a serial machine and measured the amount of parallelism which could have been extracted had we had parallel directives. The simulator we used to produce the instruction trace and to analyze it is the one used by Lam and Wilson in [19] for MIPS instruction code. We also used an enhanced form of the simulator which analyzes the instructions as if they were coming from parallel code (as explained in Section 4.2).

Our experimental work shows that for our chosen test algorithm, our approach can help the system extract significantly more parallelism. Henceforth, the *amount of parallelism* in a program is defined as the ratio between the number of cycles in a sequential execution of the program and the number of cycles in a parallel execution. This definition needs some explanation. The number of cycles in a sequential execution of a program is actually the total number of operations performed by that program. Dividing this number by the total number of parallel cycles provides an "average amount of parallelism per cycle". [1] This definition will be valid for comparing two different programs later in the paper in case the two programs have the same total number of operations, since then the one with the fewer parallel cycles obviously has a larger amount of parallelism. But, what if the two programs that we wish to compare differ in their total number of operations? Fortunately, for our paper, the program with the smaller number of operations is shown to also have fewer parallel cycles, making a comparison easy. However, this need not be the case in general,

---

[1] We note that using the total number of operations and the number of parallel cycles as the key measure for evaluating parallel algorithms is rather common. It is used by the so-called "work-time methodology" for describing parallel algorithms. This methodology was first presented in [23] (relying on a 1974 theorem by Brent for an abstract model of parallel computation). Later the book [13] used the methodology as a framework for describing most of the parallel algorithms presented there. For brevity, we avoid elaborating on this issue here.

where one may need to compare two programs, the one having a larger total number of operations but fewer parallel cycles than the other.

Emulations of the system we envision suggest that it can improve the amount of parallelism extracted by at least an order of magnitude, and in some cases by as much as several orders of magnitude. We also tested how much more parallelism can be extracted where concurrent writes need not be queued. The improvement in parallelism extracted ranged up to a factor of nearly 2.5.

An overview of our experimental work is given in the next section, and a review of some previous work that we built upon in Section 3. In Section 4 we explain in further detail the simulators used to measure parallelism, and Section 5 presents the program we chose as input for these measurements. The results obtained in our tests are presented in Section 6, and finally Section 7 concludes our paper.

2 Overview of Tests and Results

In studies done by Lam and Wilson [19] and by Wall [26] the amount of parallelism in programs was measured by running the instruction trace through a simulator, as explained below. Consider hypothetical computer systems having capabilities such as branch prediction (according to one scheme or another) or register renaming in order to eliminate *Write After Write* and *Write After Read* data dependences among the registers. Each set of such assumed capabilities is said to define an *abstract machine model.* The simulator can be set to simulate a variety of such models and measure for various programs the amount of extracted parallelism. Some of the abstract machine models studied allow unrealistic assumptions, such as perfect branch prediction or perfect memory disambiguation, thus providing an upper bound on the amount of parallelism (see Section 3). The abovementioned studies measured in this manner the parallelism extracted from a chosen set of standard benchmarks. For the strongest and most unrealistic abstract machine models, a mean parallelism of 23.6 was found in [26] and of 158.26 in [19]. When some of the unrealistic capabilities were modified to more feasible ones, the mean parallelism for their respective benchmarks dropped to 11.9 in [26] and to 39.62 in [19].

We propose an *envisioned system* in which the programmer can identify parts of the code as parallelizable, and the compiler and hardware have the capability to utilize this parallelism, as well as extract parallelism automatically. We examined the extent to which the envisioned system could increase the amount of parallelism which can be extracted overall. For this, we used the simulator written by Robert P. Wilson for the study in [19] (henceforth the *LW simulator*). We also enhanced the LW Simulator so that it could simulate the capability of the envisioned system in addition to the automatic extraction capabilities it originally assumes (see Section 4.2). The new simulator is henceforth referred to as the *Enhanced* simulator.

Ferrante *et al.* [7] introduced the Program Dependence Graph (PDG) which is a Directed

Acyclic Graph (DAG) representation of the program, incorporating the true data and control dependences. The nodes of the PDG represent the instructions and the edges represent dependences between the instructions. One can also present the (sequential) *execution* of the program as a DAG where each node represents an instance of an instruction. This DAG will include the following edges: (i) There will be an edge from instance j to instance i if j is the last instruction which precedes i in the sequential trace and on which i is control-dependent. (ii) Given instance i of an instruction which has several arguments, for each of these arguments there will be an edge to the last instance j of an instruction on which the argument is data-dependent (e.g., given an instance of the instruction $R1\ :=\ R2\ +\ R3$ there will be an edge to the last instance on which $R2$ is data-dependent and to the last instance on which $R3$ is data-dependent.) The sequential trace is in fact a Depth First Search of the execution graph. The length of an optimal parallel execution is the length of the longest path from the root (or "source") in the DAG, also called the *critical path.* The effect of our enhanced simulator would be to remove some of the edges from the PDG, namely the edges which create dependences between instructions that the code identifies as executable in parallel. This will cause edges to be replaced in the execution graph in a manner which brings nodes closer to the root and will usually shorten the critical path as well (see Section 4.2 for a detailed example).

This paper presents a study of Breadth First Search (BFS) using both the LW and the Enhanced simulators. [2] Our BFS program receives a graph as input, searches it starting from a given node, and returns for each node the id given to it by the search, as well as the layer, or distance from the starting node. Figure 3 shows the sequential algorithm (pseudo) code (SAC) of this program. As has been observed by Eckstein in [6], all the nodes in the same layer could in fact be visited concurrently. Furthermore, all the neighbors of all the nodes in the same layer may be treated in parallel (i.e. put in the queue and marked accordingly), as long as concurrency conflicts arising from concurrent writes are handled correctly (see Section 5.2). We produced another version of sequential code, which emulates the parallel BFS algorithm as closely as possible. The resulting Parallel Algorithm Sequential Code (PASC) appears in Figure 4 (A). Part (B) of this figure shows the fully parallel algorithm, in Pseudo Parallel (PP) code. The PASC version can be compiled and run on a MIPS machine, while the PP version is designed to run on our envisioned system.

Figure 1 illustrates by way of motivation the scheme guiding the comparative experiment we ran. The goal is to examine what can be gained by eliminating the Serialize-to-Parallelize paradox. Figure 1 (A) illustrates the comparison we really wanted to perform. We wished to compare the parallelism obtained from the sequential code, by a system with strong ILP capabilities, to that obtained from the parallel code by our envisioned system. Figure 1 (B) shows the comparisons we actually performed using the three versions of code. Comparing (1) to (2) in this part of the figure will show the advantage of programming by the parallel algorithm *even for current "ILP systems"*. In the example of our test program, direct access arrays were used in version (2). For instance, a direct access array replaced a standard

---

[2] On-line reference to the software on which the current paper relies can be obtained through the World Wide Web at **http://www.umiacs.umd.edu/users/rimon/ilp.html**

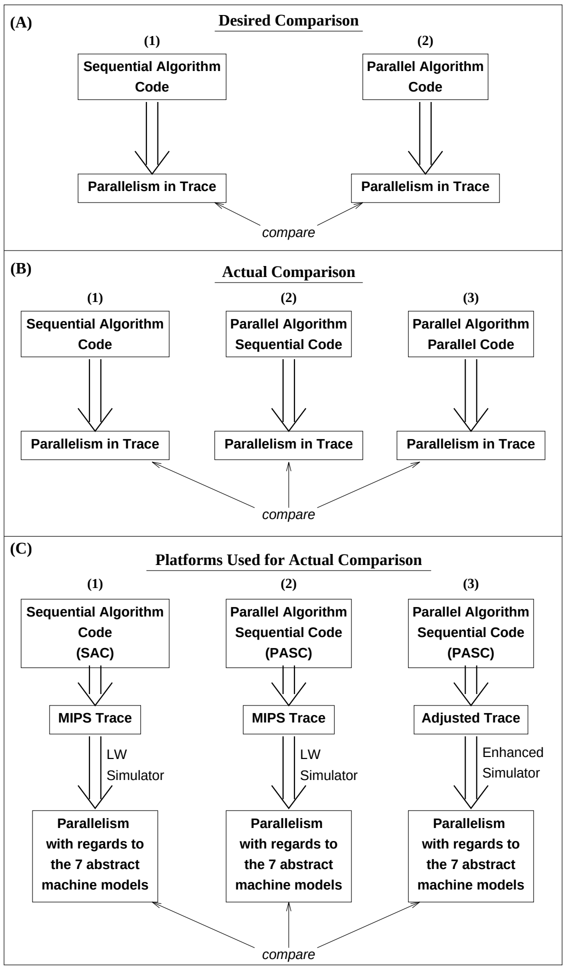

Figure 1: The Scheme of the Comparative Experiment

Machine Model	Nodes	Edges	SAC + LW Sim. (1)	PASC + LW Sim. (2)	PASC + Enh. Sim. (3)	Improvements		
						(2)/(1)	(3)/(2)	(3)/(1)
CD-MF	100	750	26.59	48.55	216.74	1.83	4.46	8.15
	500	2500	20.77	49.59	248.24	2.39	5.01	11.95
SP-CD-MF	100	750	41.43	186.73	351.81	4.51	1.88	8.49
	500	2500	31.65	174.47	443.29	5.51	2.54	14.01

Table 1: Comparison of Amounts of parallelism extracted from SAC version of code with LW simulator and from PASC version of code with the LW simulator and with the Enhanced simulator, for random graphs

first-in first-out (FIFO) queue which was used in version (1), facilitating extraction of more parallelism. The comparison of (2) to (3) shows the advantage of running the parallel algorithm on our envisioned system instead of current ones. In version (3), parallel *for* loops (presented by a DO_PARALLEL command) are performed in parallel, avoiding a branch at the end of each iteration of the loop (which determines whether another iteration should be performed). In this version PREFIX_SUMS primitives are performed in parallel and concurrent-writes are implemented without queuing. Finally, comparing (1) to (3) presents the overall advantage achieved by programming in the parallel model *and* using our envisioned system. Our results demonstrate that the overall improvement can be of more than an order of magnitude.

Figure 1 (C) presents the platforms we used for obtaining the results. To achieve the effect of our envisioned compiler for (3), we adjusted the MIPS trace of the Parallel Algorithm Sequential Code. We then used the enhanced simulator to extract the parallelism from this adjusted trace. For (1) and (2) we simply ran the Lam and Wilson simulator over the MIPS trace of the respective code. In each of the tests we obtained results for each of the seven abstract machine models defined by Lam and Wilson in [19] (see section 4.1).

Table 1 gives a representative sample of our results. For both the sequential and the parallel executions of the tested programs we assume perfect procedure inlining, i.e. removal of all call and return instructions from the trace, as well as all instructions that manipulate the position of the stack pointer. We show here results pertaining to two of the seven abstract machine models – CD-MF and SP-CD-MF. Both models assume perfect memory disambiguation (or memory aliasing), perfect register renaming, full control dependence analysis and the ability to follow unlimited multiple flows of control. The SP-CD-MF model also assumes the ability to predict branches and to speculatively execute instructions according to these predictions. See section 4.1 for further details on these assumptions. The different versions of BFS code were simulated with inputs of random graphs of 100 nodes and 750 edges, and of 500 nodes and 2500 edges. These graphs were constructed using

Machine Model	Nodes	Edges	Queuing		Improvement
			Yes	No	
CD-MF	100	750	159.70	216.74	1.36
	500	2500	122.37	248.24	2.03
SP-CD-MF	100	750	218.69	351.81	1.90
	500	2500	181.39	443.29	2.44

Table 2: Comparison of Amounts of parallelism extracted from PASC version of code with the Enhanced simulator, with or without queuing of concurrent writes, for random graphs

The Stanford GraphBase [15]. Results for all abstract machine models and additional input graphs appear in Section 6 and in the appendix. For brevity, we have excluded from the current paper results for other families of graphs. In particular, we have excluded results obtained for various graphs constructed deterministically, which are similar to the results reported here.

It is clear from the results that our approach can bring significant improvement in the amount of parallelism extracted from a program. Even when assuming existing technology, we improved the parallelism simply by rewriting the program to reflect such concurrency as we saw. This action alone brings an improvement of at least 83% in this sample of results, and sometimes much more (Column *(2)/(1)*). By changing to our envisioned system instead, we can further improve the parallelism extracted from the PASC code version by at least 88% for these sample results (Column *(3)/(2)*). Changing both the code and the system brings an improvement of around an order of magnitude in the amount of parallelism (Column *(3)/(1)*). The full results in the appendix of this paper show an overall improvement ranging from a factor of 2.73 up to a factor of 1334.94.

We presented in the introduction the paradox of Queuing Concurrent-Writes. In order to show how much can be lost by such queuing practice, we provided the Enhanced simulator with the ability to reflect such queuing, if so desired (see Section 4.3). Table 2 shows results of measuring the parallelism in the PASC version of the BFS program with and without queuing. The results without queuing, presented in this table, are those from column *(3)* in Table 1. We see that for these sample results, eliminating this practice can bring an improvement in parallelism ranging from 36% to over 100%. The full results (see appendix) show improvements ranging from 5% to 247%.

3 Studies on the Automatic Extraction of ILP

Compiler techniques for extracting ILP include software pipelining [18, 14, 16], loop unrolling and trace scheduling [9]. Runtime techniques include speculative execution, conditional execution, and branch prediction [9]. One kind of machine which uses the parallelism found by compiler or runtime techniques is multiscalar machines [27]. Since relatively little attention has been given to using these techniques for programs which were not initially designed for a strictly serial processor, we mention one outstanding issue with respect to doing so. To run such programs efficiently, the hardware may need to have the ability to handle a number of concurrent threads in more efficient ways than in many existing machines. However, this is hardly a disadvantage in current times. In [10], Hennessy and Patterson imply that an increase in ability to efficiently handle multi-threading is already needed in the context of exploiting large amounts of parallelism which are extracted from serial code.

The literature reports numerous efforts to study and observe the amount of parallelism which can be automatically extracted from a program. For brevity, we elaborate only on works that measured the parallelism in the instruction trace of C or Fortran programs. Specifically, we focus on the works of [19, 26] that measured this parallelism using a simulator. Other works annotated the high level program, adding code so that data regarding parallelism is collected at runtime [17, 5]. When working with an instruction trace, The simulator analyzes the dependences for each instruction and determines accordingly the (earliest) parallel cycle in which the instruction can be executed. Each of these recent works makes several alternative assumptions regarding the ability of the simulated machine to extract parallelism. Each set of assumptions defines an *abstract machine model*, as explained earlier. The papers report their results with respect to each of their abstract machine models. Common to both Lam and Wilson's work and Wall's one are "aggressive" abstract machine models whose ability to extract parallelism exceeds the feasible. Such an abstract model is valuable as it provides a yardstick for the amount of parallelism which one could realistically expect. These two works consider also weaker abstract machine models, where less parallelism can be extracted.

Each of the most aggressive abstract machine models in [19, 26] assumes the following capabilities:

perfect branch prediction – At the beginning of the execution it is already known which path will be taken by every branch. Accuracy of current branch prediction techniques has been shown to exceed 85% for many applications [20, 12].

perfect memory disambiguation – At the beginning of the execution the contents of all address registers are already known. This becomes an issue in two different settings: (i) *Indirect memory accesses:* If these contents are not known in advance we must assume that all indirect memory accesses are to the same location, possibly enforcing false data dependences. Wall [26] refers to this problem as *the aliasing problem.* (ii) *Indirect jumps:* The lack of this foreknowledge would mean that all instructions after the jump cannot be executed until the target address is resolved.

perfect register renaming – Register storage dependences which are not required for preserving correctness of the original code can be eliminated by using register renaming techniques [9]. In order to guaranty elimination of *all* such dependences we must also assume having an unbounded number of registers.

These assumptions provide an upper bound on the possible amount of parallelism but are unrealistic since we are not aware of general ways to accomplish perfect branch prediction or perfect memory disambiguation without executing the program, and an unbounded number of registers is clearly unrealistic. Both studies show that any of several restrictions of these assumptions cuts the mean possible amount of parallelism by at least half. Austin and Sohi [2] use similar models, as well.

Wall measured the parallelism extracted from a suite of 18 programs under different abstract machine model assumptions. When assuming his aggressive "perfect" machine model, he found that the mean parallelism is 23.6 over the chosen suite of benchmarks. Lam and Wilson found a mean parallelism of 158.26 over a suite of seven non-numeric programs, with their similarly aggressive "oracle" model. Wall tried limiting the number of registers to 256, using branch prediction techniques which require a total of over 146 Kbytes of memory. Since perfect memory disambiguation is assumed, this model is still unrealistic, and yet the mean parallelism for this model was only 11.9. Lam and Wilson found a mean parallelism of 39.62 for a model with speculative execution, full control dependence analysis and pursuing multiple flows of control, all this while still assuming perfect memory disambiguation.

It is clear from the results of both studies that the amount of parallelism which can be extracted automatically is rather limited, even when we invest much hardware and software in the effort.

4 Measuring Parallelism

We used the LW simulator where possible. In particular, all the results in this paper concerning simulations of systems which are not based on explicit parallel programming, were obtained using the LW simulator. After describing this simulator, we review its enhancements for simulating two different systems that do allow explicit parallel programming: Section 4.2 describes one in which concurrent-writes take one parallel cycle and Section 4.3 slows this by queuing concurrent-write.

4.1 The LW Simulator

We worked with what we call the LW simulator, which was originally used in [19] to measure parallelism. Some of the basic results in the present paper were obtained using this simulator as is. Using *pixie* [24], the simulator produces the instruction trace of the program. It then analyzes the dependences and determines the parallel cycle in which to execute each instruction. Two major forms of dependence are taken into account – data dependences and control dependences. Regarding data dependences, only Read after Write (RAW) dependence is considered. Write after Write and Write after Read dependences can be removed by perfect register renaming techniques as explained in the previous section.

As for control dependences, the simulator pursues three possible techniques for relaxing control flow constraints - Control Dependence Analysis (CD), Executing Multiple Flows of Control (MF) and Speculation with Branch Prediction (SP). The branches are predicted based on statistics which are collected beforehand by running the simulated program with the same inputs to be used in the simulation. Let us consider an example to illustrate these three techniques.

```
1.        if ( a == 0 )
2.            b = 5;
3.        c = 7;
4.        if ( d == 0 )
5.            e = 3;
```

Note that there are no data dependences in this example. When using none of the above techniques, all the instructions will have to wait until line 1 is executed, since the compiler cannot tell whether the following instructions depend on the first branch. Line 5 will also have to wait until the result of the branch on line 4 is known.

Control Dependence Analysis determines the dependence of instructions on their preceding branches. When an instruction is not dependent on the preceding branch, it can be executed before the outcome of the branch is known. With CD the compiler can deduce that only line 2 depends on the outcome of the first branch, and the rest of the code could be executed disregarding this outcome. In this case we could execute line 3 in parallel with line 1. When we do not have the ability to execute multiple flows of control, we cannot follow a new branch until we have resolved all previous branches. Therefore as long as we do not have MF we must execute all branches in order. Thus Line 4 will have to wait until the first branch is resolved. Line 5 still has to wait for its preceding branch, since it is dependent upon it.

The capability to pursue Multiple Flows of Control allows us to fetch and execute instructions from different regions of code at the same time. Multiple Flows of Control are significant only when combined with Control Dependence Analysis which might discover parallelism across such regions. With MF we only have to execute branches in order if one is dependent on the other. In our example, using MF as well as CD would enable us to execute line 4 in parallel with line 1, since there is no control dependence between the two branches. Line 5 could then be executed in the following cycle.

With Speculation and Branch Prediction we try to predict the outcome of each branch and will execute instructions along the predicted path even before the branch has been resolved. If the prediction was wrong, we must start again on the correct path, therefore the instructions would actually wait for their preceding branch. Suppose in our example that the first branch was mispredicted but the second one was not. In this case we will have to execute the first branch before we can execute any of the following instructions (note that we are not using CD, so we do not know that line 3 is independent of the branch).

In the second cycle we can execute lines 2, 3 and 4 as well as line 5, since we assumed the second branch was correctly predicted.

Using different combinations of these three techniques, we can measure parallelism for seven abstract models of machines defined by Lam and Wilson in [19] as follows:

BASE – No technique used for relaxing control flow constraints.

CD – Using only control dependence analysis.

CD-MF – Using control dependence analysis and executing multiple flows of control.

SP – Using only speculation with branch prediction.

SP-CD – Using speculation with branch prediction and control dependence analysis.

SP-CD-MF – Using all three techniques for relaxing control flow constraints.

ORACLE – Assuming there are no control flow constraints at all, only data dependences are considered. The machine has *oracle* knowledge of the control flow.

(See [19] for more details and examples).

4.2 Enhancing the Simulator

To measure the parallelism which could be achieved by using a parallel programmer's model, we enhanced the LW simulator so that it would disregard dependences between those blocks of code that the programmer specified as executable in parallel. For example, given some serial code, suppose the programmer knows that the different instances of a loop are independent of each other. The programmer will be able to specify this loop as executable in parallel and the enhanced simulator will "understand" this specification.

The general thesis that led to this work [25] advocated that using a parallel programmer's model is worthwhile. In order to test this thesis in a meaningful way we need to present results which are as widely applicable as possible. Therefore, we do not suggest here a specific system which does this, but rather offer a general scheme of what needs to be done. Committing to a specific solution would have produced a less robust result, depending on too many ad-hoc parameters and thus not necessarily applicable to the general case.

To avoid defining a programming language and developing a specific compiler, we did the following. We analyzed the sequential machine code of our test program to identify the boundaries of the parallelizable blocks of code, and fed these boundaries directly to the enhanced simulator. In order to enable the simulator to parallelize these blocks, we made it disregard dependences among them. For each block we will consider only dependences coming from blocks which appeared earlier in the trace and which are *not* specified as parallel to this block.

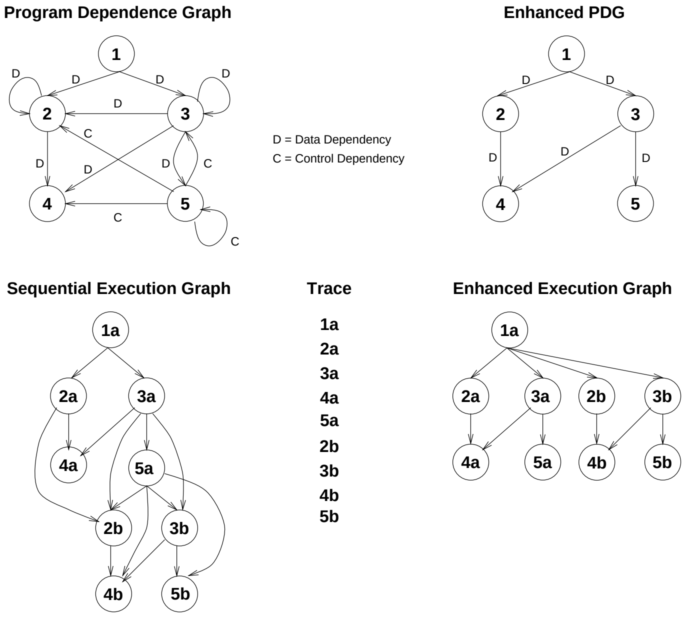

Figure 2: Examples of Program dependence graphs and execution graphs for regular and enhanced executions.

The LW simulator analyzes each instruction as it appears in the trace and determines the parallel cycle in which it should be executed. This decision is made based on the instruction's dependence on all the instructions that preceded it in the trace. The instruction is assigned to the earliest cycle succeeding all cycles that include an instruction on which it depends. The Enhanced simulator reconsiders each dependence upon encountering it and determines whether it should now be disregarded as explicated above.

Other adjustments were made to the instruction trace of the program as follows. Each instance of a serial loop ends with a branch instruction. Suppose now that the same loop is specified as parallel. These instances could be spawned off by some mechanism and executed in parallel. The branch instruction becomes irrelevant, and should not exist in an actual implementation. Thus, we remove each branch instruction which appears at the end of each instance of the loop which is being parallelized. We also remove instructions which increment the loop counter, assuming that the spawning mechanism will keep track of the indices. While removed from the execution not to create unwanted dependences, the removed instructions are nevertheless counted. This is done to compensate for parallel control-related instructions which will replace them in an envisioned system. In other words, the instructions are included in the count to enable better approximation of the expected performance of an envisioned system.

The resulting simulator is called the *Enhanced* simulator.

Let us demonstrate the analysis performed by the Enhanced simulator with the following example, given in pseudo instructions:

```
1.  LOADI    R1, 0           ;Loading 0 into R1
2.  ADD      A1, A1, R1      ;Add R1 to A1
3.  ADD      R1, R1, 1       ;Add 1 to R1
4.  STORE    [A1], R1        ;Store R1 at address found in A1
5.  BST      R1, R3, #2      ;If R1 is smaller than R3, branch to inst. #2
```

Figure 2 shows the program dependence graph representing this program fragment. The numbers in the nodes correspond to the numbers of the instructions in our example. An edge from node i to node j illustrates that instruction j is dependent on instruction i. Edges marked D represent data dependences and those marked C represent control dependences. The figure also shows a possible instruction trace, looping twice through the code, and the sequential execution graph corresponding to this trace. The letters attached to the numbers in the trace and execution graph mark the different instances of the instructions. Note that the length of the critical path, or the depth of the sequential execution DAG, is 4 (assuming all edges are of size 1, representing the latency). If the programmer were to identify this example as a parallelizable loop, the Enhanced simulator would disregard dependences between different iterations, and the resulting "enhanced" program dependence graph is presented in Figure 2. For the same trace presented in the figure, the execution graph for the enhanced simulator is also presented. However, the following caveat is in place. Since instruction 5 is the branch at the end of the loop, it would actually not appear in an envisioned parallel execution. As explained earlier in this section, the branch instruction is nevertheless counted to enable better approximation of an envisioned system. The critical path of the enhanced simulation graph is only 2, which is half that of the regular simulation. If the loop were to iterate more than twice, the improvement of the enhanced simulation over the regular one would be even greater.

4.3 Queuing Concurrent Writes

We pointed out earlier that the present day practice of implementing concurrent-writes by queuing them slows down the execution of parallel programs. Therefore we added a simple feature to our Enhanced simulator which when activated emulates queuing of concurrent-writes (slowing down the execution). It works as follows. The stores to memory which should be treated as concurrent writes are identified in the machine code and fed to the simulator. When such a concurrent write instruction is analyzed, the simulator assigns it to the earliest parallel cycle after the last concurrent-write to the same memory location which precedes the instruction in the serial trace. This enables studying the gap between the amount of parallelism with and without queuing of concurrent-writes.

In addition to Arbitrary Concurrent Writes, the parallel code PP uses a PREFIX_SUMS primitive. See later reference in Section 5.4 to the parallel code and this primitive. While PREFIX_SUMS is more general than Arbitrary Concurrent Writes, it is conceivable that a processor supporting ILP, which can do a k-fold PREFIX_SUMS concurrently, can execute such k-fold PREFIX_SUMS in the same time as a single (i.e., 1-fold) PREFIX_SUMS. Thus it would make sense to consider the possibility of no queuing even for a strong primitive such as PREFIX_SUMS.

5 The Test Program

For our experiment we used a (standard) program which performs Breadth First Search (BFS) on an undirected graph. For each node we wish to obtain the index in the search order as well as the *layer* to which it belongs - i.e., the length of the shortest path from the node at which the search begins. BFS traverses the graph starting from a given node, and then visiting each of its neighbors in arbitrary order. Visiting a node means giving the node an index and assigning it to the correct layer, as well as marking (in arbitrary order, or in parallel using *arbitrary concurrent writes*) all unvisited neighbors of this node as "visited" and putting them into a queue. Once all the nodes in one layer are visited, the nodes in the queue are taken out and visited, creating the next layer, whose nodes can be visited in arbitrary order. A sequential version is followed by parallel ones.

5.1 Sequential Version

Figure 3 shows the straightforward C code of the main procedure *bfs* for the sequential algorithm. We denote this version of code as the SAC version.

We have omitted the variable declarations from the code. Following are descriptions of the variables used:

current The index of the node we are currently visiting.

i,j Running loop indices.

id The running counter for the id given to each node according to the search order.

LayerNum The running counter for the layers identified by the search.

nghbr The index of the neighboring node we are currently treating.

NumNghbrs The number of neighbors of the current node.

LayerSize The number of nodes in the layer we are currently visiting.

NextLayerSize The number of nodes in the next layer.

data An array of entries, holding for each node the data obtained by the search. Each entry holds the *id* (initially set to zero) and *layer* obtained for the corresponding node.

Also omitted here are the following procedures which are called from *bfs*:

```
bfs( StartingNode ) {
  id = 1;
  LayerNum = 0;
  Qput( StartingNode );
  LayerSize = 1;
  do {
                  /* Each iteration treats one layer */
    NextLayerSize = 0;
    for ( i = 0; i < LayerSize; i++ ) {
                  /* Each iteration treats one node in the current layer */
      current = Qget();
      data[ current ].id = id++;
      data[ current ].layer = LayerNum;
      NumNghbrs = GetNumNghbrs( current );
      for ( j = 0; j < NumNghbrs; j++ ) {
                        /* Each iteration visits one neighbor */
        nghbr = GetNghbr( current, j );
        if ( data[ nghbr ].id == 0 ) {
                        /* Node has not yet been visited */
          data[ nghbr ].id = -1;
          NextLayerSize++;
          Qput( nghbr );
        }
      }
    }
    LayerNum++;
    LayerSize = NextLayerSize;
  } while ( LayerSize ≠ 0 );
}
```

Figure 3: Straightforward Sequential Algorithm C (SAC) Code for *bfs* procedure

Qput(node) An interface call to the queue which holds the nodes during the search. *Qput* adds the given node to the queue.

int Qget() An interface call to the queue. *Qget* returns the next node in the queue.

int GetNumNghbrs(current) Returns the number of neighbors of the *current* node.

int GetNghbr(current, j) Returns the index of the node which is the *j*th neighbor of the *current* node. The order of the neighbors is arbitrary but is set throughout the execution.

5.2 The Parallel Algorithm

Although BFS is known primarily as a sequential algorithm, it is also known that certain parts of the code can be executed in parallel [6]. First of all when visiting a certain node, we sequentially check all its neighbors and put them into the queue if they have not yet been visited. In fact we could treat all the neighbors in parallel since they are independent of each other. Furthermore the original algorithm gets one item at a time from the queue but in fact all the nodes in a certain *layer* can be found in the queue immediately after all nodes of the preceding layers have been visited. All these nodes can be extracted from the queue and visited in parallel. These visits are generally independent of one another, but there is one caveat: a node of the next layer can sometimes be reached from more than one node of the current layer. Using concurrent-writes, and in particular the arbitrary concurrent-write convention, we will make sure that each node of the next layer is being put in the queue exactly once.

This paper examines using such parallelization for a modern uniprocessor which allows ILP (and not only for a multiprocessor machine).

Figure 4 (B) shows the *bfs* procedure written in pseudo parallel (PP) code (further discussed in Section 5.4). Figure 4 (A) shows a translation of the parallel code into sequential C code (denoted as Parallel Algorithm Sequential C code, or PASC for short).

Notice that the PASC version exposes much more parallelism than the SAC version while still using a sequential programming language. The concrete changes from the SAC version are as follows:

- The queue structure is now implemented as a simple array, with the interface calls identifying exactly which place in the array they are using. Thus we have *Qput(node, place)* which puts the given *node* in the given *place* in the array (e.g. Line 5 in Figure 4), and *Qget(place)* which returns the *node* stored in the given *place* (e.g. Line 12).
- The new variable PrevLayers holds the total number of nodes in the previous layers. It is used to compute the correct place in the queue to put the next node (Line 24).

```
bfs( StartingNode ) {
 id = 1;
 LayerNum = 0;
 PrevLayers = 1;
 Qput(StartingNode,0);
 LayerSize = 1;
 do {
  /*Iteration treats one layer*/
  NextLayerSize = 0;
  for (i = 0; i < LayerSize; i++) {
 /* Iteration treats one node in layer */
   current[i] = Qget(id-1+i);
   data[current[i]].id = id+i;
   data[current[i]].layer = LayerNum;
   NumNghbrs[i] = GetNumNghbrs(current[i]);
   for (j = 0; j < NumNghbrs[i]; j++){
   /*Iteration visits one neighbor*/
    nghbr[i,j] = GetNghbr(current[i], j);
    if (data[nghbr[i,j]].id == 0){
    /*Node has not yet been visited*/
     data[nghbr[i,j]].id = -1;
     NextLayerSize++;
     Qput(nghbr[i,j],
          NextLayerSize+PrevLayers);
    }
   }
  }
  id += LayerSize;
  LayerNum++;
  LayerSize = NextLayerSize;
  PrevLayers += NextLayerSize;
 } while (LayerSize≠0);
}
```

(A)

```
bfs( StartingNode ) {
 id = 1;
 LayerNum = 0;
 PrevLayers = 1;
 Qput(StartingNode,0);
 LayerSize = 1;
 do {
  /*Each instance treats one layer*/
  NextLayerSize = 0;
  DO_PARALLEL (i ∈ [0;LayerSize-1]) {
/* Each instance treats one node in layer */
   current[i] = Qget(id-1+i);
   data[current[i]].id = id+i;
   data[current[i]].layer = LayerNum;
   NumNghbrs[i] = GetNumNghbrs(current[i]);
   DO_PARALLEL(j ∈ <[0;NumNghbrs[i]-1]){
   /*Each instance visits one neighbor*/
    nghbr[i,j] = GetNghbr(current[i], j);
    if (data[nghbr[i,j]].id == (0,0))
    /*Node has not yet been visited*/
     data[nghbr[i,j]].id = (-1,i)
    if (data[nghbr[i,j]].id == (-1,i)) {
     prfx[i,j]=PREFIX_SUMS(NxtLayerSize,1);
     Qput(nghbr[i,j],prfx[i,j]+PrevLayers);
    }
   }
  }
  id += LayerSize;
  LayerNum++;
  LayerSize = NextLayerSize;
  PrevLayers += NextLayerSize;
 } while (LayerSize≠0);
}
```

(B)

Figure 4: (A) Parallel Algorithm Sequential C (PASC) Code for *bfs* Procedure. (B) Parallel Pseudo (PP) Code for *bfs* Procedure.

- Instead of incrementing *id* with every new iteration of the loop (see Line 12 in Figure 3), we now update it only at the end of each layer (Line 28 in Figure 4), and use the loop index *i* to obtain the correct id to be given to the node (Line 13 there).
- Instead of reusing the variables *current, NumNghbrs* and *nghbr*, we now have arrays, e.g. *current[]*. Each iteration will use the corresponding entry in the array (e.g. Line 12). This eliminates data dependences resulting from accessing the same variable in different iterations. Since *nghbr* is accessed in every iteration of the inner *for* loop, and this is done for every iteration of the outer *for* loop, it is replaced by a two dimensional array, *nghbr[,]*, in order to avoid all data dependences among all the iterations.

  Implementation Comment: We see two possible implementations for the arrays just described. The first, which is also the one used in the LW simulation, relies on the fact that the dependences avoided by using arrays are only storage dependences, i.e., WAW or WAR. Since these dependences can be resolved by renaming techniques, and our simulator assumes perfect renaming, we did not even need to implement the arrays in the BFS program used for the tests. The second possible implementation would use techniques for flattening nested parallelism [4]. In the BFS algorithm, this would mean replacing the outer loop (Line 10) and the inner loop (Line 16) by a single "full" loop, where each iteration of an inner loop forms an iteration of the full loop. This would alleviate the need for implementing in memory the full size of the two dimensional arrays. This second implementation seems to us a suitable choice for the envisioned system.

We ran The LW Simulator with the PASC code version and compared the amount of parallelism found with that extracted from the SAC version using the same simulator. As shown in Section 6, For most of the simulator's machine models, the PASC version allows much more parallelism to be extracted, up to an order of magnitude more in some cases.

5.3 Enhanced Simulation of the Parallel Algorithm

As explained in Section 4.2 above, we must identify the blocks of code which could be executed in parallel when using our envisioned system. In the *bfs* program these are all the instances of the inner *for* loop beginning at Line 10 until Line 27 in Fig 4, each of which visits one neighbor, and this for all instances of the outer *for* loop from Line 16 to 26, each visiting one node in the current layer.

Figure 5 demonstrates graphically the execution of the code blocks as they relate to the nodes of a sample graph portion. The dashed lines in the smallest boxes of the figure represent the instructions. Any instructions that appear in sequence in reference to the *time* axis will be executed in the order they appear, unless the simulator finds that they are not dependent. When two such boxes are horizontally side-by-side, it means they can

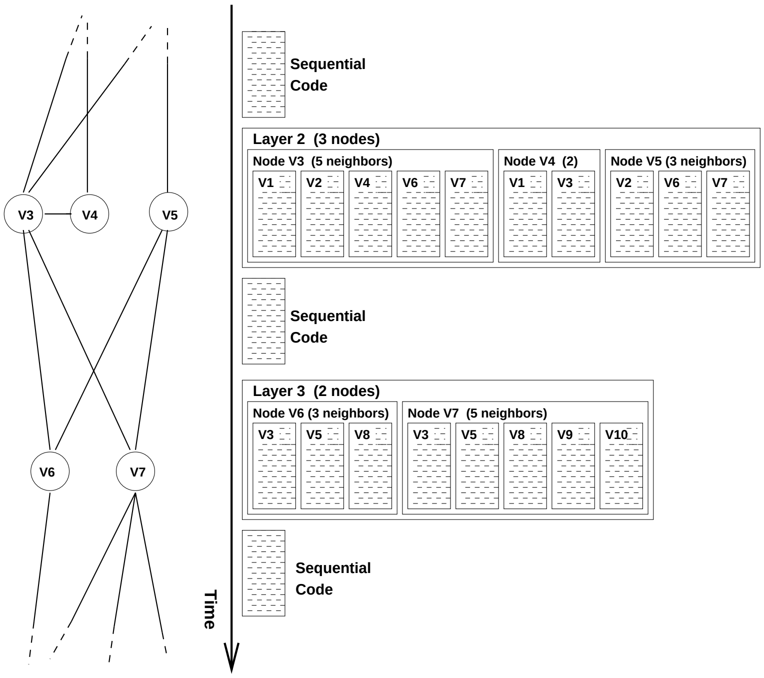

Figure 5: Execution, in time, of the code blocks relating to a sample portion of a graph input to *bfs*

be executed in parallel even if they appear to the LW Simulator to be dependent. In other words, a dependence will be considered only if it is from a previous layer of the graph, or if it is from the current instance of the inner *for* loop in figure 4. Dependences on instructions from a different instance of the loop in the same layer are disregarded.

We removed from the trace of the test program all instances of instructions of the following three kinds:

- The branch instructions at the end of the inner and outer *for* loops.
- The instructions which set the condition for the aforementioned branches. These are MIPS instructions which compute the value of a condition and put this value into a register which is then used by the following branch.
- the instructions which increment i and j in the *for* loops.

These instructions are necessary only for a serial execution of the code. When executing the loop instances in parallel there will be no branches at the end of each instance, and the loop indices will be provided by a spawning mechanism (possibly in hardware). We wish to take into account, albeit approximately, the number of cycles required for the spawning instruction, which is likely to involve some overheads. Therefore we did count the “removed” instructions in the total number of instructions, and assigned them to cycles in the parallel execution. However, we did not actually execute them, in order to prevent them from creating dependences which may hold up succeeding instructions.

Running the Enhanced Simulator with the PASC version of the code gives us an approximation of what can be expected from our envisioned system. Comparing the amount of parallelism extracted in this way to the amount extracted by the LW Simulator with the SAC version shows that our approach can improve the amount of parallelism by an order of magnitude, or more. The detailed results appear in Section 6

5.4 The Envisioned Parallel Code

Figure 4 (B) shows Pseudo Parallel (PP) code for the *bfs* procedure. This code provides the programmer with a PRAM-style parallel model. In particular, in this fully synchronous programmer's model each step takes the same time to execute in each concurrent thread, and no thread may continue to a next step until all other threads have finished executing the current step. Compiler and runtime techniques may then be used to relax synchrony during execution, provided that this does not modify the semantics of the execution. In the enhanced simulation described above, we have relied on such relaxed synchrony.

The commands in the PP code which differ from sequential C are the following:

DO_PARALLEL (Lines 10 and 16) Directs the compiler to treat the applicable section (in case of Line 10 this section would be from Line 11 to Line 27) as parallelizable, much in the way that is done in parallel programming languages such as Fortran90 [1], HPF [11], and C* [21]. A possible execution is one where the system spawns a number of threads, each with a unique index, based on the argument in the DO_PARALLEL command.

PREFIX_SUMS (Line 23) This is a procedure call which can be called from several threads in parallel and receives two arguments: a variable $BaseVar$ is passed by reference, and an $Increment$ is passed by value. It returns one value as explained next. All threads which call PREFIX_SUMS with the same $BaseVar$ variable are sorted in some (arbitrary) order $t_1, t_2, \ldots, t_k$ where k is the number of those threads. Thread t_j will then return the value $BaseVar + Increment_1 + Increment_2 + \ldots + Increment_j$ (in this case into the new variable $prfx[i,j]$). In addition, the value of $BaseVar$ is updated to $BaseVar\ +\ Increment_1\ +\ Increment_2\ +\ \ldots\ +\ Increment_k$

Note that this description does not dictate an implementation, but only defines what the return values should be.

Concurrent Write (Line 21) This is not an explicit command, but the compiler and runtime system can detect when more than one thread is trying to write to the same variable in the same step. Such an event is identified as a concurrent write. Only one of these threads, chosen arbitrarily, will be allowed to succeed writing. Note that in the PP code the *id* field of the *data* array records has been expanded to hold a second value in addition to the id of the node. The thread which is successful in the concurrent write will write its index i together with the value into the *id* field, and so in the next step each thread can find out whether or not it was successful.

The PP code would run on our envisioned system. This would be supported by the compiler and possibly hardware capabilities. The compiler would identify parallelism automatically, as done in some modern processors, as well as derive the parallelism from the explicit parallel directives in the code.

6 Results

Figure 1 (C) presents the platforms used for deriving our results. Item (1) represents running the LW simulator with the SAC version, (2) represents running the same simulator with the PASC version, and finally (3) represents running the enhanced simulator with the adjusted trace of the PASC version. The first two items reflect present day systems, while the third item provides estimated results for the envisioned system.

We have defined the *amount of parallelism* in a given piece of code as the ratio between the number of instructions in a sequential execution (i.e., total number of instructions executed) and the number of cycles required for the parallel execution of the code. Following [19], we assume perfect procedure inlining for both executions. This means that all call and return instructions are removed from the trace, as are instructions which manipulate the position of the stack pointer.

For the results reported in this section, random graphs constructed using Knuth's Stanford GraphBase [15] were used as inputs to *bfs*. We bring here a sample of the tests we conducted. Full results appear in the Appendix.

6.1 Comparing the Sequential Algorithm to the Parallel Algorithm on a Sequential Machine

Can parallel programming be advantageous even on (current, or moderately updated) uniprocessors which support instruction level parallelism? To investigate this point, we ran the LW Simulator both on the Sequential Algorithm Code (SAC) and on the Parallel Algorithm Sequential Code (PASC) (Comparing items (1) and (2) of Figure 1 (C)). The total number of machine instructions executed in the SAC version is slightly larger than that in the PASC version – e.g. for a random graph of 500 nodes and 2500 edges, the SAC version executes 116824 instructions, while the PASC version executes 88975 instructions. The amounts of parallelism extracted from the two versions are therefore not fully comparable. Since in all our experiments, the total number of instructions executed in the PASC version never exceeded that in the SAC version (for more on this, see later reference to Table 5), it is important to understand that these results (Comparing items (1) and (2) of

Machine Model	Nodes; Edges	SAC + LW Sim. (1)		PASC + LW Sim. (2)		PASC + Enh. Sim. (3)		Improvements in Parallelism		
		par	cycles	par	cycles	par	cycles	(2)/(1)	(3)/(2)	(3)/(1)
BASE	100; 750	1.52	21246	1.32	18388	64.39	377	0.87	48.78	42.36
CD		1.72	18988	1.46	16631	75.62	321	0.85	51.79	43.97
CD-MF		26.59	2722	48.55	500	216.74	112	1.83	4.46	8.15
SP		10.17	3734	8.83	2750	192.66	126	0.87	21.82	18.94
SP-CD		32.39	1721	97.10	250	279.02	87	3.00	2.87	8.61
SP-CD-MF		41.43	1003	186.73	130	351.81	69	4.51	1.88	8.49
ORACLE		42.31	1003	252.86	96	505.73	48	5.98	2.00	11.95
BASE	500; 2500	1.64	69949	1.37	63591	123.59	703	0.84	90.21	75.36
CD		1.87	61390	1.51	57533	134.29	647	0.81	88.93	71.81
CD-MF		20.77	5524	49.59	1752	248.24	350	2.39	5.01	11.95
SP		9.27	12383	7.43	11694	295.53	294	0.80	39.78	31.88
SP-CD		25.38	4521	75.16	1156	425.91	204	2.96	5.67	16.78
SP-CD-MF		31.65	3625	174.47	498	443.29	196	5.51	2.54	14.01
ORACLE		32.74	3505	257.82	337	668.35	130	7.87	2.59	20.41

Table 3: Comparison of amounts of parallelism (denoted par) extracted for SAC and PASC versions of code with the LW Simulator and for the PASC version with the Enhanced Simulator, for random graphs. The number of cycles is denoted cycles.

Figure 1 (C)), actually understate the advantage of the the PASC version over the SAC version. The difference in the total number of instructions is still small enough to make these results meaningful.

Table 3 shows the amount of parallelism extracted from the two versions in columns *(1)* and *(2)*, and the ratio between them in the column marked *(2)/(1)*. The results shown are for input graphs of 100 nodes and 750 edges, and of 500 nodes and 2500 edges. Figure 6 illustrates the results for this ratio for inputs of 100 or 500 node graphs and various numbers of edges (see legend).

For the BASE model we see that the amount of parallelism extracted from both versions is very similar. (In fact, in most cases it is slightly higher for the SAC version, but comparing the amount of parallelism becomes less meaningful because the SAC version has slightly more instructions). This applies also to the CD and SP models. To understand this, we note that in these models the branches must be executed in order, since there are no multiple flows of control. This requirement already dictates a long critical path, so that even where there are no data dependences not much can be done to reduce the number of parallel cycles. Since both the SAC and the PASC versions have the same flow of control, they produce about the same amount of parallelism for these models.

Once the constraints of branch ordering are sufficiently relaxed by additional techniques, we see that the PASC version has a clear advantage, and allows more parallelism by avoiding

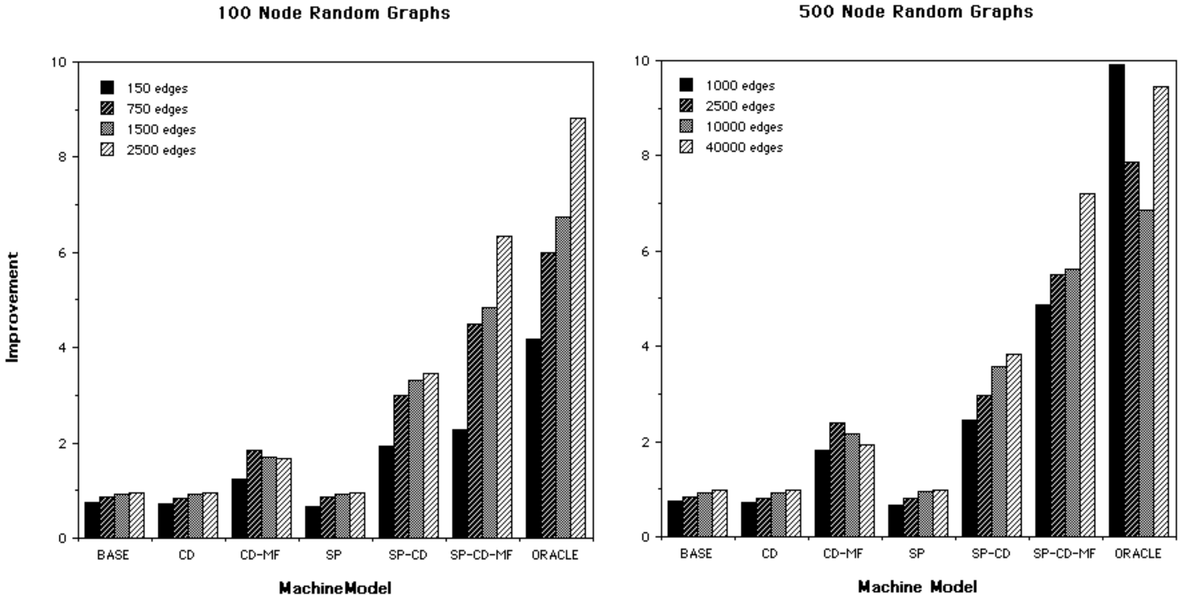

Figure 6: The improvement of PASC version over SAC version, regarding the amount of parallelism extracted by the LW simulator, for 100 and 500 node random graphs. Alongside the parallelism measure we also show the number of cycles required for a parallel execution.

unnecessary data dependences. The improvement for the PASC version grows as the models regard less control dependences, and is largest for the ORACLE model.

Table 4 shows the amount of parallelism and improvements for inputs of 100 nodes over varying numbers of edges. All the results here are for the SP-CD abstract machine model. Note that the more edges there are in the graph, the less layers there will be in the BFS, and each layer will be larger. It is clear from this table and also from Figure 6 that both versions of code allow more parallelism when the layer size is larger. This is to be expected since every time we start a new layer we introduce more control and data dependences. Note that the more edges we have, the more instructions, and particularly the more branches, will appear in the execution; where the branches are strictly ordered, as in the BASE and CD models, a growing layer size might not increase parallelism. The effect of the layer size is more significant in the PASC version because here we come closer to performing the parallel algorithm, which can visit all the neighbors of all the nodes in a single layer at the same time. Therefore, the larger the layers are for a fixed number of nodes, the less parallel cycles will be needed.

Number of Edges	SAC + LW Sim. (1)		PASC + LW Sim. (2)		PASC + Enh. Sim. (3)		Improvements in Amount of Parallelism		
	par	cycles	par	cycles	par	cycles	(2)/(1)	(3)/(2)	(3)/(1)
150	15.53	1685	30.02	282	49.22	172	1.93	1.64	3.17
750	32.39	1721	97.10	250	279.02	87	3.00	2.87	8.61
1500	53.55	1721	177.19	247	625.21	70	3.31	3.53	11.68
2500	81.78	1721	282.45	247	996.64	70	3.45	3.53	12.19

Table 4: Comparison of Amounts of parallelism extracted from SAC and PASC versions of code with the LW Simulator and from the PASC version with the Enhanced Simulator, for the SP-CD model, with input random graphs of 100 nodes and varying numbers of edges

6.2 Comparing the Parallel Algorithm on a Sequential Machine to the Parallel Algorithm on Our Envisioned System

The previous section suggests that even with current technology there is an advantage in using the parallel algorithm. Below, we report how much more can be achieved by using our Envisioned System. The column marked *(3)* in Table 3 shows the results of running the Enhanced Simulator with the Parallel Algorithm Sequential Code (PASC). The column marked *(3)/(2)* shows the improvement of these results over running the LW simulator with the same code version (column *(2)*). Figure 7 illustrates the improvements for graphs of 100 or 500 nodes and various numbers of edges.

As the machine models grow more lenient in enforcing dependences, the advantage of the Envisioned System usually decreases, because with models assuming a more powerful machine even the LW Simulator can extract most of the parallelism from the *parallel* code version. However, since many of the assumptions underlying the more lenient models are unrealistic (e.g., perfect branch prediction in the ORACLE model) the results for these models are less relevant.

There are a few exceptions to the behavior just described, but these cases do not veer from the rule significantly. The three columns of improvements in parallelism in Table 3 report improvement increases (rather than decreases, as expected) for the 100 node graph when moving from BASE to CD model, and for both graphs when moving from SP-CD-MF to the ORACLE model. This appearance of anomaly, minimal as it is, is explained next. Recall that the *improvement* measure compares two quotients (between the number of instructions and the number of parallel cycles), where the nominators of these quotients are different for the two versions of code. Therefore, the ratios of improvement for the different models are not as precise as we would like them to be. To overcome the "noise" which caused this imprecision, Table 5 uses another measure of improvement: the difference in the number of parallel cycles. Note that this measure consistently reflects decrease in

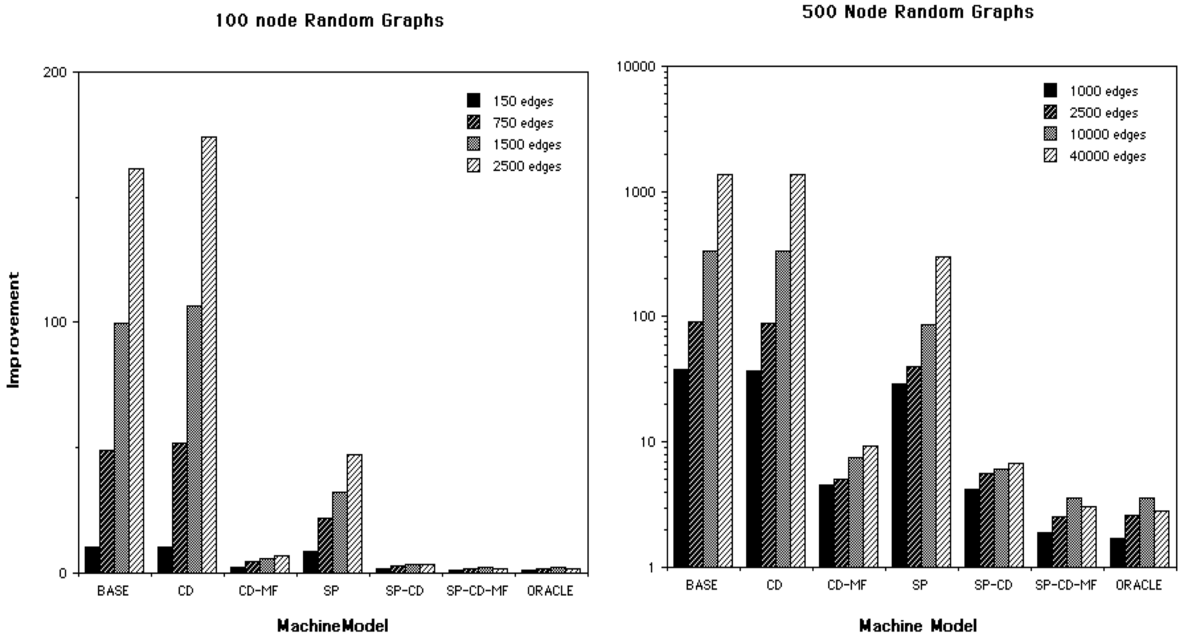

Figure 7: The improvement of the PASC with Enhanced Simulator over the same code with LW Simulator, for 100 and 500 node graphs (the latter in logarithmic scale).

Machine Model	Nodes	Edges	SAC + LW Sim. (1)	PASC + LW Sim. (2)	PASC + Enh. Sim. (3)	Improvements in Number of Cycles		
						(1)–(2)	(2)–(3)	(1)–(3)
BASE	100	750	21246	18388	377	2858	18011	20869
CD			18988	16631	321	2357	16310	18667
CD-MF			2722	500	112	2222	388	2610
SP			3734	2750	126	984	2624	3608
SP-CD			1721	250	87	1471	163	1634
SP-CD-MF			1003	130	69	873	61	934
ORACLE			1003	96	48	907	48	955
BASE	500	2500	69949	63591	703	6358	62888	69246
CD			61390	57533	647	3857	56886	60743
CD-MF			5524	1752	350	3772	1402	5174
SP			12383	11694	294	689	11400	12089
SP-CD			4521	1156	204	3365	952	4317
SP-CD-MF			3625	498	196	3127	302	3429
ORACLE			3505	337	130	3168	207	3375

Table 5: Comparison of number of parallel cycles required for SAC and PASC versions of code with the LW Simulator and for the PASC version with the Enhanced Simulator, for random graphs

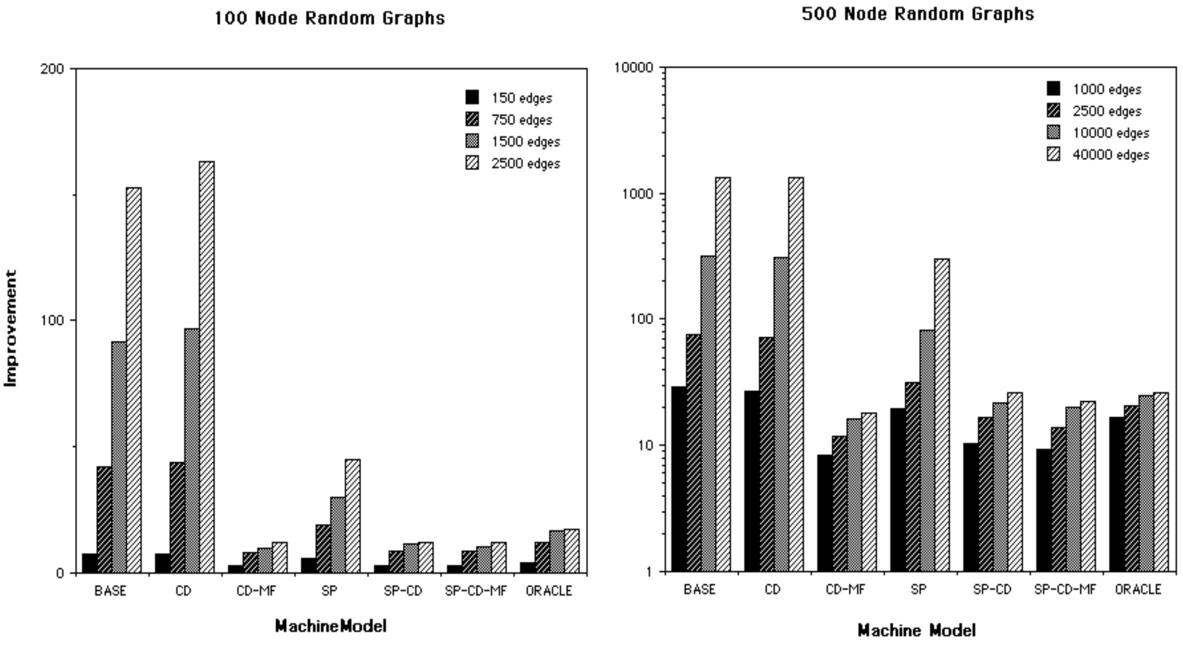

Figure 8: The improvement of the PASC with Enhanced Simulator over SAC with LW Simulator, for 100 and 500 node random graphs (the latter in logarithmic scale).

improvements as the models get more lenient.

From Table 4 and Figure 6 we can observe that the larger the size of the layers in the BFS (as a result of more edges in the random graph), the greater the advantage obtained by the Enhanced Simulator. This is simply because visiting all the nodes in a layer in parallel, and having more nodes in a layer, implies that more parallelism is found.

6.3 Comparing the Sequential Algorithm on a Sequential Machine to the Parallel Algorithm on Our Envisioned System

Column *(3)* in Table 3 presents the amount of parallelism found in our simulations of the Envisioned system. The improvement of the Envisioned system over the sequential system is actually cumulative with respect to the improvements described in the preceding two sections – the improvement of the parallel algorithm (PASC) over the sequential algorithm (SAC) and the improvement of the Enhanced simulator over the LW simulator. This means that in Table 3, column (3)/(1) is the product of columns (2)/(1) and (3)/(2). Similar to its two components, this overall improvement generally decreases with stronger machine models and increases with larger layer sizes. Figure 8 illustrates the overall improvement graphically.

Machine Model	Nodes	Edges	Queuing		Improvement
			Yes	No	
BASE	100	750	57.80	64.39	1.11
CD			66.51	75.62	1.14
CD-MF			159.70	216.74	1.36
SP			145.36	192.66	1.33
SP-CD			188.18	279.02	1.48
SP-CD-MF			218.69	351.81	1.61
ORACLE			315.26	505.73	1.60
BASE	500	2500	95.37	123.59	1.30
CD			97.40	134.29	1.38
CD-MF			122.37	248.24	2.03
SP			150.84	295.53	1.96
SP-CD			178.41	425.91	2.39
SP-CD-MF			181.39	443.29	2.44
ORACLE			273.22	668.35	2.45

Table 6: Comparison of Amounts of parallelism extracted from PASC version of code with the Enhanced simulator, with or without queuing of concurrent writes, for random graphs

6.4 How Much is Lost by Queuing Concurrent Writes

Table 6 shows results of running the Enhanced simulator using the PASC version of code, with and without queuing of concurrent writes. Recall that without queuing of concurrent writes this is our envisioned system. As the results were obtained under the assumption that a number of threads can execute concurrently, they also apply to multiprocessor parallel machines. For the tests shown here we see that eliminating the practice of queuing concurrent writes can improve parallelism by at least 11%, and the improvement ranges up to 145%. The improvement may exceed 100% even when considering reasonable machine models such as CD-MF.

Recall the standard way of looking at the execution as a DAG, whose nodes are instructions and edges are dependences, as described in Section 2. Queuing a concurrent write of n accesses amounts to having a path of n nodes in the execution DAG since there will be an edge connecting every two accesses (or nodes) in the DAG. Without queuing, these same n nodes will form a "star" in the execution DAG where $n-1$ edges will lead from the node representing the first access to all other $n-1$ accesses. So, a path of n nodes (with queuing) corresponds to a path of 2 nodes (without queuing). The impact on the number of parallel cycles in the execution depends on the effect of these n-node paths on the overall depth of the DAG. With respect to the abstract machine models, the impact of queuing on depth of DAGs is greater for the more lenient models, since there is an overall decrease in the depth of DAGs due to factors other than concurrent-writes. This can be observed in Table 6.

Edges	Queuing		Improvement
	Yes	No	
150	47.03	49.22	1.05
750	188.18	279.02	1.48
1500	339.26	625.21	1.84
2500	512.98	996.64	1.94

Table 7: Comparison of amounts of parallelism extracted from PASC version of code with the Enhanced simulator, with or without queuing of concurrent writes, for the SP-CD model with random graphs of 100 nodes over varying numbers of edges.

The more accesses there are to the same memory location in the concurrent-writes, the greater will be the improvement in parallelism without queuing over the parallelism with queuing. In the *bfs* program, more threads will be accessing the variables for a node in parallel as the number of edges increases in the random input graphs. Such increase in the improvement by eliminating queuing is indeed reflected in Table 7.

7 Conclusion

Our paper began with a statement of two apparent paradoxes in some current systems and approaches: the Serialize-to-Parallelize Paradox, and the Paradox of Queuing Concurrent-Writes. We suggest a system which would allow the programmer to express parallelism explicitly in a program, even when programming for uniprocessors. The compiler and hardware of this system would be capable of using the programmer's directives to increase Instruction Level Parallelism in the execution.

Our experimental results quantify the extent to which such an envisioned system can improve parallelism with respect to a test algorithm. Considerable improvements have been observed. We chose Breadth First Search as a test algorithm. We showed that even for existing systems, running strictly sequential code, we can significantly improve ILP by identifying parallelism in the algorithm and designing the code so that it reveals this parallelism. The current paper enforces the call in [25] to start immediately including the topic of parallel algorithms and parallel algorithmic thinking in the standard curriculum for computer science and engineering undergraduates, since this curriculum should expose them to principles which they need to command as they graduate as well as over a life time career. The benefit from parallel algorithms (and parallel programming), as demonstrated in this paper, is expected to increase in main stream computer systems in the near future.

The emulation of the envisioned system was done in two modes – one that emulates queuing of concurrent writes and the other which avoids queuing. Comparing the results for these two modes show that un-queuing the updates can sometimes improve parallelism by over 100%, even for relatively weak abstract machine models.

What Next? It will be interesting to study whether parallel programming of other algorithms would also result in similar improvements in parallelism. We should also remember that by picking a test algorithm rather than a test problem, we ignored altogether the opportunity of switching to an alternative algorithm for the same problem. The rich literature on parallel algorithms suggests that we could expect even bigger improvements in parallelism by switching to an alternative (parallel) algorithm, but this also should be experimentally demonstrated. Basically, our results rely on the same compiler and instruction-code language as Lam-Wilson's work. Other platforms should also be considered.

A Appendix

We present here the full numerical results, some of which were presented so far only in graphical form. Table 8 presents the results for random graphs of 100 nodes, with 150, 750, 1500 or 2500 edges. Table 9 presents the results for random graphs of 500 nodes, with 1000, 2500, 10000 or 40000 edges. The columns marked *(3)*, or PASC+queuing, hold results for running the Enhanced simulator, with queuing of concurrent writes, over the PASC version.

The random graphs used as input to BFS were all constructed using The Stanford GraphBase [15], which generated them by choosing the endpoints for each edge uniformly at random from the set of nodes.

Num. of Edges	Machine Model	SAC + LW Sim. (1)	PASC + LW Sim. (2)	PASC + queuing (3)	PASC + Enh. Sim. (4)	Improvements				
						(2)/(1)	(3)/(1)	(4)/(1)	(4)/(2)	(4)/(3)
150	BASE	2.19	1.66	15.56	16.76	0.76	7.11	7.65	10.10	1.08
	CD	2.61	1.87	17.35	18.94	0.72	6.65	7.26	10.13	1.09
	CD-MF	12.46	15.62	31.94	36.18	1.25	2.56	2.90	2.32	1.13
	SP	6.27	4.08	34.55	35.42	0.65	5.51	5.65	8.68	1.03
	SP-CD	15.53	30.02	47.03	49.22	1.93	3.03	3.17	1.64	1.05
	SP-CD-MF	20.12	45.76	52.25	54.97	2.27	2.60	2.73	1.20	1.05
	ORACLE	20.73	86.38	86.38	86.38	4.17	4.17	4.17	1.00	1.00
750	BASE	1.52	1.32	57.80	64.39	0.87	38.03	42.36	48.78	1.11
	CD	1.72	1.46	66.51	75.62	0.85	38.67	43.97	51.79	1.14
	CD-MF	26.59	48.55	159.70	216.74	1.83	6.01	8.15	4.46	1.36
	SP	10.17	8.83	145.36	192.66	0.87	14.29	18.94	21.82	1.33
	SP-CD	32.39	97.10	188.18	279.02	3.00	5.81	8.61	2.87	1.48
	SP-CD-MF	41.43	186.73	218.69	351.81	4.51	5.28	8.49	1.88	1.61
	ORACLE	42.31	252.86	315.26	505.73	5.98	7.45	11.95	2.00	1.60
1500	BASE	1.36	1.25	102.73	124.33	0.92	75.54	91.42	99.46	1.21
	CD	1.52	1.38	118.60	147.36	0.91	78.03	96.95	106.78	1.24
	CD-MF	43.57	74.68	248.66	420.82	1.71	5.71	9.66	5.63	1.69
	SP	11.23	10.43	262.07	334.08	0.93	23.34	29.75	32.03	1.27
	SP-CD	53.55	177.19	339.26	625.21	3.31	6.34	11.68	3.53	1.84
	SP-CD-MF	68.50	331.55	377.28	717.46	4.84	5.51	10.47	2.16	1.90
	ORACLE	69.96	470.59	568.38	1151.71	6.73	8.12	16.46	2.45	2.03
2500	BASE	1.30	1.23	147.18	198.20	0.95	113.22	152.46	161.14	1.35
	CD	1.44	1.35	166.50	234.90	0.94	115.62	163.12	174.00	1.41
	CD-MF	62.20	104.75	410.38	742.18	1.68	6.60	11.93	7.09	1.81
	SP	11.78	11.25	403.27	532.56	0.96	34.23	45.21	47.34	1.32
	SP-CD	81.78	282.45	512.98	996.64	3.45	6.27	12.19	3.53	1.94
	SP-CD-MF	104.61	664.43	591.23	1245.80	6.35	5.65	11.91	1.87	2.11
	ORACLE	106.84	942.77	690.74	1835.92	8.82	6.47	17.18	1.95	2.66

Table 8: Comparison of Amounts of parallelism extracted for SAC and PASC versions of code with the LW Simulator and for the PASC version with the Enhanced Simulator, for 100 node random graphs

Num. of Edges	Machine Model	SAC + LW Sim. (1)	PASC + LW Sim. (2)	PASC + queuing (3)	PASC + Enh. Sim. (4)	Improvements				
						(2)/(1)	(3)/(1)	(4)/(1)	(4)/(2)	(4)/(3)
1000	BASE	2.04	1.56	47.01	59.17	0.76	23.04	29.00	37.93	1.26
	CD	2.40	1.74	49.76	63.86	0.72	20.73	26.61	36.70	1.28
	CD-MF	13.72	25.01	69.35	114.91	1.82	5.05	8.38	4.59	1.66
	SP	6.74	4.46	91.49	131.06	0.66	13.57	19.45	29.39	1.43
	SP-CD	16.82	41.30	110.12	174.90	2.46	6.55	10.40	4.23	1.59
	SP-CD-MF	21.18	103.42	118.05	195.78	4.88	5.57	9.24	1.89	1.66
	ORACLE	21.79	216.25	216.25	365.95	9.92	9.92	16.79	1.69	1.69
2500	BASE	1.64	1.37	95.37	123.59	0.84	58.15	75.36	90.21	1.30
	CD	1.87	1.51	97.40	134.29	0.81	52.09	71.81	88.93	1.38
	CD-MF	20.77	49.59	122.37	248.24	2.39	5.89	11.95	5.01	2.03
	SP	9.27	7.43	150.84	295.53	0.80	16.27	31.88	39.78	1.96
	SP-CD	25.38	75.16	178.41	425.91	2.96	7.03	16.78	5.67	2.39
	SP-CD-MF	31.65	174.47	181.39	443.29	5.51	5.73	14.01	2.54	2.44
	ORACLE	32.74	257.82	273.22	668.35	7.87	8.35	20.41	2.59	2.45
10000	BASE	1.32	1.23	291.49	416.36	0.93	220.83	315.42	338.50	1.43
	CD	1.47	1.36	297.34	453.90	0.93	202.27	308.78	333.75	1.53
	CD-MF	56.09	120.67	392.58	915.18	2.15	7.00	16.32	7.58	2.33
	SP	11.62	10.95	411.50	942.73	0.94	35.41	81.13	86.09	2.29
	SP-CD	68.51	245.54	472.15	1507.35	3.58	6.89	22.00	6.14	3.19
	SP-CD-MF	85.66	481.02	498.89	1729.29	5.62	5.82	20.19	3.60	3.47
	ORACLE	88.37	607.49	633.43	2168.27	6.87	7.17	24.54	3.57	3.42
40000	BASE	1.22	1.20	967.97	1628.63	0.98	793.42	1334.94	1357.19	1.68
	CD	1.34	1.31	1018.09	1778.67	0.98	759.77	1327.37	1357.76	1.75
	CD-MF	197.34	380.73	1793.69	3539.55	1.93	9.09	17.94	9.30	1.97
	SP	12.59	12.40	1815.15	3778.88	0.98	144.17	300.15	304.75	2.08
	SP-CD	241.04	925.78	2136.55	6246.26	3.84	8.86	25.91	6.75	2.92
	SP-CD-MF	302.11	2175.95	2708.84	6678.40	7.20	8.97	22.11	3.07	2.47
	ORACLE	310.90	2941.45	3077.87	8168.19	9.46	9.90	26.27	2.78	2.65

Table 9: Comparison of Amounts of parallelism extracted for SAC and PASC versions of code with the LW Simulator and for the PASC version with the Enhanced Simulator, for 500 node random graphs

References

[1] ANSI. ANSI Fortran draft S8, version 111.

[2] T. M. Austin and G. S. Sohi. Dynamic dependency analysis of ordinary programs. In *19th Annual International Symposium on Computer Architecture*, 1992.

[3] G. E. Blelloch, S. Chatterjee, J. C. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *4th Symposium on Principles and Practice of Parallel Programming*. ACM SIGPLAN, 1993.

[4] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8(2):119–134, 1990.

[5] D.-K. Chen, H.-M. Su, and P.-C. Yew. The impact of synchronization and granularity on parallel systems. In *17th Annual International Symposium on Computer Architecture*, pages 239–248, 1990.

[6] D. M. Eckstein. *Parallel Processing using Depth-First Search and Breadth-First Search.* PhD thesis, Computer Science Department, University of Iowa, Iowa City, IA, 1977.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its uses in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[8] P. B. Gibbons, Y. Matias, and V. Ramachandran. The QRQW PRAM: Accounting for contention in parallel algorithms. In *Proceedings of the Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 638–648, 1994.

[9] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach.* Morgan Kaufman Publishers Inc., San Mateo, California, 1990.

[10] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*, page 346. Morgan Kaufman Publishers Inc., San Mateo, California, second edition, 1994. Uncorrected preliminary manuscript.

[11] High Performance Fortran Forum. High performance Fortran language specification, 1993.

[12] W. W. Hwu, T. M. Conte, and P. P. Chang. Comparing software and hardware schemes for reducing the cost of branches. In *13th International Symposium on Computer Architecture*, pages 224–233, 1989.

[13] J. JáJá. *An Introduction to Parallel Algorithms.* Addison Wesley Publishing Company, 1992.

[14] R. B. Jones and V. H. Allan. Software pipelining: An evaluation of enhanced pipelining. In *Proceedings of the 24th International Symposium on Microarchitecture*, pages 82–92, 1991.

[15] D. E. Knuth. *The Stanford GraphBase.* Addison-Wesley Publishing Company, New York, NY, 1993.

[16] S. M. Krishnamurthy. Ultra-pipelining: An efficient software pipelining algorithm. In *COMPCON Spring '93*, pages 586–591, 1993.

[17] M. Kumar. Measuring parallelism in computation-intensive scientific/engineering applications. *IEEE Transactions on Computers*, 37(9):1088–1098, 1988.

[18] M. Lam. Software pipelining: An effective scheduling technique for vliw machines. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 318–328. ACM, 1988.

[19] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *19th Annual International Symposium on Computer Architecture*, pages 46–57, 1992.

[20] S. McFarling and J. Hennessy. Reducing the cost of branches. In *13th Annual International Symposium on Computer Architecture, Tokyo*, pages 396–403, 1986.

[21] J. R. Rose and G. L. Steele, Jr. C*: An extended c language for data parallel programming. In *Proceedings Second International Conference on Supercomputing*, volume 2, pages 2–16, 1987.

[22] Y. Shiloach and U. Vishkin. An $O(logn)$ parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–56, 1982.

[23] Y. Shiloach and U. Vishkin. An $O(n^2logn)$ parallel max-flow algorithm. *Journal of Algorithms*, 3(2):128–146, 1982.

[24] M. D. Smith. Tracing with *pixie*. Version 1.1, Stanford University, April 1991.

[25] U. Vishkin. Can parallel algorithms enhance serial implementation? (extended abstract). In *8th International Parallel Processing Symposium.* IEEE, 1994.

[26] D. W. Wall. Limits of instruction-level parallelism. RR 93/6, Digital Western Research Laboratory, Palo Alto, California, November 1993.

[27] S. Weiss and J. E. Smith. *Power and PowerPC.* Morgan Kaufman Publishers Inc., San Francisco, CA, 1994.