Performance Modeling and Measurement of Parallelized Code for Distributed Shared Memory Multiprocessors

Abdul Waheed* and Jerry Yan*

NAS Parallel Tools Group
MRJ Technology Solutions
NASA Ames Research Center
Mail Stop T27A-2
Moffett Field, CA 94035-1000
waheed@nas.nasa.gov

yan@nas.nasa.gov

NAS-98-012
March 1998

Abstract

This paper presents a model to evaluate the performance and overhead of parallelizing sequential code using compiler
directives for multiprocessing on distributed shared memory (DSM) systems. With increasing popularity of shared
address space architectures, it is essential to understand their performance impact on programs that benefit from shared
memory multiprocessing. We present a simple model to characterize the performance of programs that are parallelized
using compiler directives for shared memory multiprocessing. We parallelized the sequential implementation of NAS
benchmarks using native Fortran77 compiler directives for an Origin2000, which is a DSM system based on a
cache-coherent Non Uniform Memory Access (ccNUMA) architecture. We report measurement based performance of
these parallelized benchmarks from four perspectives: efficacy of parallelization process; scalability; parallelization
overhead; and comparison with hand-parallelized and -optimized version of the same benchmarks. Our results indicate
that sequential programs can conveniently be parallelized for DSM systems using compiler directives but realizing
performance gains as predicted by the performance model depends primarily on minimizing architecture-specific data
locality overhead.

PDF--147.5 Kbytes
PostScript--316.8 Kbytes

Performance Modeling and Measurement of Parallelized Code for Distributed Shared Memory Multiprocessors

Abdul Waheed and Jerry Yan[†]

NAS Technical Report NAS-98-012 March 1998

{waheed,yan}@nas.nasa.gov
NAS Parallel Tools Group
NASA Ames Research Center
Mail Stop T27A-2
Moffett Field, CA 94035-1000

Abstract

This paper presents a model to evaluate the performance and overhead of parallelizing sequential code using compiler directives for multiprocessing on distributed shared memory (DSM) systems. With increasing popularity of shared address space architectures, it is essential to understand their performance impact on programs that benefit from shared memory multiprocessing. We present a simple model to characterize the performance of programs that are parallelized using compiler directives for shared memory multiprocessing. We parallelized the sequential implementation of NAS benchmarks using native Fortran77 compiler directives for an Origin2000, which is a DSM system based on a cache-coherent Non Uniform Memory Access (ccNUMA) architecture. We report measurement based performance of these parallelized benchmarks from four perspectives: efficacy of parallelization process; scalability; parallelization overhead; and comparison with hand-parallelized and -optimized version of the same benchmarks. Our results indicate that sequential programs can conveniently be parallelized for DSM systems using compiler directives but realizing performance gains as predicted by the performance model depends primarily on minimizing architecture-specific data locality overhead.

† Both authors are employees of MRJ Technology Solutions, Inc., NASA Contract NAS2-14303, Moffett Field, CA 94035-1000

1 Introduction

Distribute Shared Memory (DSM) systems are becoming popular in high performance computing because they offer ease of programming due to a global address space and scalability to large number of nodes. Although DSM systems facilitate programming, they can potentially introduce performance bottlenecks that require additional effort on the part of a user to discover and eliminate [20]. Non Uniform Memory Access (NUMA) architectures can incur orders of magnitude greater latencies to access data that reside farther from the processor in memory hierarchy [11]. These systems often use cache-based commodity processor with cache coherence implemented in hardware to hide latency. Memory traffic generated by protocols that keep the caches coherent is another potential source of performance degradation. While the developers of compilation and parallelization tools for shared memory systems have addressed some of these problems, extensive user input is still required to fully benefit from these tools [2,3,10,16]. Understanding the sources of parallelism in a program and potential overhead due to subtleties of a DSM architecture is essential for effectively using these systems.

Due to the growing disparity between processor and memory speeds, tool developers have been focusing on measurement-based tools to analyze memory performance. Several state-of-the-art microprocessors provide on-chip performance counters to facilitate these measurements [20]. However, most of the existing tools and techniques are limited to evaluating cache and memory performance for a single processor [19]. These tools typically do not directly address multiprocessor memory performance issues. There are examples of research prototype DSM systems that can support memory performance measurements across multiprocessor nodes [7]. Unfortunately, such tools are not yet widely available for commercial multiprocessors. We present a performance model that accounts for inherent parallelism in a program, which can result in potential speedup as well as overhead when that program is executed on a DSM system. This model can be used to analyze the efficacy of parallelization and quantitatively measure the overhead of parallelizing a program. Quantitative evaluation of this overhead provides an indirect measure of effective utilization of available memory subsystem performance.

In this paper, we present a performance model to characterize the execution of a compiler directives-based parallelized program. We subsequently apply this model to evaluate the performance of our parallelized version of NAS benchmarks on SGI Origin2000, which is a commercial DSM system with a ccNUMA architecture. Each node of the system consists of two MIPS R10000 processors with two levels of separate data and instruction caches for each processor; and 4GB of main memory shared between two processors

on a node. Multiple system nodes are connected in a hypercube topology through a high speed network. We used native tools to parallelized the sequential implementation of NPBs [14]. These tools include: *Power Fortran Accelerator* (PFA), which can automatically insert parallelization directives in sequential code and transform the loops to enhance their performance; *Parallel Analyzer View* (PAV), which can annotate the results of dependence analysis of PFA and present them graphically; and *Fortran77* compiler with MP runtime library to compile and executed the parallelized code [13]. In addition to using these tools, we inserted some directive by hand to assist the compiler and improve the performance.

We explain the directives-based parallelization paradigm in Section 2. A performance model and metrics to evaluate different aspects of a directives-based parallelized program are presented in Section 3. Section 4 reports detailed measurement based evaluation of the parallelized NAS benchmarks using performance model and metrics of Section 2. We briefly survey the related research efforts in Section 5 and conclude in Section 6.

2 Compiler-Directed Parallelism

Compiler-directed parallelism has been traditionally used for vector supercomputers. It has recently started attracting attention of mainstream vendors due to increasing popularity of Symmetric Multiprocessing (SMP) systems. Parallelization directives can be inserted in legacy sequential code to tap the multiprocessing potential of an SMP architecture. These directives are in the form of special comments that are ignored by a compiler without appropriate multiprocessing flag. Thus, there is no need to maintain separate sequential and parallelized versions of the same code. There is an ongoing effort of standardizing these directives to port programs across different SMP platforms [15].

Potential parallelism of a DSM system can be exploited in one of three ways: message-passing: use of data-parallel languages: or compiler-directed multiprocessing. *Message-passing* provides the user with explicit control over communication and synchronizations through commonly used message-passing libraries [12]. *Data-parallel programming languages* allow the users to write SPMD programs without explicit message-passing, which is handled by the compiler and its runtime system. The main source of parallelism is the program data, which can be distributed among different processors through compiler directives. High Performance Fortran (HPF [6]) is a standard for these directives that have been used by several compiler developers. Both message-passing and data-parallelism force a user to develop a parallel algorithm, which is a challenging task. Due to the simplicity of programming shared memory systems, compiler developers have been investigating different techniques to exploit parallelism directly by the compilers for such systems. This process can be accomplished automatically with a compiler or through some hints provided by the user to the compiler [15].

Before inserting compiler directives in sequential code, one has to identify parts of the program that can be parallelized without affecting the correctness. The main source of parallelism is the loops whose iterations can be scheduled on multiple processors without any data access dependence or conflicts among different iterations. This requires dependence analysis for every loop nest of source code. For a given loop nest, it is customary to parallelize the outer-most loop to have significant work for each set of iterations that are scheduled on multiple processors. The user may have to modify some loop nests to resolve dependences on the outer-most loop index to parallelize the loop. If there are data dependencies between different iterations of the outer loop, parallelization is inhibited to preserve correctness of the program. As illustrated in Figure 1, this parallelization is an iterative process, which continues until most of the loops contributing to the overall execution time are parallelized. Finally, the parallelized code is compiled and linked using with appropriate runtime libraries to execute on a target multiprocessor.

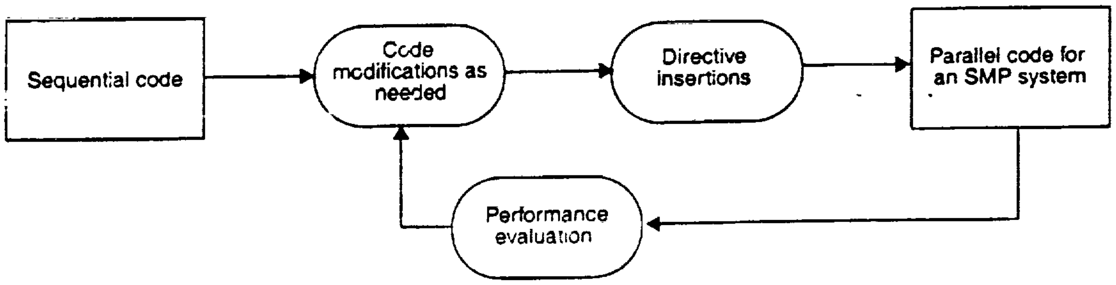


Figure 1. General methodology of parallelizing sequential code using compiler directives for shared memory multiprocessing.

Compared to the process-level parallelism for message-passing programs, directive-based parallelism constitutes a finer-grained, loop-level parallelism. Figure 2 provides an example of this parallelism implemented through MIPS Fortran compiler directives for multiprocessing [13]. The C$DOACROSS directive instructs the compiler to divide the outer loop iterations equally among the available processors. This is the default loop scheduling, which is implemented by the runtime system until specifically instructed otherwise by additional compiler directives. Data distribution directive, C$DISTRIBUTE works at the level of memory pages rather than array elements in data-parallel languages. Thus, the data distribution is relatively coarse-grain.

Directives-based parallelism is supported by the MP runtime library on Origin2000, which implements a

```
      integer i, j, k
      double precision temp
      double precision a(256,256,256), b(256,256,256)
c$distribute a(*,*,BLOCK)
c$distribute b(*,*,BLOCK)
c$doacross local(k,j,i,temp)
      do k = 1, 254
         do j = 0, 255
            do i = 1, 255
               tmp = 1.0d+00 / a(i,j,k)
               b(i,j,k) = a(i,j,k) * tmp
            enddo
         enddo
      enddo
```

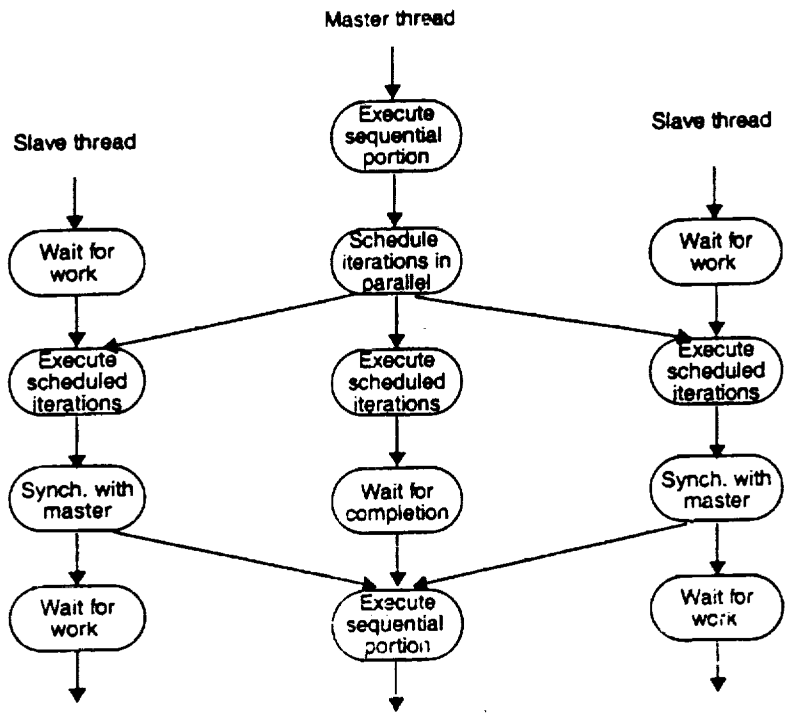
Figure 2. An example of instruction-level parallelism using MipsPro Fortran77 compiler directives for multiprocessing.

fork-and-join paradigm of parallelism. A *master thread* initiates the program, creates multiple *slave threads*, schedules the iterations of parallelized loops on all the threads including itself, waits for the completion of a parallel loop by all the slave threads, and executes sequential potions of the program. Slave threads wait for work (i.e., for parts of parallel loops) when the master thread executes a sequential portion of the code. Figure 3 represents this runtime system graphically. Clearly, the main disadvantage of this type of parallelism is the overhead to synchronize different threads that execute different iterations of a loop.

Considering ease of programming, directives-based parallelism has clear advantages over message-passing and data-parallelism. However, performance impact of using this programming style on a DSM system is a relatively unexplored area. We focus on performance evaluation of directives-based parallelized programs in subsequent sections.

3 Performance Model and Metrics

Compared to message-passing and data-parallelism, compiler-directed parallelism is comparatively fine-grained. Parallelism is discovered from the loops in sequential program whose iterations can be scheduled on multiple processors. It is simpler to quantify the amount of parallelism that has been discovered in a directives-based parallelized program. Based on these initial measurements, we can estimate the performance with multiple processors under ideal conditions of utilization. We use these estimates to quantify the overhead of directives-based parallelization techniques that is otherwise hidden from the user. This analysis helps the user to decide whether or not a locality optimization effort will be useful. We first



Master thread
Slave thread
Slave thread
Execute sequential portion
Wait for work
Schedule iterations in parallel
Wait for work
Execute scheduled iterations
Execute scheduled iterations
Execute scheduled iterations
Synch. with master
Wait for completion
Synch. with master
Wait for work
Execute sequential portion
Wait for work

Figure 3. Execution of a parallel loop using fork-and-join paradigm with three threads.

explain the performance model with respect to DSM system architecture that we are focusing on. Subsequently, we define metrics to evaluate parallelization and scalability of the parallelized code.

3.1 Performance Model

Consider a sequential program consisting of N blocks, such that only one block is executed at any time. Unless otherwise indicated, we shall use the term *block* interchangeably with *subroutine*. This is true for most of the programs developed in a structured manner. The *sequential execution time* of the program is denoted by T_s and is calculated as:

$$T_s = \sum_{i=1}^{N} t_i, \tag{1}$$

where t_i is the execution time spent in the i-th block. We have to measure the aggregate time spent in every block of the code that substantially contributes toward the overall sequential execution time. Therefore, we define the *sequential cost* for executing the i-th block as a fraction:

$$SC_i = \frac{t_i}{T_s}. \tag{2}$$

When a program is executed in parallel using fork-and-join paradigm, synchronization overhead is incurred by slave threads to wait for parallel work and by the master thread to wait for all the slave threads to finish executing a particular parallel loop. The execution time of a directives-based parallelized program is denoted by T_p and is given by:

$$T_p = \sum_{i=1}^{N} t_i + t_o = \sum_{i=1}^{N} (tp_i + ts_i) + t_o = \sum_{i=1}^{N} tp_i + \sum_{i=1}^{N} ts_i + t_o, \tag{3}$$

where the (useful) execution time spent in the i-th block (t_i) is the sum of time spent in parallelized loops of that block (tp_i) and the remaining sequential code of that block (ts_i). *Parallelization overhead* for the entire program is given by t_o because it is non-trivial to measure it for each individual parallelized block of the program using profiling. Considering the architecture of a ccNUMA-based DSM system, parallelization overhead is an intricate function of following factors:

1. aggregate synchronization time between threads during execution of a parallelized program;
2. number of parallel loops;
3. aggregate load imbalance between threads during execution of a parallelized program;
4. non-local memory accesses by each thread; and
5. resource contention between a thread and other users on the system.

While the first four factors may not change from one execution to another, the resource contention due to other users of the system affects in an unpredictable manner. Since directives-based parallelized programs rely on access to shared data structures for synchronization as well as computations requiring non-local data, they are particularly susceptible to the contention from other users. Quantitative calculation of parallelization overhead and other metrics are presented in the following subsection.

3.2 Performance Metrics

Consider that a subroutine j in the program has K parallelized loops. Then we define the metric *parallel coverage* of subroutine j as:

$$PC_j = \frac{\sum_{i=1}^{K} tp_i}{T_s}. \tag{4}$$

Note that parallel coverage of a subroutine can be determined by profiling the execution of a sequential program. This technique is often used to determine the fraction of code that can be executed in parallel [4]. The total parallel coverage of a parallelized program is equal to the sum of parallel coverages of all subroutines in the program. If there are L subroutines in a program, then the parallel coverage of the entire program is calculated as:

$$PC = \sum_{j=1}^{L} PC_j . \qquad (5)$$

A value of PC close to 1.0 (or 100%, if expressed as a percentage) will be an ideal value for a parallelized program indicating that there is no sequential code and no parallelization overhead. Therefore, executing such a program on n processors should result in a speedup of n, provided that all the processors are fully utilized during the entire execution. A higher value of this metric is desirable because it represents a better parallelization of sequential code.

Amdahl's law based on fixed workload can be used as a measure of scalability of the parallelized code under fork-and-join execution model. According to Amdahl's law if a is the sequential fraction of a program, the maximum possible speedup that can be obtained on an n processor system is given by:

$$S_n = \frac{1}{a + \frac{1-a}{n}} = \frac{n}{1 + a(n-1)} . \qquad (6)$$

where a is the fraction of serial portion of the code. Noting that parallel coverage $PC=1-a$, we can express ideal speedup according to Amdahl's law as:

$$S_n = \frac{n}{PC + n(1 - PC)} . \qquad (7)$$

Using this definition of theoretical speedup, we can now calculate the combined value of parallelization overhead as:

$$t_o = T_p - \sum_{i=1}^{N} (tp_i + ts_i) = T_p - \frac{T_s}{n}(PC + n(1 - PC)) . \qquad (8)$$

where T_p is the measured execution time on n processors.

Parallel coverage and speedup metrics defined by equations (5) and (7), respectively for independent assessment of a directives-based parallelized program. In order to compare the performance of a directive-based parallelized program with the same program parallelized using a different technique, we use execution time as a metric. Additionally, equation (8) will be used for evaluating parallelization overhead for directives-based parallelized programs.

4 Performance Evaluation

Performance is evaluated from three perspectives: efficacy of parallelization process; scalability of parallelized programs; and performance comparison of directives-based parallelized program against the hand-parallelized and optimized code. The metrics discussed in Section 3.2 are used for this evaluation.

4.1 Analysis of Parallelization

Parallel coverage is defined in Section 3.2 as a metric to represent the efficacy of parallelization process. This metric was calculated for all NAS benchmarks parallelized using compiler directives for shared memory multiprocessing. For these calculations, the benchmarks are compiled with instrumentation to measure the time spent in each subroutine that contains parallel code blocks. We execute these programs on a single processor of Origin2000.

Table 1 presents detailed measurements related to parallel coverage obtained in BT. Sequential implementation of BT contains a number of modular subroutines that solve Navier-Stokes equations using a Block Tridiagonal algorithms. An inspection of these subroutines indicates that most of this algorithm contains sufficient parallelism. Quantitatively, these measurements indicate that the code responsible for more than 99% of the entire execution time is parallelized. This level of parallelism was attained after iteratively analyzing the source code and discovering possibilities of parallelization by minor modifications in some loop nests.

The same measurement procedure was repeated to calculate parallel coverages for FT, CG, and MG benchmarks. A summary of these calculations is reported in Table 2. Unlike BT, we relied on native SGI tools (PFA and PAV) to parallelize these benchmarks. Furthermore, we had to manually perform inter-procedural analysis to parallelize a few important loops in FT.

Table 2. Parallel coverage of FT, CG, and MG benchmarks.

Benchmark	Execution time (sec)	Execution time for parallel blocks (sec)	Parallel Coverage (%)
FT	203.70	200.15	98.26
CG	50.65	48.49	95.75
MG	96.93	90.56	93.43

The results shown in Table 2 suggest that 93%–99% of the code is parallelized. It should be noted that

Table 1. Parallelization statistics obtained from measurements of BT on an Origin2000 node. Sequential cost and parallel coverage is expressed as a percentage of total execution time, which is 2723.96 sec for this particular execution.

Subroutines with parallelized code	Sequential overall time (sec)	Execution time for parallel blocks (sec)	Sequential Cost (%)	Parallel Coverage (%)
add	19.05	19.05	0.69	0.69
rhs_norm	0.13	0.13	0	0
exact_rhs	2.31	0.83	0.08	0.03
initialize	6.17	0.19	0.22	0
lhsinit	2.35	2.34	0.08	0.08
lhsx	357.80	357.80	13.79	13.79
lhsy	375.06	375.00	13.76	13.76
lhsz	453.21	453.20	16.63	16.63
compute_rhs	272.46	272.45	10.00	10.00
x_backsubstitute	103.75	103.75	3.80	3.80
x_solve_cell	304.49	304.48	11.17	11.17
y_backsubstitute	106.87	106.40	3.92	3.90
y_solve_cell	306.05	305.00	11.23	11.23
z_backsubstitute	106.87	106.80	3.92	3.92
z_solve_cell	307.25	307.10	11.28	11.27
Total	2723.80	2715.50	99.99	99.69

when a program is 100% parallelized, a linear speedup could be obtained provided that all the processors are equally utilized throughout the execution. This theoretical speedup will be used as a criteria to evaluate the actual performance of parallelized code in the following subsections.

4.2 Analysis of Scalability

Figure 4 presents the scalability characteristics of the four parallelized benchmarks. The ideal execution time values are calculated assuming a linear speedup from sequential execution times. Theoretical execution time values are determined according to speedup obtained from equation (8) in Section 3.2. The speedup is less than ideal or theoretical values for BT and FT. However, CG and MG exhibit close to ideal speedup values. BT and FT are relatively larger programs compared to CG and MG. Additionally, algorithms for BT and FT depend on a regular pattern of data accesses which is not the case for CG and MG [5]. Lack of structured data accesses helps loop-level parallelization paradigm by reducing parallelization overhead unlike message-passing or data-parallelism. Therefore, BT and FT are susceptible to overhead due to data locality as well as synchronization. Since these overhead are not significant for CG and MG due to their structure as well as smaller number of parallelized loops, the speedup is close to ideal.

In fact, CG and MG show better than ideal speedup for some number of processors. This is not unusual for a cache-based DSM system. Ideal or theoretical speedup is determined with respect to sequential execution time, which is constrained by the amount of data that can be kept in caches. With computation and data distributed on multiple cache-based processors of Origin2000, the effective cache size also increases resulting in higher than expected speedup for some executions of CG and MG.

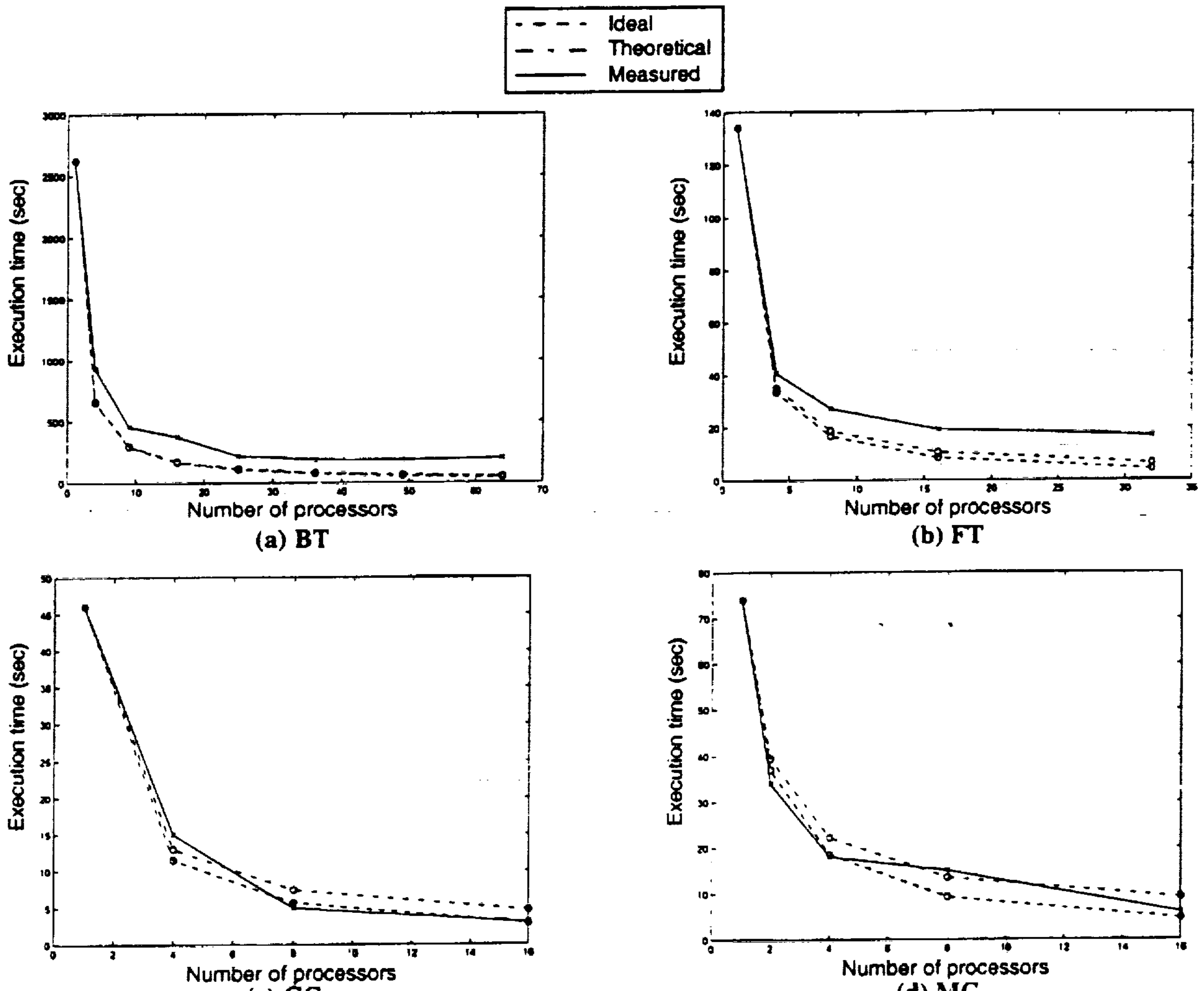


Figure 4. Scalability characteristics of directives-based parallelized programs and their comparisons with ideal and theoretical speedup.

Based on the results of scalability measurements, it can be observed that speedup close to the ideal and theoretical values are attainable by parallelizing programs using directives-based approach. However, the differences from the expected theoretical values of speedup should be expected for larger applications with regular data accesses. In those cases, careful data distribution becomes important to obtain high speedup values. In fact, many argue in favor of using fine-grained data distributions, similar to those used in message-passing programs, in conjunction with shared memory multiprocessing directives to leverage the benefits of both paradigms.

4.3 Parallelization Overhead

Measurement based results presented in Section 4.2 indicate that parallelization overhead is inevitable even when the performance is close to ideal. The overhead stem from the cache-based DSM architecture as well as excessive synchronization to support loop-level parallelization at the runtime. In order to put these overhead in proper perspective, we first present the measured values of parallelization overhead for directives-based parallelized implementation of NAS benchmarks in Section 4.3.1. Then we analyze synchronization overhead using a synthetic loop nest in Section 4.3.2.

4.3.1 Measurement of Parallelization Overhead

Compared to FT. CG. and MG. considerably more time was spent on BT to analyze and tune its performance. Speedup characteristics of BT based solely on its parallelization did not show any appreciable reduction in execution time with increasing number of processors even with close to ideal parallel coverage as discussed in Section 4.1. This is due to the overhead of accessing data not found in caches or local memory. Therefore, all parallelized loops were re-examined and additional directives that enable data distribution at the granularity of pages of memory were inserted. This resulted in significant performance improvement compared to its initial unoptimized implementation. As shown in Figure 4(a), parallelization overhead is small as a result of additional data distribution directives. However, as we know from the speedup characteristics of CG and MG. close to ideal speedup is attainable by removing data locality overhead such that most of the data accesses are limited to the first level caches. We first try to assess the quantitative value of this overhead for BT using equation (8).

Table 3 lists the ideal, theoretical, and measured execution times for BT using multiple processors. Parallelization overhead is presented as a percentage of measured execution time. Clearly, the actual speedup is lower than the expected theoretical maximum value for any number of processors. Note that the parallelization overhead continues to increase with the number of processors and accounts for about 75% of the total execution time with 64 processors. This behavior is an indication of non-optimal data placement that results in non-local data accesses as well as cache coherence traffic. As we mentioned in Section 3.1, it is difficult to quantify the parallelization overhead due to a number of factors that can potentially aggravate it. Although the measurements presented in Table 3 suggest that the bottleneck could be due to data locality overhead, it is practically impossible to isolate its quantitative contribution to overall overhead due to other factors including synchronization and resource contention.

Table 3. Calculation of parallelization overhead of BT on on a range of 1 to 64 nodes of Origin2000.

Number of processors	Ideal execution time (sec)	Theoretical execution time (sec)	Measured execution time (sec)	Parallelization overhead (%)
1	2723	2723	2723	0
4	680	687	931	26.20
9	303	310	455	31.86
16	170	178	374	52.41
25	109	117	216	45.83
36	76	84	186	54.84
49	56	64	182	64.84
64	43	51	198	74.24

Among parallelization overhead, synchronization overhead can be measured using SGI's SpeedShop toolset, which can determine the time spent in synchronization primitives of MP library. This profiling information is obtained using hardware performance counters on MIPS R10000 processors. These measurement based experiments were carried out for BT, FT, CG, and MG using relatively small number of processors. Running such experiments for larger number of processors results in perturbation of the actual program to a point that profiling itself becomes a significant overhead. The results of these experiments are reported in Table 4. Synchronization overhead for each case is obtained as a percentage of measured execution time. Synchronization overhead were as high as 19% in some cases. The last column lists the total parallelization overhead obtained by subtracting measured execution time from the theoretical execution time according to equation (8). In two cases, this calculation is not possible due to better than expected speedup of CG and MG, which is a consequence of untuned sequential versions of these programs as discussed in Section 4.2.

Although the measurements report up to 19% overhead due to synchronization, it is incorrect to assume that synchronization overhead is a result of parallel loop scheduling alone. Synchronization and data locality overhead are strongly correlated with each other. The time that a master thread spends waiting for slaves to finish executing a parallel loop could be due to a combination of two reasons: (1) time to synchronize multiple threads; and (2) load imbalance between master and some of the slave threads due to their non-local data accesses. If resource contention from other users is also considered, the problem of isolating one particular type of overhead becomes even more complex.

Table 4. Parallelization overhead for directives-based parallelized NAS benchmarks.

Benchmarks	Number of processors	Theoretical execution time (sec)	Measured execution time (sec)	Measured synchronization overhead (sec)	Total overhead (sec)
BT	4	804	1053	208 (19.75%)	249 (23.65%)
	9	363	444	80 (17.98%)	81 (18.24%)
FT	4	35.24	39.66	2.62 (6.6%)	4.42 (11.14%)
	8	18.79	23.02	2.37 (10.3%)	4.23 (18.38%)
CG	4	12.97	14.58	2.80 (19.2%)	1.61 (11.04%)
	8	7.46	4.78	0.74 (15.5%)	—
MG	4	22.14	18.41	0.63 (3.4%)	—
	8	13.50	14.92	0.60 (4.0%)	1.42 (9.5%)

4.3.2 Analysis of Loop Synchronization Overhead

Before reaching any conclusions about parallelization overhead, a few simple experiments were carried out to measure synchronization overhead for distributing loop iterations. Code fragment listed in Figure 5 is used to isolate this overhead from any other as much as possible. Note that all variables accessed in this loop nest are labeled "local". We compiled and linked this code without any compiler optimization flags. This guarantees that all data accesses in parallelized loops are from first level of caches without any non-local accesses. Multiple SpeedShop profiling experiments with this code were executed on 4, 8, 9, and 16 processors.

```
    integer i, j, k,l
    double precision uo, u1

  u0 = 1.0
  u1 = 1.0
 c$doacross local(i,j,k,l,u0,u1)
 do l = 1, 128
    do k = 1, 128
      do j = 1, 128
        do i = 1, 128
          u0 = u1+1
        end do
      end do
    end do
  enddo

  end
```

Figure 5. A synthetic program to analyze the synchronization overhead for directives-based parallelized programs.

Figure 6 presents the experimental results. Each bar represents measured synchronization overhead for one execution of the program The total execution time for four processors is about 1.2 seconds. which scales linearly with increasing number of processors. This is consistent with the expected behavior due to a very simple program. The overhead measurements are consistent for smaller number of processor showing a variation in the range of 6%–19%. The 16 processor case shows larger overhead because it is presented as a fraction of total execution time, which is very small in this case. Although we tried to ensure that data locality overhead does not affect the measurements, we cannot isolate the overhead due to resource contention from other users.

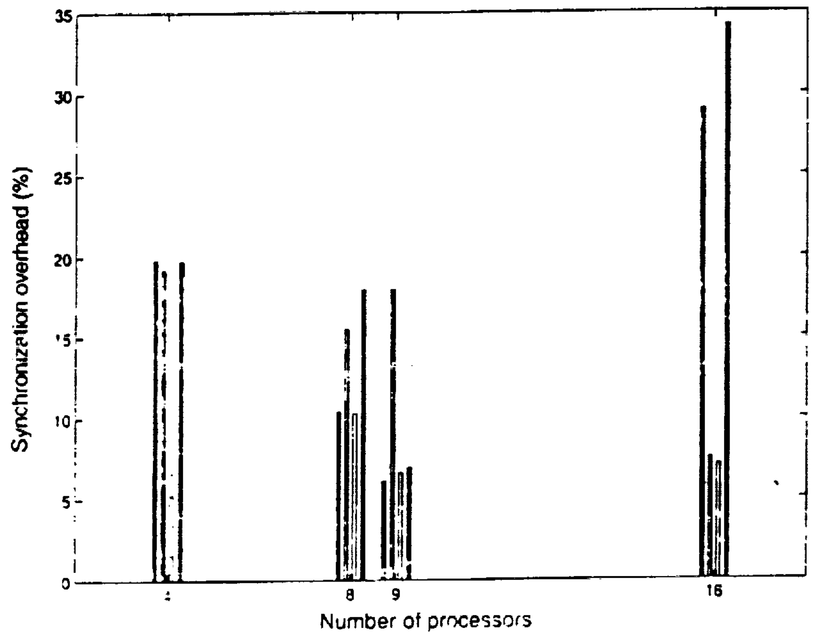


Figure 6. Synchronization overhead for the synthetic loop nest.

Based on the results reported in this subsection, two conclusions can be drawn:

1. Assuming a properly tuned sequential version of a program to calculate accurate values of theoretical speedup, it is possible to calculate the aggregate value of parallelization overhead.
2. It is impractical to quantitatively isolate the impact of different sources of parallelization overhead.

Calculation of aggregate parallelization overhead using the performance model of Section 3 provides useful information to the user. A high value of this overhead, despite near ideal parallel coverage, almost certainly indicates a memory performance bottleneck. Parallelization overhead on a cache-based DSM system will continue to reduce as most of the data is placed closest to the processor in the available memory hierarchy.

4.4 Comparative Performance Analysis

NAS benchmarks were originally written as a suite of paper-and-pencil benchmarks to allow high-performance computing system vendors and researchers to develop their own implementations to evaluate specific architectures of their interest [5]. NAS also provides a hand-parallelized message-passing implementation of the benchmarks based on MPI message-passing library [14]. This implementation is carefully written and optimized for a majority of existing high performance computing platforms. Therefore, we compare the performance of our directive-based implementation against the MPI-based hand-parallelized implementation. It should be noticed that an MPI-based implementation differs from a directives-based shared-memory implementation of the same program in two important respects:

1. program runs under Single Program, Multiple Data (SPMD) paradigm and shares data with explicit message-passing among multiple processes; and
2. data is distributed such different processors "own" different elements of an array according to the type of distribution.

In contrast, shared-memory parallelized programs are executed under a fork-and-join paradigm with a global address space. Additionally, data distribution directives result in the ownership of different pages of data (arrays) by different processors, in contrast to the ownership of specific elements of an array.

Figure 7 presents the comparison between directives-based parallelized benchmarks and hand-parallelized, MPI-based versions of the same. In all of these cases, performance improves with the number of processors. For BT and FT, the MPI-based implementations perform slightly better than the shared-memory implementation due to data placement. Directives-based data distribution results in placing pages of arrays on multiple processors Coarse granularity of data distribution starts becoming a bottleneck for larger number of processors because all loop iterations that use a particular data element cannot be co-located at the same node. Therefore, as the number of processors increases, multiple processors have to access data from pages that they do not own locally, which adversely impact the overall execution time. In contrast, a message-passing program is designed in a way that the programmer controls locality of every data element. As the number of processors increases, the amount of data owned by a processor reduces proportionately. This is a particularly favorable situation for a cache-based DSM system because larger proportions of local data can reside in caches to enhance memory system performance. We tuned BT's data locality for almost all of the parallelized loops to ensure that each loop iteration is scheduled at a processor that owns elements of an array accessed during those iterations. Consequently, the performance of BT is comparable to its hand-parallelized implementation. Performance of two implementations of CG and MG is also comparable (see Figure 7 (c) and (d)). In case of CG and MG, data locality does not become a bottleneck due to comparatively smaller size of code with smaller number of memory accesses. Therefore, performance remains comparable with the hand-parallelized implementations of CG and MG.

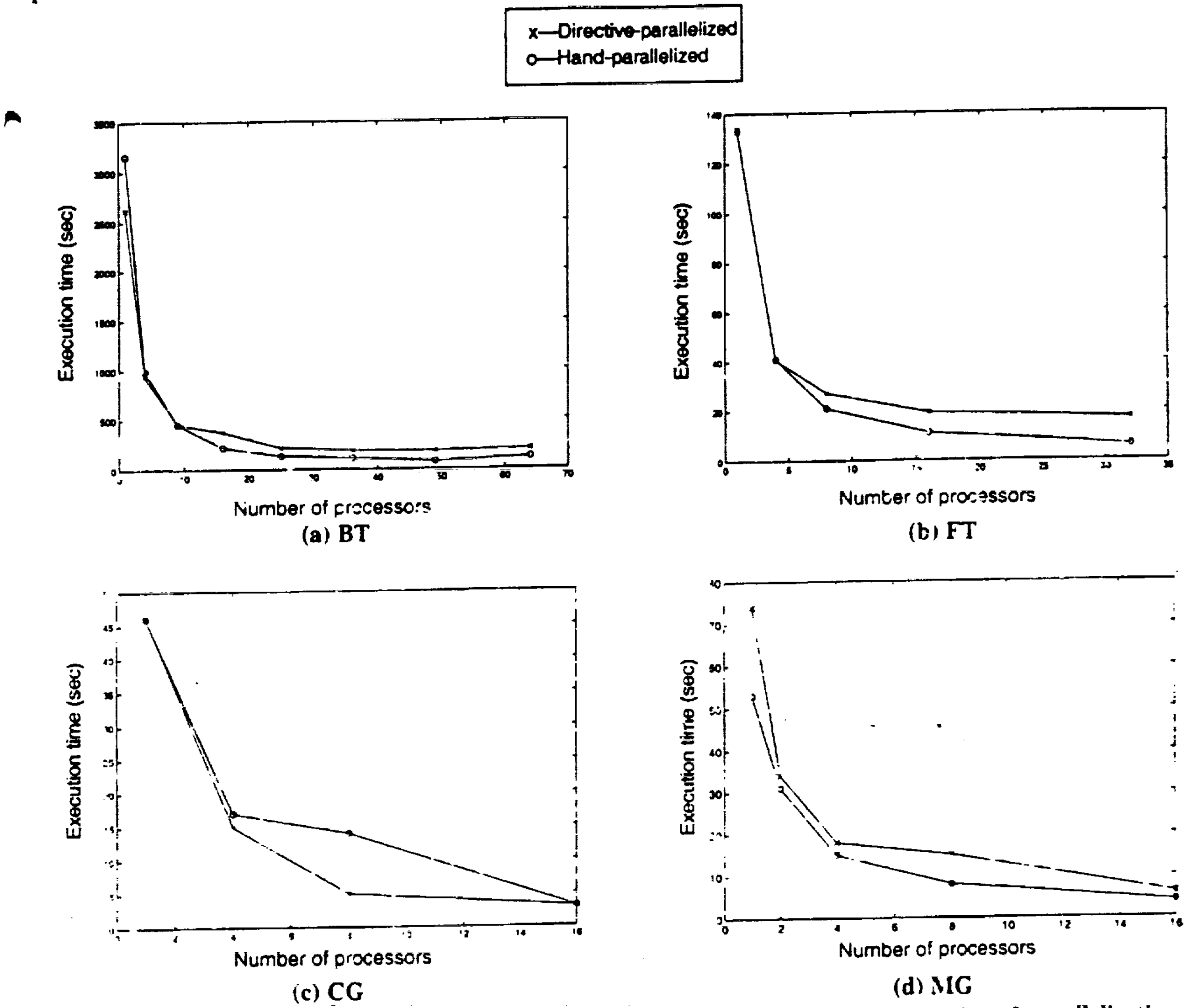


Figure 7. Performance comparison of shared-memory multiprocessing directives-based parallelization with MPI-based, hand-parallelized and -optimized versions of the same benchmarks.

4.5 Summary of Performance Evaluation

As a first step in evaluation process, the parallel coverage of each parallelized program was determined. Despite above 90% parallel coverage in all cases, programs cannot achieve close to ideal or theoretical speedup due to parallelization overhead. Our extensive experiments indicate that a useful quantitative measure of parallelization overhead is obtained by the performance model presented in this paper, which calculates aggregate overhead without trying to isolate different types of overhead. Based on our experience with performance tuning described here, we conclude that parallelization overhead can be significantly reduced by improving data locality. Superior speedup of message-passing implementation of same benchmarks due to improved data locality supports this conclusion.

5 Related Work

Recent performance evaluation studies have examined the effect of data locality on the performance of DSM systems. Anderson reports that overhead for programs that were parallelized with near 100% parallel coverage and executed on Stanford DASH (a ccNUMA DSM system) resulted in significantly inferior speedup characteristics [4]. Performance was improved by analyzing data distribution. In our case, we conclude that single processor cache performance is another key factor that can improve performance, in addition to appropriate data distribution. Hristea et al present the results of several experiments to evaluate the performance of memory subsystem for ccNUMA systems [8].

Several research efforts have focused on parallelizing sequential programs for shared-memory multiprocessors. These efforts are becoming increasingly important due to the revival of shared-memory multiprocessors with improved scalability via distributed memory and hardware cache-coherence. SUIF compiler system incorporates various modules that can be used to analyze the sequential program, parallelize the loops, distribute program arrays, and perform inter-procedural analysis [3,4]. Polaris is another parallelizing compiler that can generate parallelized code for SMPs [16,18]. CAPTools is a semi-automatic parallelization tool that transforms a sequential program to a message-passing program by user-directed distribution of arrays [9]. Fortran-D [1] and various implementations of High Performance Fortran (HPF [6]) are examples of parallelizing compilers that work for sequential programs that can benefit from data parallelism. KAP [10] and PFA [13] are examples of commercial parallelization tools for SMPs. We have experimented with most of these tools to parallelize sequential NAS benchmarks. Based on this experience and results reported in this paper, we consider that tools for SMPs are simple to learn and use and their performance is promising.

6 Discussion and Conclusions

Directives-based parallelism is essentially a fine-grained parallelism that works at the level of individual loop iterations. This is fundamentally different from conventional coarse-grained parallelism at the level of processes or threads. When it is implemented carefully, it can obtain much better load-balance compared to the conventional message-passing or data-parallel techniques. On the other hand, the user is required to spend additional time to ensure proper data locality to obtain performance comparable to hand-parallelized, message-passing based implementation.

We presented a performance model to characterize the performance of directives-based parallelized programs for an Origin2000 system. Using measurements, we quantitatively evaluated the fraction of code that was parallelized. Further evaluation indicated reasonable speedup as well as significant parallelization overhead. Based on extensive tuning of one parallelized program and some isolated experiments presented in this paper, we conclude that non-local data accesses are the main source of parallelization overhead. Performance can be optimized by keeping data at a level in memory hierarchy, which is closer to the processor. Based on these results, we continue to further tune parallelized NAS benchmarks.

Evaluation of parallelization overhead based on performance model presented in this paper emphasizes the need for appropriate instrumentation of multiprocessor memory subsystem. Such instrumentation is readily accessible to a user for measurements limited to a single node only. Without hardware or software based instrumentation of non-local memory accesses and cache-coherence traffic, direct measurement of data locality overhead is not possible. Some commercial tool developers realize this problem and are working on tools that furnish multiprocessor memory performance measurements.

References

[1] V. Adve, J-C. Wang, J. Mellor-Crummey, D. Reed, M. Anderson, and K. Kennedy, "An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs," *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[2] Saman P. Amarasinghe, "Parallelizing Compiler Techniques Based on Linear Inequalities," Ph.D. Dissertation, Dept. of Electrical Eng., Stanford University, Jan. 1997.

[3] S. P. Amarasinghe, J. M. Anderson, M. S. Lam and C. W. Tseng, "The SUIF Compiler for Scalable Parallel Machines," *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing*, July, 1995.

[4] Jennifer-Ann M. Anderson, "Automatic Computation and Data Decomposition for Multiprocessors," Technical Report CSL-TR-97-719, Computer Systems Laboratory, Dept. of Electrical Eng. and Computer Sc., Stanford University, 1997.

[5] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow, "The NAS Parallel Benchmark 2.0," Technical Report NAS-95-020, December 1995.

[6] High Performance Fortran Forum, High Performance Fortran Language Specification, Version 1.0, Scientific Programming, 2(1 & 2), 1993.

[7] Mark Horowitz, Margaret Martonosi, Todd. C. Mowry, and Michael D. Smith, "Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors," *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[8] Cristina Hristea, Daniel Lenoski, and John Deen, "Measuring Memory Hierarchy Performance of Cache-Coherent Multiprocessors Using Micro benchmarks." *Proceedings of SC '97*, San Jose, California, Nov. 1997.

[9] C. S. Ierotheou, S. P. Johnson, M. Cross, and P. F. Leggett "Computer aided parallelisation tools (CAPTools)—conceptual overview and performance on the parallelisation of structured mesh codes" *Parallel Computing*, Vol.22, 1996, pp.163-195.

[10] Kuck & Associates, Inc., "Experiences With Visual KAP and KAP/Pro Toolset Under Windows NT," Technical Report, Nov. 1997.

[11] James Laudon and Daniel Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server." *Proceedings of the 24th Annual International Symposium on Computer Architecture*, Denver, Colorado, June 2–4, 1997, pp. 241-251.

[12] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," May 5, 1994.

[13] *MIPSpro Fortran77 Programmer's Guide*, Silicon Graphics, Inc. Available on-line from: http://techpubs.sgi.com/library/dynaweb_bin/0640/bin/nph-dynaweb.cgi/dynaweb/SGI_Developer/MproF77_PG/@Generic__BookView.

[14] NAS Parallel Benchmarks. Available on-line from: http://science.nas.nasa.gov/Software/NPB.

[15] *OpenMP: A Proposed Standard API for Shared Memory Programming*. Oct. 1997. Available on-line from http://www.openmp.org.

[16] David A. Padua, Rudolf Eigenmann, Jay Hoeflinger, Paul Petersen, Peng Tu, Stephen Weatherford, and Keith Faigin, "Polaris: A New-Generation Parallelizing Compiler for MPPs," Technical Report CSRD # 1306, University of Illinois at Urbana-Champaign, June 15, 1993.

[17] Cherri M. Pancake, "The Emperor Has No Clothes: What HPC Users Need to Say and HPC Vendors Need to Hear.", *Supercomputing '95*, invited talk, San Diego, Dec. 3–8, 1995.

[18] Insung Park, Michael J. Voss, and Rudolf Eigenmann, "Compiling for the New Generation of High-Performance SMPs," Technical Report, Nov. 1996.

[19] Harvey J Wassermann, Olaf M. Lubeck, Yong Luo, and Federico Bassetti, "Performance Evaluation of the SGI Orign2000: A Memory-Centric Characterization of LANL ASCI Application." *Proceedings of SC '97*, San Jose, California, Nov. 1997.

[20] Marco Zagha, Brond Larson, Steve Turner, Marty Itzkowitz, "Performance Analysis Using the Mips R10000 Performance Counters," *Proceedings of Supercomputing '96*, Pittsburgh, Pennsylvania, Nov. 1996.

	NAS TECHNICAL REPORT
NAS	Title: **Performance Modeling and Measurement of Parallelized Code for Distributed Shared**
	Author(s): Abdul Waheed and Jerry Yan
	Reviewers: "I have carefully and thoroughly reviewed this technical report. I have worked with the author(s) to ensure clarity of presentation and technical accuracy. I take personal responsibility for the quality of this document."
Two reviewers must sign.	Signed: __________ Name: _H. Jin__________ Signed: __________ Name: _M. Hribar__________
After approval, assign NAS Report number.	Branch Chief: Approved: __________
Date:	NAS ReportNumber: NAS-98-012