RC 20045 (04/27/95)
Computer Science/Mathematics

IBM Research Report

The Performance Impact of Granularity Control and Functional Parallelism

José E. Moreira

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

Dale A. Schouten and Constantine D. Polychronopoulos

Center for Supercomputing Research and Development
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main St. Urbana, IL 61801-2307

Research Division
Almaden · T.J. Watson · Tokyo · Zurich

The Performance Impact of Granularity Control and Functional Parallelism*

José Moreira† Dale Schouten‡ Constantine Polychronopoulos‡

moreira@watson.ibm.com {schouten,cdp}@csrd.uiuc.edu

† IBM T J Watson Research Center
Yorktown Heights, NY 10598-0218

‡ Center for Supercomputing Research and Development,
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main St. Urbana, IL 61801-2307

Abstract

Task granularity and functional parallelism are fundamental issues in the optimization of parallel programs. Appropriate granularity for exploitation of parallelism is affected by characteristics of both the program and the execution environment. In this paper we demonstrate the efficacy of dynamic granularity control. The scheme we propose uses dynamic runtime information to select the task size of exploited parallelism at various stages of the execution of a program. We also demonstrate that functional parallelism can be an important factor in improving the performance of parallel programs, both in the presence and absence of loop-level parallelism. Functional parallelism can increase the amount of large-grain parallelism as well as provide finer-grain parallelism that leads to better load balance. Analytical models and benchmark results quantify the impact of granularity control and functional parallelism. The underlying implementation for this research is a low-overhead threads model based on user-level scheduling.

Keywords: dynamic scheduling, functional parallelism, task granularity, parallel processing, threads.

*This work was supported by the Office of Naval Research under grant N00014-94-1-0234. Computational facilities were provided by the National Center for Supercomputing Applications. José Moreira was at the University of Illinois during the development of this research.

1 Introduction

Several studies have documented the adverse effect of runtime overhead on the performance of parallel programs. Runtime overhead arises from different sources, including bookkeeping operations, scheduling overhead, memory and communication/synchronization latency, and architectural and operating system idiosyncrasies. In this paper we study the effect of task granularity on overall performance. We also consider the benefit of exploiting nested functional (nonloop) parallelism and contrast it to data (loop) parallelism.

In [2], [4], and [11] it was shown that, given a fixed number of processors, exploiting excessive parallelism increases parallel execution time. Although it is intuitively clear that only a limited amount of parallelism can take advantage of a fixed number of resources, the magnitude to which runtime overhead affects performance was demonstrated in the above studies. In order to alleviate this problem [11] and other subsequent studies provided an environment that allows the user to control the number of parallel tasks a given parallel application generates. Given a fixed number of resources, a user or compiler can restrict the maximum number of parallel tasks of a parallel application to less than or equal to a predetermined amount.

However, in multiprogramming environments where many parallel processes may be competing for the same resources, the effective number of processors allocated to a process may vary from 1 to P, where P is the total number of physical processors. Alternatively, a machine with P processors may be statically partitioned into P/k clusters of k processors each; this, however, could unnecessarily penalize large parallel processes while some clusters may be underutilized.

This paper reports on an implementation which employs the notion of dynamic granularity control. At any given time, the number of parallel activities a process generates is proportional to the number of physical resources allocated to that process. This allows the operating system to dynamically allocate a varying number of processors to different processes. In fact, the number of processors allocated to a particular process may vary over its lifetime.

The immediate impact of granularity control is the elimination of unnecessary overhead due to frequent context switching, creation and scheduling of tasks, additional interprocessor communication, and increased memory latency. Our method relies on a program representation which encapsulates the hierarchy of computations inherent in a parallel application. This allows for parallelism to be exploited first at the highest level of this hierarchy which corresponds to the outermost loops and the first-level function calls. Subject to resource availability, inner levels of parallelism are exploited by decomposing nested parallelism. A related focus of this work is the performance implications of the exploitation of functional (nonloop) parallelism. Our experiments indicate that functional parallelism can improve performance by a significant margin, even in situations where data (loop) parallelism is in abundance. This paper is organized as follows: Section 2 describes the programming model and target machine architecture. Section 3 describes an autoscheduling threads model, *nano*Threads. Queue management and granularity control issues are addressed in Sections 4 and 5 respectively. The environment used for our measurements is described in Section 6. An analytical model showing the benefits of the exploitation of functional parallelism, together with some experimental results from synthetic benchmarks, are presented in Section 7. The set of benchmarks used for more general measurements is listed in Section 8, and the results from these measurements are shown in Section 9. Finally, related work is discussed in Section 10, and concluding remarks are given in Section 11.

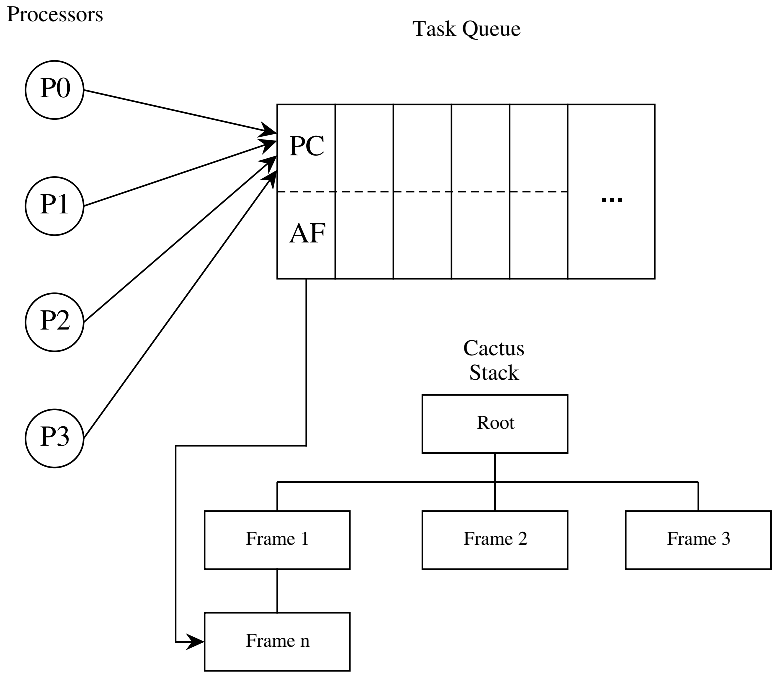
2 Machine and Programming Model

The target machine model is a shared address space multiprocessor with a multiprogramming environment. Therefore only a subset of the machine's processors will be allocated to a particular program. We call this subset of processors a *partition* and let this partition be *time-variant*, meaning that processors may be added or removed by the operating system during the execution of the job.

The program model is the *hierarchical task graph* [9], or HTG, combined with *autoscheduling* [20]. The HTG is an intermediate program representation that encapsulates data and control dependences at various levels of task granularity. This structure is used to generate autoscheduling code, which includes the scheduling operations directly within the program code. The HTG represents a program in a hierarchical structure, thus facilitating task-granularity control. Information on control and data dependences allows the exploitation of functional (task-level) parallelism, in addition to data (loop-level) parallelism. A brief summary of the properties of the HTG is given here and details can be found in [4, 8, 16, 9, 20].

The hierarchical task graph is a directed acyclic graph $HTG = (HV, HE)$ with unique nodes START and STOP $\in HV$, the set of vertices. Its edges, HE, are a union of control (HC) and data dependence (HD) arcs: $HE = HC \cup HD$. The nodes represent program tasks and can be of three types: *simple*, *compound*, and *loop*. A simple node represents the smallest schedulable unit of computation. A compound node is an acyclic task graph (ATG) comprised of smaller nodes, each recursively defined as an HTG. A loop node represents a task that is either a serial loop (in which case all iterations must be executed in order) or a parallel loop (in which case the iterations may be executed simultaneously in any order). The body of a loop can be an HTG. An HTG may have local variables that can be accessed by any task in the HTG. In the general model, each task may have internal task-local variables.

Each HTG is associated with a set of boolean flags (local variables) that mark the execution of nodes and arcs in the HTG. As a task executes, its autoscheduling *drive code* at the exit block of the task updates the values of these boolean flags. Each node is associated with an *execution tag*: an expression on the boolean flags that is derived from the data and control dependences and represents the execution condition for that node. Let $\varepsilon(x)$ denote the execution tag of node x. Whenever the values of the boolean flags cause $\varepsilon(x)$ to evaluate to `TRUE`, node x will be *enabled* and ready to execute. This evaluation of $\varepsilon(x)$ is also performed by the drive code for all the successors of a task and the enabled tasks are placed in a task queue. Autoscheduling and the HTG effectively implement a *macro dataflow* model of execution in a conventional multiprocessor.



Scheduling loop
repeat
    fetch task T
    execute T
until DONE

Figure 1: Task queue, cactus stack, and main scheduling loop for a *nano*Threaded program.

3 *Nano***Threads**

*Nano*Threads [19] is a threads architecture that combines low-overhead threads and autoscheduling. Each *nano*Thread corresponds to an HTG task. Since the tasks effectively schedule themselves, via the exit blocks of the HTG, the user-level scheduler is a simple loop. Given a pointer to a task queue, it retrieves tasks from the queue and executes them directly (via a goto or function call) rather than performing a context switch. Each processor in the partition allocated to a job runs such a scheduling loop. Only one system-level thread or shared memory process per processor is needed, and it only has to be created at the beginning of the job execution. As soon as the system-level thread starts running, it enters the scheduling loop and then all the user-level scheduling is done by the drive code of autoscheduling. We call these system-level threads *virtual processors* since they are doing the actual work. Every virtual processor attached to a particular job accesses the same task queue.

Each task in the queue contains two pieces of information: a code pointer (program counter) and a pointer to an activation frame (AF) containing the data local to that task. The activation frames are organized in a tree-structure called the *cactus-stack*. This is illustrated in Figure 1, together with the main scheduling loop. An entire parallel loop is represented by a single task descriptor which, in addition to the above information, contains a loop iteration counter. This allows the processors to perform dynamic loop scheduling.

The overhead associated with task dispatch is small, consisting of loading a register with the activation frame pointer and jumping to the beginning of the code for the task. Other overhead of enqueueing tasks and allocating activation frames is incurred in the program code, specifically in the entry and exit blocks of the nodes of the HTG.

4 Queue Management

Processors dispatch ready tasks and enqueue new tasks from and onto a task queue respectively. A centralized queue provides good load balancing, but it can also be a source of contention. In addition, a centralized queue does not facilitate preferential assignment of tasks to processors.

Simple queueing models [17] show that a single queue may work well for medium-size systems with 8 to 16 processors, but becomes a performance bottleneck in larger systems with 64 or more processors. The ability to bind tasks to specific processors is useful in exploiting locality of accesses, especially if this can be done without adversely affecting load balancing. A distributed task queue is used to address both the contention and the locality issues. Each processor has its own local queue, and the enqueueing and dequeueing policies work as follows:

- *Enqueueing:* When a processor determines that a new task has become ready for execution, the corresponding task descriptor is placed in the processor's local queue.
- *Dequeueing:* A processor first tries to fetch a task from its own local queue. If its queue is empty, it then searches the local queues of other processors for work.

On a heavily loaded system, local tasks tend to cluster on the same processor, enhancing locality of access. Consider the loop nest:

doall $I = 1, N$
    doall $J = 1, M$
        Body
    end
end

Assume the descriptor for the **doall** I is in the local queue of processor P_0. Other idle processors will fetch from P_0's queue and participate in the execution of **doall** I. The execution of one iteration of **doall** I by processor P_i will create the corresponding descriptor for one instance of **doall** J in the local queue of P_i. While there are enough iterations of **doall** I to keep all processors busy, each processor will be fetching from its own local queue. Thus, contention is kept to a minimum and locality inside each iteration of **doall** I can be preserved. When the iterations of **doall** I are exhausted (i.e. they have all been *issued*) idle processors will start participating in the execution of remote instances of **doall** J, thus trading locality for load balance.

5 Granularity Control

Determining the best task granularity is one of the fundamental optimization problems in parallel processing. The granularity or *grain size* of a task is informally used to indicate the size of the task, typically measured in number of instructions. Typically the overhead of creating and scheduling a task is approximately the same regardless of its execution time. A large grain task spends a smaller percentage of time performing system functions such as scheduling and allocating its activation frame. The efficiency may be higher with larger tasks, thus reducing the relative overhead of parallelism. On the other hand, maximum exploitation of parallelism naturally leads to fine granularity tasks which can facilitate load balancing and increased utilization at the expense of efficiency. The smaller the overhead, the finer the granularity that can be exploited effectively. However, even in systems that support fine granularity, such as *nano*Threads, adjusting task granularity dynamically may be beneficial, and in fact necessary, under certain conditions, as supported by our experimental results.

Granularity control in *nano*Threads is based on hybrid scheme, part static and part dynamic, as explained below. The main goal of the static part is to guarantee a minimum task size at all times, while the dynamic part uses runtime information to select the appropriate level of granularity to exploit (i.e. above the minimum set by the compiler, and up to the maximum present in the application).

5.1 Static Granularity Control

Compile-time analysis can be used to guarantee that no task in the HTG is smaller than a certain size. The minimum size depends on the per task overhead of the system. The details of selecting an appropriate minimum size are beyond the scope of this paper. Static minimum granularity control can be implemented through *task merging*, a process described in [4].

Another aspect of static granularity control is to help prevent unnecessarily conservative dynamic granularity control decisions. When the overhead for task scheduling is negligible, as compared to the task size, there is little advantage in serializing the task. On the other hand, a dynamic granularity control decision that inhibits the parallelism of such a large task can only have a negative effect. Using static granularity control to prevent serialization can enhance performance. Our current implementation does not include automatic static granularity control, but does include facilities for specifying it manually.

5.2 Dynamic Granularity Control

Compile-time preconditioning of a program enforces a minimum granularity. In our scheme, however, the exact task size is determined at runtime and depends on program properties as well as runtime conditions. The latter depends largely on workload characteristics. Modern processor scheduling systems, such as Process Control [11] and Scheduler Activations [2], space-share the processors in a multiprocessor to improve utilization and locality of reference. In this case, the number of processors allocated to a particular job depends on other jobs running at the same time.

Dynamic granularity control works by exploiting parallelism beginning with very coarse tasks (outermost loops or first-level calls) and progressively moves to the inner subtasks. The granularity control decision is made when a new level of hierarchy in the HTG is about to be exploited, that is, at the entry block of a compound task (ATG). The decision will result in setting the execution mode for a particular ATG: serial or parallel. If the serial mode is chosen, then the tasks comprising this compound task will be executed sequentially by the processor assigned to the ATG. If the parallel mode is chosen, then the subtasks will be queued separately, and thus made available for parallel execution.

The decision is local to each ATG and does not restrict the decisions made at a lower level of the hierarchy.

An ideal granularity control scheme decomposes compound tasks into subtasks only to the extent that all processors are utilized. Generating more tasks creates unnecessary overhead. Generating fewer task results in underutilization. The heuristics for granularity control presented and evaluated in this paper attempt to approach this ideal behavior with very simple operations so that granularity control does not become a significant overhead by itself. In our current implementation, when execution enters a compound node (ATG), its execution mode is set by the following:

$$mode = \begin{cases} \textbf{parallel} & \text{if } Q_n/P \leq \alpha \\ \textbf{serial} & \text{otherwise} \end{cases} \tag{1}$$

where Q_n is the number of tasks in the queue, P is the number of processors currently assigned to the process, and α is a given threshold. This mechanism responds to the runtime change in the number of processors appropriately. As the number of processors P increases, the ratio $\frac{Q_n}{P}$ decreases and more tasks are created, making more work available for the processors. Conversely, as the number of processors decreases, fewer tasks are made available. In general, the best value for α is machine and application dependent, but experience indicates that small values like $\alpha = 1$ or $\alpha = 2$ work well for a variety of cases.

5.3 Distributed Task Queues

If the task queue is physically distributed, the total number of tasks on all queues is a valid measure of the system load. However, maintenance of this information requires updates of a centralized counter with each task enqueueing or dequeueing operation. This creates the same type of contention that the distributed queue seeks to avoid.

A fully distributed scheme involves each processor checking only its local queue. The same criteria as in the centralized queue is used (Expression 1), but Q_n now represents the size of a processor's local queue and $P = 1$. Since idle processors fetch tasks from the queue of other processors, the local queue of a processor contains some information about the state of the entire system. However, information is less accurate than in the case of a centralized queue. A fully distributed queue was employed in the *nano*Threads library.

6 Measurements Environment

*Nano*Threads was implemented on an SGI Challenge shared-memory multiprocessor [7]. The machine on which measurements were obtained was configured with 32 MIPS R4400 processors and 2 Gigabytes of main memory. Each processor had a local cache with hardware support for cache consistency.

Two different versions of *nano*Threads were implemented on the SGI Challenge. One is a C++ *nano*Threads library that allows the user to describe the HTG structure of a program as a collection of C++ classes. The actual body of the tasks can be coded either in C++ or in Fortran.

The other implementation is a *nano*Threads compiler based on the Parafrase-2 compiler [18]. This compiler uses the HTG representation of a Fortran program to automatically generate C++ code with an embedded scheduler. In both cases, system-level shared-address processes are created at the beginning of the program execution to implement the virtual processors, while all the *nano*Threads operations are performed by user-level code.

The benchmark suite used to evaluate the performance impact of granularity control and functional parallelism consists of 1) synthetic benchmarks designed to produce specific behaviors of parallelism, and 2) application kernels that are representative of the behavior of actual programs. The SGI Challenge timer facilities were used to measure wall-clock execution times. The measurements were made in single user time whenever possible, or when the machine was lightly loaded.

7 Benefits of Functional Parallelism

The most obvious benefit of exploiting functional parallelism is to supply additional parallelism when loop parallelism alone does not saturate the system. Many non-numeric applications exhibit functional parallelism but no data parallelism. A typical example of such a case is Quicksort, which is discussed in Section 9. However, functional parallelism can increase processor utilization and load balance even in the presence of abundant data or loop-level parallelism. In this section we present performance models that quantify these improvements.

Consider the task graph shown in Figure 2. All n tasks can execute concurrently. Let each task be composed of a sequential part involving q computations and a parallel part involving p computations. Assume that the overhead for parallel processing is negligible. Consider the case in which all tasks are identical. The serial execution time of a task T_i is given by:

$$t_{T_i} = q + p. \tag{2}$$

The serial execution time of the entire task graph is:

$$t = n(q + p). \tag{3}$$

Loop Parallelism: We first consider the case in which only loop parallelism is exploited. The tasks are executed in sequence, each task running on all P processors. The total execution time is given by:

$$t_{P(\text{loop})} = n(q + \frac{p}{P}) = qn + \frac{pn}{P}. \tag{4}$$

Loop and Functional Parallelism: Alternatively, the set of P processors could be distributed uniformly among the n tasks so that all tasks execute concurrently, thus exploiting functional parallelism. For simplicity assume n divides P. In this case the execution time for the graph on P processors is the same as the execution time of a task on P/n processors:

$$t_{P(\text{fp})} = q + \frac{p}{P/n} = q + \frac{pn}{P}. \tag{5}$$

We define the *functional speedup*, $S_{P(\text{fp})}$, as the ratio of the time to execute the task graph with only loop parallelism to the time to execute the same task graph with loop *and* functional parallelism. By normalizing $q + p = 1$, we can

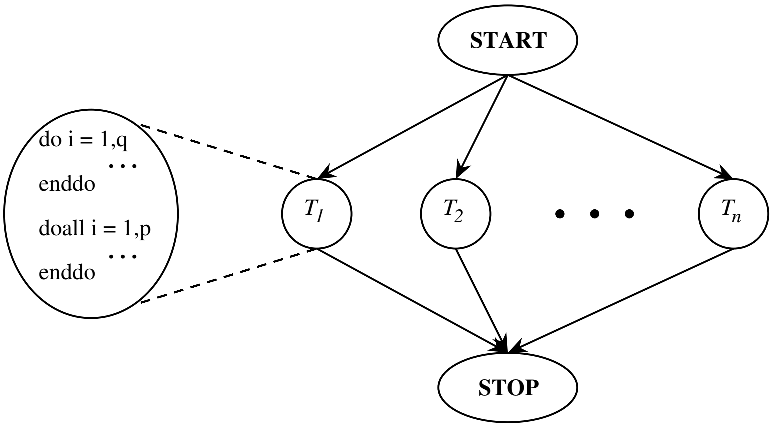


Figure 2: Task graph used in the performance models. Each task has a serial part and a parallel part and the tasks are independent.

compute $S_{P(\text{fp})}$ as a function of n and p:

$$S_{P(\text{fp})}(n,p) = \frac{t_{P(\text{loop})}}{t_{P(\text{fp})}} = \frac{(1-p)n + \frac{pn}{P}}{(1-p) + \frac{pn}{P}} = \frac{Pn(1-p) + pn}{P(1-p) + pn}. \tag{6}$$

Figure 4(a) is a plot of functional speedup for varying numbers of processors when $n = 4$ and $p = (0.5, 0.8, 0.9)$. Experimental points from a synthetic benchmark that implements the task graph of Figure 2 are also shown. Large granularity tasks, with $p + q = 256$ million floating point multiplies, were used to minimize the impact of overhead, thus matching the assumptions of the analytical model. The functional speedup is expected to improve with smaller task granularities as the relative overhead, and therefore q, increases.

Let us consider the case in which the parallel section in the tasks varies. Using the same task graph of Figure 2, let the parallel part of task i be of size ip, thus the serial execution time for task T_i is:

$$t_{T_i} = q + ip. \tag{7}$$

and the serial execution time of the entire task graph is:

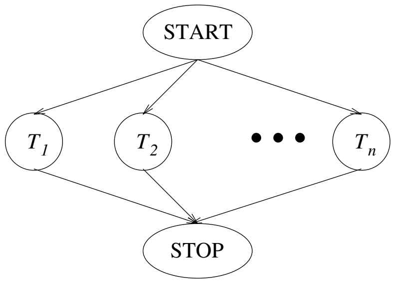
$$t_1 = \sum_{i=1}^{n}(q + ip) = nq + p\frac{n(n+1)}{2}. \tag{8}$$

By exploiting loop parallelism on P processor, the execution time is:

$$t_{P(\text{loop})} = \sum_{i=1}^{n}(q + \frac{ip}{P}) = nq + \frac{p}{P}\frac{n(n+1)}{2}. \tag{9}$$

Static scheduling for functional parallelism can be done by assigning a fixed partition of P/n processors to each task. This results in a total execution time for the task graph equal to the execution time of the longest task (T_n) on

doall $i = 1, P$
    do $j = 1, i$



    enddo
enddoall

Figure 3: Parallel loop with internal functional parallelism.

P/n processors:

$$t_{P(\text{fps})} = q + \frac{pn}{P/n} = q + \frac{p}{P}n^2. \tag{10}$$

From Equations (9) and (10) it can be seen that $t_{P(\text{fps})}$ may actually be *larger* than $t_{P(\text{loop})}$, in which case the exploitation of functional parallelism is detrimental to performance[1]. Figure 4(b) is a plot of the resulting functional speedup:

$$S_{P(\text{fps})}(n,p) = \frac{t_{P(\text{loop})}}{t_{P(\text{fps})}} = \frac{n(1-p) + \frac{p}{P}\frac{n(n+1)}{2}}{(1-p) + \frac{p}{P}n^2} = \frac{2Pn(1-p) + pn(n+1)}{2(P(1-p) + pn^2)}. \tag{11}$$

Further performance improvements can be obtained by allowing dynamic assignment of processors to task. As the smaller tasks finish, their processors are reassigned to other tasks. The execution time of the task graph is still determined by the longest task, T_n, but the number of processors assigned to T_n varies with time. The execution of the task graph can be divided into $n+1$ stages: in stage 0 the sequential part of all tasks is computed, taking time q. In stage 1, each task gets P/n processors that operate on a section p of the parallel part. This stage takes time $\frac{p}{P/n}$ completing task T_1. For the next stage, each of tasks $T_2, \ldots, T_n$ gets $P/(n-1)$ processors (ideally) to compute the next section p. The execution time for the whole graph is given by:

$$t_{P(\text{fpd})} = q + \sum_{i=0}^{n-1} \frac{p}{P/(n-i)} = q + \frac{p}{P}\sum_{i=0}^{n-1}(n-i) = q + \frac{p}{P}\sum_{i=1}^{n} i = q + \frac{p}{P}\frac{n(n+1)}{2} \tag{12}$$

which corresponds to uniformly distributing the total work among all P processors. The functional speedup is given by:

$$S_{P(\text{fpd})}(n,p) = \frac{t_{P(\text{loop})}}{t_{P(\text{fpd})}} = \frac{n(1-p) + \frac{p}{P}\frac{n(n+1)}{2}}{(1-p) + \frac{p}{P}\frac{n(n+1)}{2}} = \frac{2Pn(1-p) + pn(n+1)}{2P(1-p) + pn(n+1)} \tag{13}$$

which is always greater than 1, as can be seen in Figure 4(c). Again, experimental results from a synthetic benchmark support the analytical model.

[1] This may occur for large values of p which satisfy $q < \frac{pn}{2P}$

In the previous case, functional parallelism was external to the parallel loops and additional speedup was obtained by executing serial sections of independent tasks concurrently. Benefits from functional parallelism can also be observed when the parallel task graph is inside the body of a parallel loop. Decomposing the parallel loop iterations into independent subtasks can provide potentially better load balance.

Consider the parallel loop in Figure 3. Let each task T_i be strictly serial with execution time t. The serial execution time of the loop nest is given by:

$$t_1 = \frac{P(P+1)}{2} nt. \tag{14}$$

Loop Parallelism: Exploiting only loop parallelism on P processors, the execution time of the entire loop will be the execution time of iteration $i = P$, the longest loop iteration:

$$t_{P(\text{loop})} = Pnt. \tag{15}$$

In this case, the maximum load difference between two processors is the difference between iterations $i = 1$ and $i = P$, or $(P-1)nt$.

Loop and Functional Parallelism: When the functional parallelism inside each loop iteration is exploited, the maximum load imbalance is reduced to the size of the largest subtask (t). Processors that finish their iterations early become available to execute subtasks of iterations from other processors. There are a total of $\frac{P(P+1)}{2}n$ tasks to be executed. Ideally, the tasks are uniformly distributed among the processors (if P divides n), and in this case the execution time is:

$$t_{P(\text{fp})} = \frac{P(P+1)}{2} \frac{n}{P} t. \tag{16}$$

The functional speedup obtained by using functional parallelism is:

$$S_{P(\text{fp})} = \frac{t_{P(\text{loop})}}{t_{P(\text{fp})}} = \frac{Pnt}{\frac{(P+1)n}{2}t} = \frac{2P}{P+1}. \tag{17}$$

A plot of this speedup as a function of the number of processors, together with some experimental points, is shown in Figure 4(d). The experimental points were obtained from a synthetic benchmark corresponding to the task graph of Figure 3, with $n = 16$ and each task equal to 5 million floating point multiplies.

8 Description of Benchmarks

In order to demonstrate the efficiency of *nano*Threads in real programs, several benchmarks have been tested. Unless otherwise specified, the time measured is the time actually required to perform the requisite calculations, excluding the time required to initialize the system or read in data. We give here a quick description of each benchmark.

Matrix Multiply (MM): The matrix multiply application is written in C++ using the *nano*Threads library. The code simply performs a straightforward implementation of the definition of matrix multiply. The two outermost loops are run in parallel as doall loops.

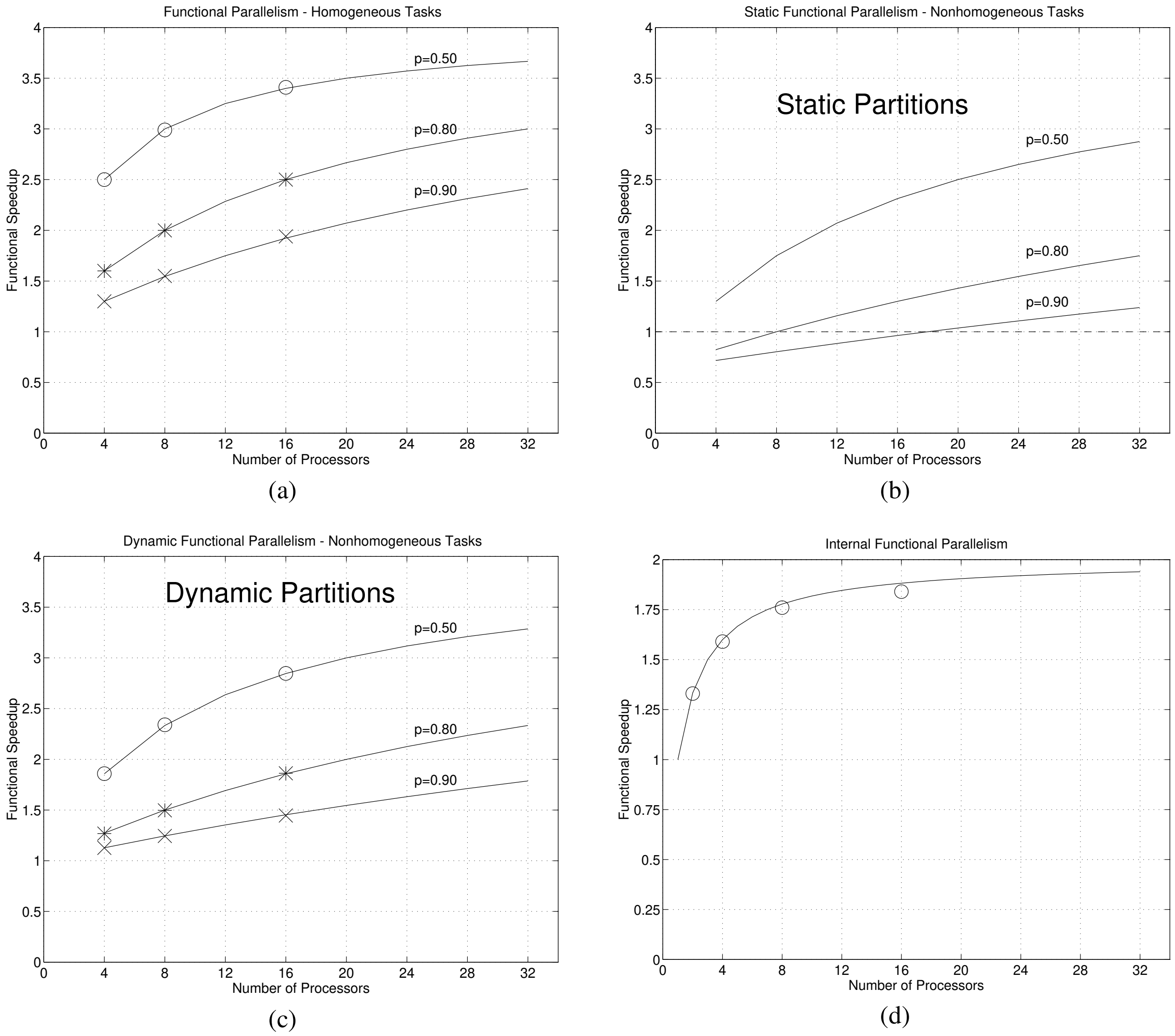


Figure 4: Functional speedup as a function of the number of processors and fraction of parallelism p. In the above plots the lines represent results from the analytical models and the points are actual measurements of synthetic benchmarks. In plots (a), (b), and (c) functional parallelism is external to loop parallelism. In plot (d) functional parallelism is internal to loop parallelism. In (a) all the tasks are of same size (homogeneous). In (b) the tasks are of different sizes (nonhomogeneous) and fixed-size processor partitions are statically assigned to tasks. In (c) the tasks again are of different sizes but processors are dynamically allocated to the tasks.

Strassen's Matrix Multiply (SMM): An implementation of Strassen's matrix multiply algorithm [10] was also written in C++ using the *nano*Threads library. Strassen's algorithm is a recursive algorithm that breaks down a matrix into four quadrants, recursively performs multiplies on the quadrants and performs a combination of matrix adds and subtracts. The corresponding task graph is shown in Figure 5. Where a traditional matrix multiply of two $n \times n$ matrices is O(n^3), Strassen's algorithm is O($n^{2.807}$). Because the overhead is high, our implementation uses Strassen's algorithm only for large ($>= 64 \times 64$) matrices. When the quadrant size drops below 64, then the traditional parallel matrix multiply is used.

Complex Matrix Multiply (CMM): A C++ implementation using the *nano*Threads library. A complex multiply is equivalent to four real multiplies and two real adds. As shown in the corresponding task graph of Figure 5, these matrix operations represent functional parallelism. Each matrix A is represented by two matrices, one real and one imaginary, $A = A_r + jA_i$. The matrix $C = A \times B$ is calculated according to the equation:

$$C = (A_r B_r - A_i B_i) + j(A_r B_i + A_i B_r). \tag{18}$$

All of the measurements with the matrix multiplies in this paper are taken with matrices of size 256×256.

Two Dimensional Fast Fourier Transform (FFT2): Fortran code that performs a two dimensional fast Fourier transform was obtained from CMU's Task Parallel Suite [3] and hand-parallelized with the C++ *nano*Threads library. There are two nested parallel loops that are exploited. There is also one routine performing matrix transpose that exploits functional parallelism.

Perfect Benchmark TRFD (TRFD): A C++ version of the Perfect Club benchmark TRFD was implemented with the *nano*Threads library. This code was written from a high-level description of the functionality of TRFD. The code operates on a four-dimensional triangular matrix, performing several multiplies and rearrangements of the matrix. Only three loops were parallelized. Two of the parallel loops are perfectly nested.

Quicksort (QUICK): Quicksort[13] is a recursive "divide and conquer" algorithm that sorts a vector of elements. It does so by partitioning the vector into two parts, based on the value of the elements, and recursively applying the same technique to both parts. In the average case, Quicksort is O($n \log n$). Because of high fixed overhead, vectors below a certain fixed size (256 elements) are sorted using a conventional nonrecursive sorting technique. The two recursive calls are independent and parallelized using functional parallelism, as is the partition phase in each call. This benchmark was coded in Cedar Fortran, and the autoscheduling compiler automatically generated C++ autoscheduling code. The measurements were taken with a vector of size 1048576 elements.

Computational Fluid Dynamics (CFD): This is the kernel of a Fourier-Chebyshev spectral computational fluid dynamics code [14]. The task graph representing this code is shown in Figure 5. It consists of 4 stages that operate on matrices of size 128×128. The first stage involves six 2-dimensional FFTs, the second stage six element by element matrix multiplies, the third stage three matrix subtracts, and the fourth and final stage three 2-dimensional FFTs. The parallelism at the task graph level is exploited, as is the loop parallelism inside each matrix operation $(\times, -, \mathrm{FFT})$. This benchmark was coded in Cedar Fortran, and translated by the autoscheduling compiler.

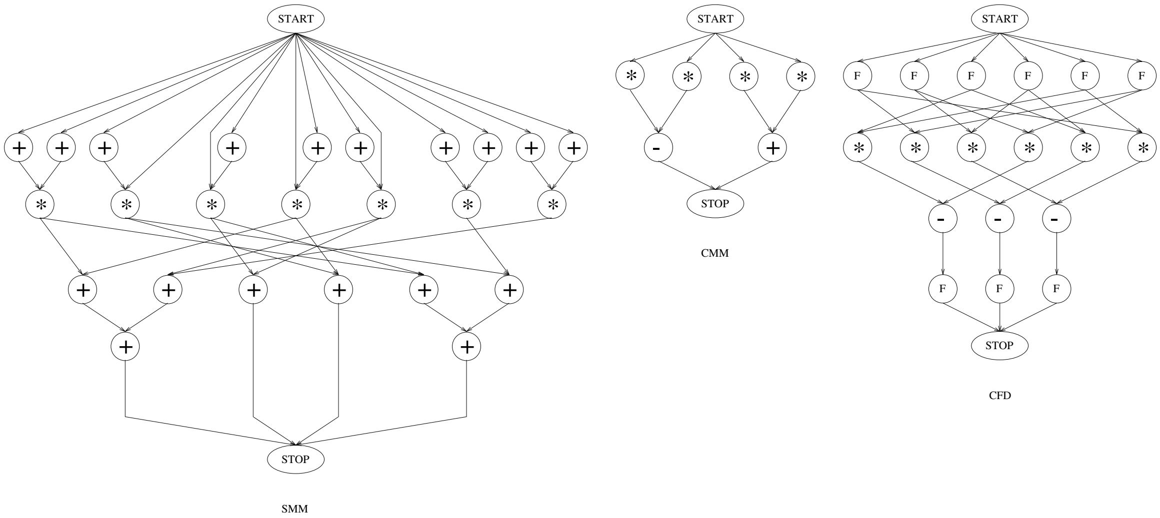


Figure 5: Task graphs for SMM, CMM, and CFD. F, *, +, and $-$ stand for 2D FFT, matrix multiply (element-by-element multiply in the case of CFD), matrix add, and matrix subtract, respectively.

9 Results

The results presented in this section are compiled from a set of execution times for each benchmark. The experimental points were obtained by running each program three times on a given number of processors and selecting the minimum execution time observed. For the *nano*Threads library benchmarks, a distributed queue was used. For the autoscheduling compiler benchmarks, a centralized queue was used. In both cases, the granularity control parameter α was set to 1 (see Section 5).

9.1 Measurements of Speedup

Figures 6 through 9 are speedup plots for the seven benchmarks. In each case the speedup S_P on P processors is computed as the ratio of serial execution time (for a serial version of the benchmark) to the parallel execution time on P processors. Different versions of parallel code are tested for each benchmark.

Figure 6 plots the speedup for the real and complex matrix multiply, MM and CMM. Speedups are shown for the cases with (GC) and without (No GC) granularity control. Both benchmarks contain two nested parallel loops, thus each contains enough loop parallelism to occupy all available processors. In addition, CMM exploits an outer level of functional parallelism. The effect of granularity control is noticeable in reducing the overhead of exploiting too many loops in parallel. This benefit increases with the number of processors, since the overhead for fetching iterations from a parallel loop is proportional to the number of processors.

Figure 7 shows the speedup for Strassen's algorithm (SMM) and the two-dimensional FFT (FFT2). Four versions are compared: with functional parallelism and granularity control (FP and GC), with functional parallelism but without granularity control (FP), with granularity control but without functional parallelism (GC), no granularity control and no functional parallelism (No FP or GC). The exploitation of functional parallelism in Strassen's algorithm is particularly

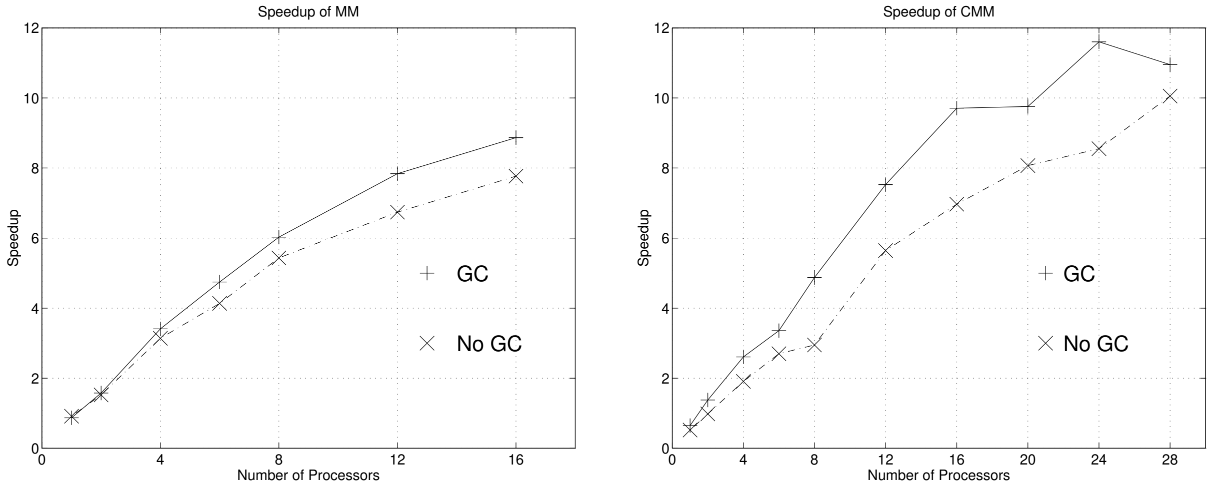


Figure 6: Speedups of MM and CMM, illustrating the benefits of granularity control.

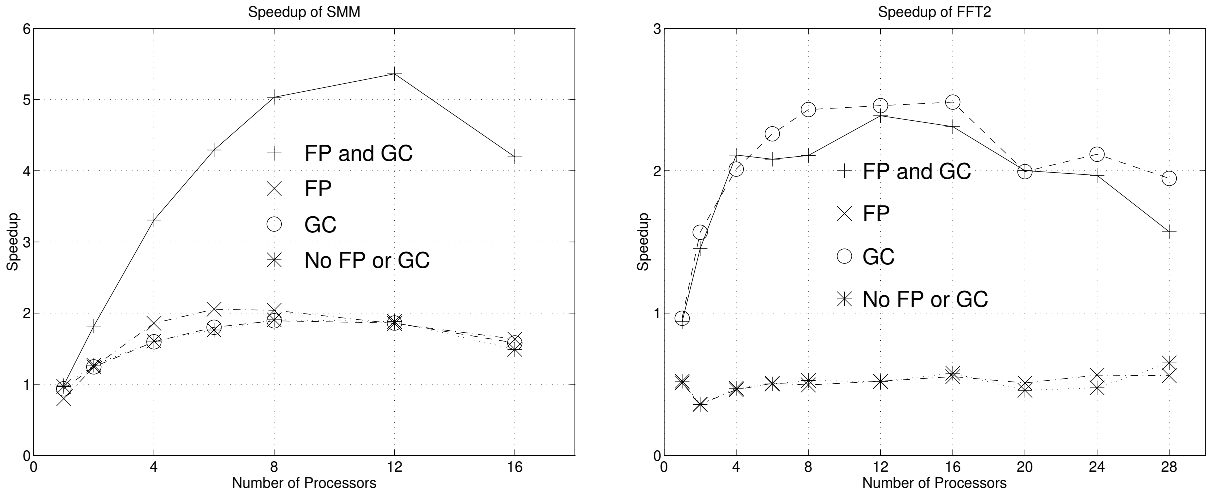


Figure 7: Speedups of SMM and FFT2 show the importance of both functional parallelism and granularity control.

important because it is a recursive algorithm and the loop parallelism occurs only at a smaller granularity. Because the recursive calls result in an exponential number of successively smaller tasks, granularity control has a significant effect as well. Only the version with functional parallelism and granularity control achieved reasonable speedup.

The body of the innermost parallel loop of benchmark FFT2 is very small. As a result, granularity control has a significant effect on its performance since it avoids unnecessary small grain parallelization. With granularity control, the version without functional parallelism generally achieves better performance than the version that exploits functional parallelism. The only functional parallelism in FFT2 that was exploited is in a matrix transpose function which causes bad cache performance. By using a profiler, pixie, we found that the number of instructions performed by the FP version is less than the number of instructions performed by the No FP version, since less time is spent by the schedulers spin waiting while there is no work to do. This indicates that the performance in a perfect memory system would be better for the version with functional parallelism.

Figure 8 shows the speedup of TRFD for the cases of *nano*Threads with granularity control (GC), *nano*Threads

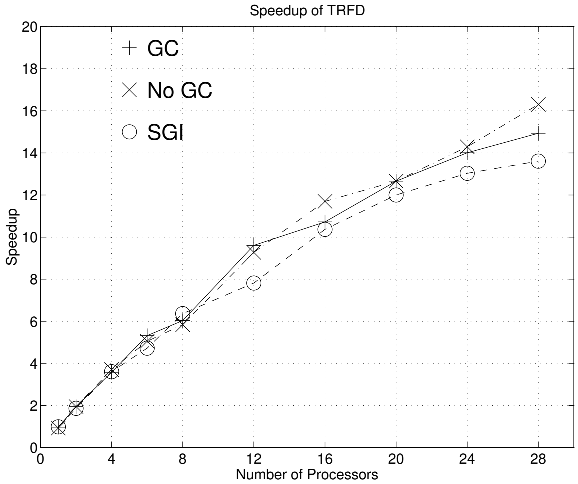


Figure 8: Speedup of TRFD, comparing the performance of *nano*Threaded code with that using SGI parallel loops.

without granularity control (No GC), and parallel loop constructs provided by the SGI compiler (SGI). This benchmark has no functional parallelism, but still *nano*Threads performs better than the SGI parallel loops. The granularity control version does not perform as well as the version without this control. This is a result of poor load balance resulting from the inhibited parallelism due to granularity control. This is an example of the case in which the smallest tasks are much larger than the scheduling overhead. TRFD has a limited amount of large grain parallelism and, therefore, the cost of unrestricted parallelism (no granularity control) is negligible. On the other hand, granularity control restricts some of the parallelism, in detriment of load balance. The SGI compiler restrict parallelism even further since it does not exploit nested parallelism, thus further hampering load balance.

Figure 9 shows the speedup curves for benchmarks compiled with the autoscheduling compiler. Quicksort (QUICK) only has functional parallelism. Versions with (GC) and without (No GC) granularity control are compared. We observe a 75% improvement on the speedup for 12 processors when granularity control is used. As with Strassen's algorithm, granularity control here avoids an exponential explosion of tasks.

For CFD we compared four versions: autoscheduling compiler with loop parallelism only (LP), autoscheduling compiler with functional parallelism only (FP), autoscheduling compiler with loop and functional parallelism (LP and FP), SGI compiler with loop parallelism only (SGI). We note that the best performance was obtained by exploiting functional parallelism only. This is mainly the result of a better cache behavior in the functional parallel version. Each task operates on entire independent matrices and therefore there is little cache interference between processors. In the loop parallel versions, processor operate concurrently on the same matrices. The SGI compiler was better than the autoscheduling compiler in exploiting loop parallelism.

9.2 Improvements due to Granularity Control

Figure 10 lists the improvement in execution time as a function of the number of processors when granularity control is used for six of the benchmarks. Let t_{PG} be the execution time on P processors with granularity control and t_{PN}

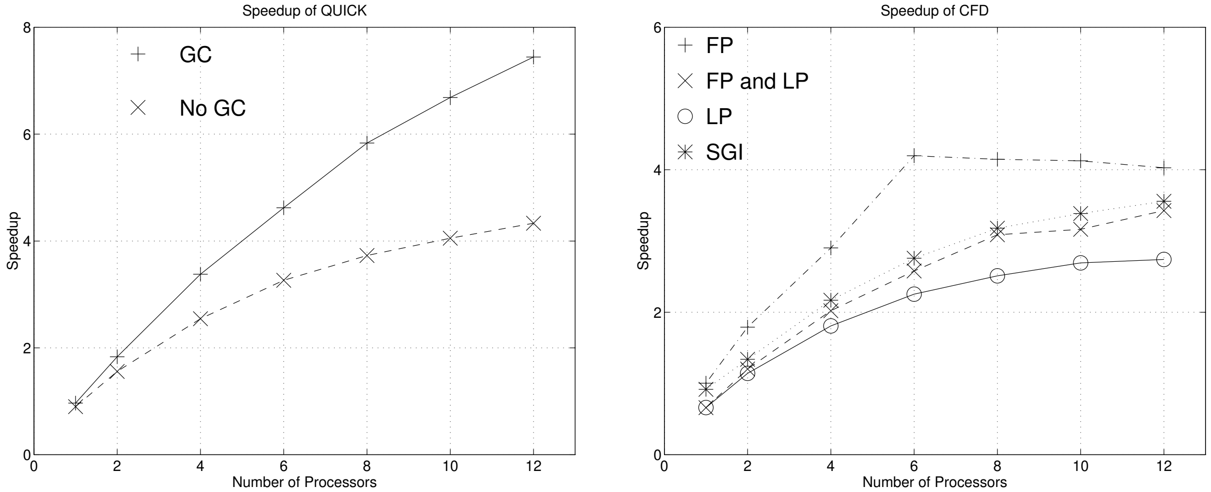


Figure 9: Speedups for QUICK and CFD codes produced by the autoscheduling compiler. A version of CFD using SGI loops was also implemented. These plots show the benefits of granularity control and functional parallelism.

be the execution time on P processors without granularity control. The percentage improvement for P processors is defined as:

$$I_{\%P} = \frac{t_{PN} - t_{PG}}{t_{PN}} \times 100 \tag{19}$$

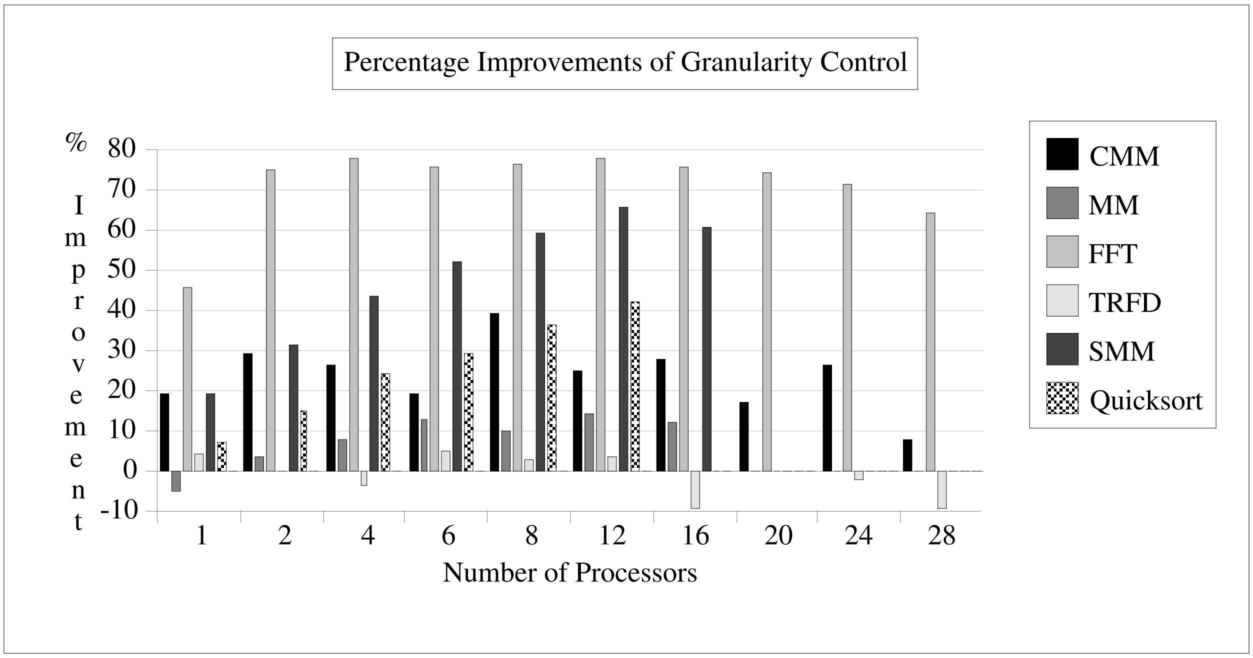
The effect of granularity control varies significantly depending on the characteristics of the algorithm and the size of the smallest task. Granularity control can make a large difference in performance, especially when the available parallelism is not large (FFT2, SMM). In the case where granularity control performs worse (TRFD), the difference is small.

10 Related Work

A variety of thread management issues and implementation alternatives are considered in [1]. It includes analysis and implementation results for different types of queue management and lock synchronization techniques.

Scheduler Activations [2] and the Process Control Approach [23] both offer user-level scheduling and dynamic adaptation to changing environments. Scheduler activations supply an execution context that may be manipulated by a user-level scheduler or the kernel to allow for user-level scheduling with the flexibility and power of kernel-level threads. Allowed kernel manipulations include the granting or removing of scheduler activations. The process control approach allows the user to request processors and gives the user control over when to release processors. Processors are released by the user on request of the kernel at *safe points* to prevent deadlock and similar problems.

Considerable work has been done on the exploitation of functional and loop parallelism by the Paradigm [22] [21] group at the University of Illinois. They rely on a *Macro Dataflow Graph* which is a directed acyclic task graph in which the nodes are weighted with the communication and computation time. Their approach performs static partitioning for data and computations on a distributed system to minimize total runtime.



Percentage Improvement Due To Granularity Control										
Number Procs	1	2	4	6	8	12	16	20	24	28
CMM	19.6	29.0	26.7	19.6	39.5	25.0	28.2	17.2	26.3	8.2
MM	-5.2	3.3	7.9	12.8	9.8	14.0	12.4	–	–	–
FFT2	45.7	75.2	78.1	75.7	76.5	78.2	76.0	74.6	71.4	64.3
TRFD	4.6	-0.1	-3.6	5.1	3.2	3.5	-9.1	0.0	-2.0	-9.2
SMM	19.2	31.7	43.9	52.2	59.5	65.5	61.0	–	–	–
Quicksort	6.96	15.04	24.64	29.39	36.11	41.79	–	–	–	–

Figure 10: The effect of granularity control.

Techniques to exploit nested parallelism include switch-stacks[5] and process control blocks (PCBs) [12]. A PCB for a parallel loop is used to schedule the iterations of that loop. Reference [12] discusses heuristics that strike a balance between efficient allocation of PCBs versus load balancing problems that arise from barrier synchronization in nested parallel loops. Switch-stacks handle nested parallelism by actually swapping stacks between processors so no one processor is left idle waiting for another to finish.

The *Psyche* [15] system has facilities for user-level threads in which many tasks normally performed by the kernel, such as interrupt handling and preemptions, are handled at the user-level. Like *nano*Threads, it relies on multiple virtual processors sharing the same address space.

Chores [6] is a paradigm for the exploitation of loop and functional parallelism. It allows dynamic scheduling at the user-level and the expression of dependences between tasks without explicit synchronization.

11 Conclusion

In this paper we have demonstrated the benefits of exploiting functional parallelism and performing dynamic granularity control. Our implementation is based on a user-level threads model that performs dynamic task scheduling.

In particular, we have shown that the exploitation of nested functional parallelism is beneficial both in terms of increasing the amount of high-level parallelism and improving the load balance of parallel loops. We have also shown the potential for significant improvements by dynamically controlling the granularity of exploited parallelism.

Results from synthetic benchmarks were used to verify analytical models of performance. Measurements with application kernels demonstrated the efficiency of our approach for real scientific computations on an existing commercial shared-memory multiprocessor.

References

[1] Thomas Anderson, Edward Lazowska, and Henry Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12), December 1989.

[2] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *13th ACM Symposium on Operating Systems Principles*, pages 95–109. ACM Sigops, October 1991.

[3] D. Bailey et al. The NAS parallel benchmarks. Technical Report RNR-95-007, RNR, March 1994.

[4] Carl J. Beckmann. *Hardware and Software for Functional and Fine Grain Parallelism*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1993.

[5] Jyh-Herng Chow and Williams Ludwell Harrison. Switch-stacks: A scheme for microtasking nested parallel loops. In *Supercomputing 90*, pages 190–199, Nov. 1990.

[6] Derek Eager and John Zahorjan. Chores: Enhanced run-time support for shared-memory parallel computing. *ACM Transactions on Computer Systems*, 11(1), February 1993.

[7] Mike Galles and Eric Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. Silicon Graphics Technical Report. Available from `http://www.sgi.com`.

[8] M. Girkar and C. D. Polychronopoulos. The HTG: An intermediate representation for programs based on control and data dependences. Technical Report 1046, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, May 1991.

[9] Milind Girkar and Constantine Polychronopoulos. Automatic detection and generation of unstructured parallelism in ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2), April 1992.

[10] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, 1989.

[11] Anoop Gupta, Andrew Tucker, and Luis Stevens. Making effective use of shared-memory multiprocessors: The process control approach. Technical Report CSL-TR-91-475A, Computer Systems Laboratory, Stanford University, 1991.

[12] S. F. Hummel and E. Schonberg. Low-overhead scheduling of nested parallelism. *IBM J. Res. Develp.*, 35(5/6):743–765, Sept/Nov 1991.

[13] D. E. Knuth. *The Art of Computer Programming, Vol. 3 Sorting and Searching*. Addison-Wesley, Reading, Mass., 1973.

[14] S. L. Lyons, T. J. Hanratty, and J. B. MacLaughlin. Large-scale computer simulation of fully developed channel flow with heat transfer. *International Journal of Numerical Methods for Fluids*, 13:999–1028, 1991.

[15] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 110–121, October 1991.

[16] Milind Girkar. *Functional Parallelism: Theoretical Foundations and Implementation*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1992.

[17] José E. Moreira. *On the Implementation and Effectiveness of Autoscheduling for Shared-Memory Multiprocessors*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, February 1995.

[18] C. D. Polychronopoulos, M. B. Girkar, Mohammad R. Haghighat, C. L. Lee, B. Leung, and D. A. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. *International Journal of High Speed Computing*, 1(1):45–72, May 1989.

[19] Constantine Polychronopoulos, Nawaf Bitar, and Steve Kleiman. *nano*threads: A user-level threads architecture. In *Proceedings of the ACM Symposium on Principles of Operating Systems*, 1993.

[20] Constantine D. Polychronopoulos. Autoscheduling: Control flow and data flow come together. Technical Report 1058, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1990.

[21] Shankar Ramaswamy and Prithviraj Banerjee. Processor Allocation and Scheduling of Macro Dataflow Graphs on Distributed Memory Multicomputers by the PARADIGM Compiler. In *International Conference on Parallel Processing*, pages II:134–138, St. Charles, IL, August 1993.

[22] Shankar Ramaswamy, Sachin Sapatnekar, and Prithviraj Banerjee. A Framework for Exploiting Data and Functional Parallelism on Distributed Memory Multicomputers. Technical Report CRHC-94-10, Center for Reliable and High Performance Computing, University of Illinois, 1994.

[23] Andrew Tucker. *Efficient Scheduling on Multiprogrammed Shared-Memory Multiprocessors*. PhD thesis, Stanford University, December 1993.