Autoscheduling in a Distributed Shared-Memory Environment [1]

José E. Moreira[†] Constantine D. Polychronopoulos[†] Akira FUKUDA[‡]
{moreira,cdp}@csrd.uiuc.edu *fukuda@is.aist-nara.ac.jp*

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

‡ Graduate School of Information Science
Nara Institute of Science and Technology
8916-5 Takayama Ikoma Nara 630-02, Japan

Abstract

The ease of programming and compiling for the shared memory multiprocessor model, coupled with the scalability and cost advantages of distributed memory computers, give an obvious appeal to distributed shared memory architectures. In this paper we discuss the design and implementation issues of a dynamic data management and scheduling environment for distributed shared memory architectures. Unlike the predominantly static approaches used on distributed and message passing machines, we advocate the advantages of dynamic resource allocation, especially in the case of multi-user environments. We propose hybrid data and work distribution techniques that adjust to variations in the physical partition, achieving better load balance than purely static schemes. We present the architecture of our execution environment and discuss implementation details of some of the critical components. Preliminary results using benchmarks of representative execution profiles support our main thesis: With minimal control, the load balancing and resource utilization advantages offered by dynamic methods often outweigh the disadvantage of increased memory latency stemming from slightly compromised data locality, and perhaps additional run-time overhead.

分散共有メモリ型並列計算機の自動スケジューリング手法

ジョゼ E. モレラ[†] コンスタンチン D. ポリクロノポーラス[†] 福田 晃[‡]

† スーパーコンピュータ研究開発センター統合科学研究所
イリノイ大学ウルバナ・シャンペイン校
1308 W. Main St. Urbana, IL 61801-2307 – USA

‡ 奈良先端科学技術大学院大学
情報科学研究科
奈良県生駒市高山町 8916-5

Abstract

分散共有メモリ型アーキテクチャの魅力は、分散メモリ型並列計算機のスケーラビリティやコスト・パフォーマンスの良さに加えて、共有メモリ型並列計算機モデルでの並列プログラミングや並列化コンパイラの容易さにある。そこで本稿では、分散共有メモリ型アーキテクチャに対する、動的なデータ管理や、スケジューリング環境の設計や実装といった問題について考察する。メッセージパッシング型並列計算機では、静的なアプローチが圧倒的に多いが、ここで我々は、特にマルチユーザー環境の場合に、動的リソース管理が有効であることを主張する。つまり、静的な方法より、よい負荷分散が得られ、物理的分割の変化にも馴染む、ハイブリッドなデータとタスクの分割テクニックを提案するのである。また、実行環境のアーキテクチャを示し、クリティカルな構成要素の一部の実装について詳しく述べる。典型的な実行プロファイル方式のベンチマーク・テストを用いた予備結果では、我々の提案する手法の優位性が示されている。すなわち、最小の制御による動的手法がもたらす負荷分散とリソースの高利用性という利点は、不完全なデータ局所性から生じる通信遅延の増加、あるいは実行時の付加的なオーバーヘッドといった欠点を、十分に補い余りある。

[1]This work was supported by the Office of Naval Research under grant N00014-94-1-0234.

1 Introduction

The distributed shared memory architecture offers the programming and compiling advantages of shared memory, and at the same time, the scalability and cost advantages of distributed memory architectures. By virtue of being the successors to shared memory and distributed memory (or more precisely message-passing) architectures, distributed shared memory (or DSM) machines have also inherited some of the programming methodologies and architectural features of their predecessors. The indisputable importance of data locality has favored static approaches to data placement and scheduling on DSM machines, just like it did on earlier generation message-passing. Thus far, this has also been the underlying guide in the design of HPF[7], Fortran D[3], and to a lesser degree of the Cray MPP Fortran[13].

In this paper we give preliminary evidence of the importance of dynamic solutions to the data management and scheduling problems. We present the architecture of distributed autoscheduling, a dynamic environment adopted from our previous work on uniform access shared memory environments. We argue that dynamic methods can achieve better load balance, and improve resource utilization (especially in multi-user environments), while they can still exploit data locality. In addition to the performance advantages, dynamic environments can deal more effectively with hardware failures; they can also take advantage of unexpected changes in the availability of resources (processors) which is common in multiprogramming environments.

The major aspect of our new model is incorporation of capabilities for exploiting data locality while preserving dynamic scheduling and load balancing. This paper is organized as follows: Section 2 describes our target machine architecture and our program model. Section 3 describes the architecture of the run-time system that implements autoscheduling in a shared-memory multiprocessor. Data and load distribution issues are addressed in Section 4. The storage of data structures in distributed autoscheduling is described in Section 5 and the task queue in Section 6. Experimental results are presented in Section 7, related work is discussed in Section 8, and concluding remarks are given in Section 9.

2 Machine and Program Model

We consider specifically as target architecture a multiprocessor machine with a shared virtual and physical address space, built with commercial RISC microprocessors. The global physical memory is divided into *modules*, and each module is connected to one particular processor. A processor can access its memory module faster than that of another processor, thus creating a nonuniform memory. All the processors assigned to a process use the same virtual to physical mapping, so a virtual address always corresponds to the same physical address.

The program model for autoscheduling is the *hierarchical task graph* (HTG), an intermediate program representation that encapsulates data and control dependences at various granularity levels, and from which autoscheduling code is generated. It 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 [2, 4, 9, 14].

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 tasks of a program and can be of three types: *simple*, *compound*, and *loop*. A simple node represents the smallest schedulable unit of computation. A compound node is recursively defined as an HTG and is therefore composed by smaller nodes. A loop node represents a task that is either a serial loop (all iterations must be executed in order) or a parallel loop (the iterations can be executed simultaneously in any order).

From the information on control and data dependences, an *execution tag* $\varepsilon(x)$ is derived for each node x. The execution tag is an expression on boolean flags that mark the execution of nodes and arcs in the HTG. Whenever the values of the boolean flags cause an execution tag to evaluate to TRUE, the corresponding node has been *enabled* and is ready to execute.

3 General Architecture of the Run-Time System

The execution environment of an autoscheduling program consists of the code and data area, defining an address space, a (time-variant) set of $n(t)$ physical processors assigned to that address space, a *task ready queue*, and a *cactus-stack*. Using conventional terminology, an executing program is a *process*.

The set of processors $n(t)$ assigned to a process at time t ($n(t) : \mathcal{Z}^+ \rightarrow [0..P]$) is called a *partition* and is controlled by the operating system, which distributes the available processors in a machine to the processes. Scheduling policies at the OS level are beyond the scope of this paper and will not be discussed further.

The *task ready queue* is a (user space) data structure that holds *task identifiers* (task-ids) of the ready tasks. A task identifier contains task-specific information, including the starting address of the task and a pointer to context information, such as the *activation frame* of the task. Each processor allocated to a process executes a loop of the following type:

```
do
    get task-id from task queue
    execute task
forever
```

When a task completes execution, its *drive code* injected by the compiler evaluates the execution tags $\varepsilon(x)$ of the affected tasks and the new ready tasks are inserted in the queue; this is the basic scheduling operation. The execution of a program begins with its START task in the ready queue and terminates when its STOP task finishes.

The *activation frames* of a parallel program cannot be implemented with the simple stack structure normally used in sequential programs, since several instances of subroutines and loop iterations can be active at the same time. Instead, a *cactus-stack*, equivalent to a (dynamic) tree of activation frames, is used. All data structures reside in shared memory, and all activation frames are organized in the cactus-stack. Stacks are not associated with processors. The only information processors need in order to execute a task is the beginning address of the code and the base address of the activation frame. Therefore, any processor can execute any task, and switching between tasks involves only loading the code pointer and frame pointer in the appropriate registers.

The drive code injected by the compiler before and after each task node in the HTG take the form of *entry* and *exit* blocks respectively. The major functions of the entry and exit blocks are as follows:

ENTRY Blocks:
- Allocate private activation frame
- Link to parent activation frame
- Execute initialization code
- Loop scheduling policy
- Granularity control

EXIT Blocks:
- Barrier synchronization
- Update control & data dependences
- Testing of execution tags
- Queue ready tasks
- Granularity control

Granularity control, listed in both the entry and exit blocks, is a key feature of autoscheduling. It works by dynamically deciding which compound tasks are to be split into smaller tasks for parallel execution, and which are to be executed as serial code, in order to achieve a good balance between the number and size of parallel tasks. A parallel program must generate enough tasks to keep all the processors in its partition busy. However, as the number of tasks increases, so does the overhead for managing these tasks. By controlling the granularity of tasks, autoscheduling avoids the generation of unnecessary parallelism that cannot exploit any more processing power, but may add to the overhead.

Autoscheduling is a dynamic scheduling environment, adjusting the number and size of tasks generated by a process at run-time in order to better exploit the resources available to a process at any given time. When operating in multiprogrammed mode the resources of a multiprocessor must be divided among computing processes executing simultaneously. Ideally, the amount of resources that an individual process receives should be a function of the total workload on the machine. Programs written for fixed configurations of processors are not appropriate for execution in such dynamic environments. Autoscheduling uses the partition allocated to a process effectively and efficiently, and adjusts to variations in the partition, making multiprogramming in multiprocessors very efficient.

4 Data and Load Distribution

Implementation of autoscheduling on a flat shared memory machine with uniform memory access is facilitated by not having to deal with the distribution of data. Since all memory accesses involve the same latency, load balance can be achieved by having a single queue of ready tasks and letting idle processors fetch the task at the head of the queue.

In a distributed shared memory machine with nonuniform memory access, the access to remote memory can be orders of magnitude slower than access to local memory. Therefore, it is important to distribute data and computations in such a manner that the computations performed by a processor involve mostly local memory accesses. In general, the higher the locality of access, the greater the performance. However, enforcing locality can degrade load balance, since the predefined distribution of work leaves less room for dynamic adjustments of load.

Autoscheduling on distributed shared memory machines necessitates data distribution in order to achieve locality and hence performance. However, in order to preserve the main properties of autoscheduling that facilitate the efficient execution of simultaneous concurrent processes, we consider data distribution under the following constraints: (1) we assume that the partition assigned to a process is not known until run-time and is time-variant, and (2) load balance must be preserved. These goals are achieved through the following mechanism:

- Data are partitioned at compile time across sets of virtual processors defined by the user (or by a smart compiler), in a manner similar to HPF [7].
- Virtual processors are assigned on demand to physical processors during run-time. Virtual processors can migrate between physical processors during the execution of a process and may even, at times, not be assigned to any physical processor. The precise mechanism is discussed in Section 5.

- Data mapped to a virtual processor are allocated on demand in the local memory of the physical processor to which the virtual processor is assigned at the time of allocation. Data can also migrate from one processor to another on demand. The precise mechanism for data allocation and migration is also discussed in Section 5.
- Iterations of parallel loops and tasks can be mapped to virtual processors. This assignment is similar to a *hint* that these iterations and tasks should be executed in a particular virtual processor (most likely because they use mostly data local to the processor). These tasks and iterations can still be executed by any processor because the shared address space in autoscheduling allows any task to be executed in any processor.

Our approach to data distribution uses features from Fortran D [3], HPF [7], and Cray MPP Fortran [13], and works as follows, at the source language level:

1. The user defines an array $V(P_1, P_2, \ldots, P_m)$, of *virtual processors.*
2. The user defines a multi-dimensional *decomposition* $D(N_1, N_2, \ldots, N_n)$ that expresses the problem domain of the operations to be performed.
3. The user defines a distribution $T : D \rightarrow V$ from decomposition D to virtual processor array V which assigns points in D to processors in V. A given distribution D can have only one active mapping to a processor array.
4. The user aligns each of the array data structures $R_1, R_2, \ldots$, used in the computations, to the decomposition D, using alignments $A_1 : R_1 \rightarrow D, A_2 : R_2 \rightarrow D, \ldots$, respectively. Each element of $R_1(i_1, i_2, \ldots, i_p)$ of R_1 is aligned with an element $D(j_1, j_2, \ldots, j_n)$ of D, and will be placed in the local memory of the processor to which $D(j_1, j_2, \ldots, j_n)$ is mapped at the moment of allocation.
5. The user specifies distribution of loop work with the construct

   $$\textbf{doall } E \textbf{ on } D \ \{\mathcal{B}\}$$

   where D is a decomposition, E is an iteration space descriptor that selects a subset of the points of D, and $\mathcal{B}$ is the body of the loop. This construct asserts that the iteration corresponding to $D(j_1, j_2, \ldots, j_n)$ should be *preferentially* executed by the virtual processor to which this point of D is mapped. Preferred locations for the execution of individual tasks can be specified with the construct

   $$\textbf{task } x \textbf{ on } v \ \{\mathcal{T}\}$$

   which asserts that task x, with body $\mathcal{T}$, is to be preferentially executed on virtual processor v.

Example: Let X, Y, and Z be data arrays, and the computation

$$Z_\triangle = X_\triangle + Y_\triangle^T$$

is to be performed, where $M_\triangle$ represents the lower triangular half of matrix M:

```
PARAMETER (N = 100)
PROCESSOR P(4,4)
DISTRIBUTION D(N,N)
DISTRIBUTE D(15,15) ONTO P
REAL X(N,N), Y(N,N), Z(N,N)
ALIGN X(I,J) WITH D(I,J)
ALIGN Z(I,J) WITH D(I,J)
ALIGN Y(I,J) WITH D(J,I)
DOALL (I,J) = (1:100,1:I) ON D
  Z(I,J) = X(I,J) + Y(J,I)
ENDDOALL
```

5 Distribution of Activation Frames

Activation frames are created dynamically in virtual memory by three types of events:

1. *At program start:* The global data activation frame is created at this point. It often holds the largest data structures. This activation frame is shared by all the processors that join the execution of the program. It is deallocated when the program terminates.
2. *At function start:* When a function begins execution it is necessary, in general, to create an activation frame for its local variables. This activation frame is shared by all the processors that cooperate on the execution of that function. A function activation frame is deallocated when the function terminates.
3. *At parallel loop start:* Activation frames are also created for parallel loops with local variables. One activation frame is created for each physical processor participating in the execution of the loop[2].

The basic requirement for the layout of an activation frame or variable size array in virtual memory is that data mapped to different virtual processors must reside in separate virtual memory pages. This way, each virtual memory page is uniquely associated with a virtual processor. The physical layout of an activation frame is built during run time, together with the mappings M from virtual processor arrays to the physical processors.

[2]One activation frame per processor is sufficient because in our execution model there can only be a maximum of P (number of physical processors) iterations of any given instance of a parallel loop under simultaneous execution.

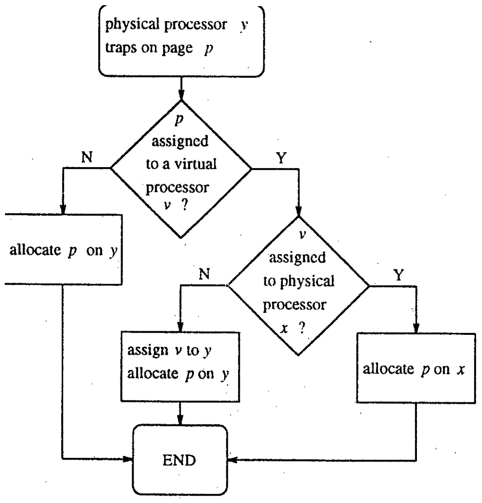
The scheme works as follows: When a processor array A is defined, the mapping M is empty, and when an activation frame is created, all its virtual pages are initially unmapped. Whenever a virtual page is accessed for the first time, a page fault occurs in the physical processor y that performed such access, which generates a *user space trap*, handled by a routine inserted by the compiler. The trap checks to which virtual processor $v \in A$ this page is mapped. If the page is not mapped to any virtual processor, because it contains non-distributed data, then a call to the operating system is made to allocate a physical frame for the virtual page in the local memory of processor y. If the page is mapped to virtual processor v, but v is not currently assigned to any physical processor, then an addition is made to M to assign v to physical processor y[3], and a call to the operating system is made to allocate a physical frame in the local memory of processor y. If the page is mapped to virtual processor v, and v is already assigned to a physical processor x, then a call to the operating system is made to allocate a physical frame in the local memory of processor x. Figure 1 is a flowchart of this mechanism.

If a page has to be swapped out of memory, then the (virtual $\rightarrow$ physical) memory mapping is updated to reflect that, and the next access to that page will cause a page fault; the same criteria as above will be used to determine where to load the page. If a physical processor has to leave a partition, then all the mappings between virtual processors and this physical processor have to be deleted. The virtual processors that lost this mapping will remain unassigned to any physical processor until an unmapped page associated with them is referenced. Figure 2 illustrates a situation where there is data migration.

6 The Distributed Task Queue

For a distributed shared memory MPP, a single task queue causes unacceptable contention, especially when task sizes are small, the number of processors is large, or both. The single queue also makes it difficult to select specific processors for the execution of each task, which is desirable when the code exhibits a high degree of data locality. The distributed queue scheme proposed here uses a two-level queue organization: one task queue per processor (*local* queue), and one additional task queue for the entire partition (the *central* queue). All queues reside in the shared address space, and any queue can be accessed by any processor in the partition. The enqueueing/dequeueing policies work as follows:

[3]The central issue of this scheme is that assignments of virtual processors to physical processors are done on demand. Instead of assigning the virtual processor to the physical processor where the fault occurred, alternative policies such as round robin and least loaded processor can also be used.



physical processor y traps on page p
p assigned to a virtual processor v ?
N
Y
allocate p on y
v assigned to physical processor x ?
N
Y
assign v to y allocate p on y
allocate p on x
END

Figure 1: Flowchart for the mechanism that assigns virtual to physical processors.

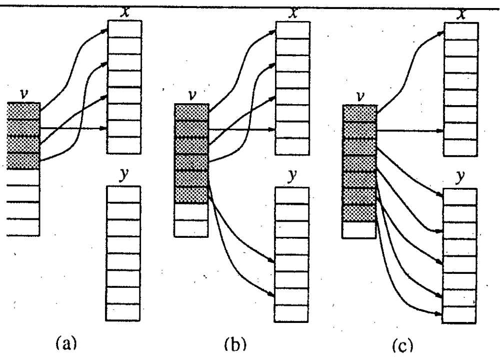


Figure 2: Migration of data assigned to virtual processor v from physical processor x to physical processor y. (a) Originally, v is mapped to x and all the data assigned to v are stored in x. (b) v is reassigned to y, old data remain in x but new data are allocated in y. (c) As time passes, old data are purged from x and reassigned to y.

- *Dequeueing:* A processor always tries to dequeue tasks first from its local queue. If its local queue is empty, then it tries to dequeue from the central queue. Only if the central queue is empty will a processor then start searching for tasks from other processor queues.

- *Enqueueing:* When a task is enabled by an exit block executing in a given processor, the criteria used to decide where to queue the tasks are the following, applied in the order enumerated:

1. If the compiler has identified a preferential virtual processor for the execution of this task, either by analysis or through directives, and this virtual processor is assigned to a physical processor, then the task is enqueued in the local queue of that physical processor. If the virtual processor in unassigned, then the task is placed in the central queue.
2. If the task is *large*[4] and has no preferential virtual processor to execute, then it is placed in the central queue (overhead is less significant).
3. If the task has no preferential virtual processor to execute and it is *small*, it is enqueued in the processor local queue, unless this queue is full, in which case it is enqueued in the central queue.

When a physical processor is forced to leave a partition, it must transfer the tasks in its queue to the central queue, including the continuation of the task that was suspended, (if the processor was interrupted in the middle of a task).

7 Results

The experimental results presented here were obtained through simulation of the execution of high-level code in the autoscheduling environment described in the paper. The benchmark suite consists of five simple programs that illustrate the main advantages of dynamic scheduling over static, and a sixth program that demonstrates the benefits of exploiting functional parallelism in this environment. Each of the first five programs has some properties that allow the comparison between static and dynamic scheduling for different types of computation:

1. *Block matrix add:* regular and balanced computation, poor temporal cache locality (only benefit is from prefetching a cache line).
2. *Block matrix multiply:* regular and balanced computation, with good cache locality (both temporal and space).
3. *Block triangular matrix add:* regular but unbalanced computation, poor temporal cache locality.
4. *Block triangular matrix multiply:* regular but unbalanced computation, good cache locality.
5. *Life:* irregular and unbalanced, good cache locality.

Each of these programs uses three matrices: A, B, and C. Each matrix is of size $N \times N$ and blocked as a $p \times p$ matrix of blocks of size $m \times m$, where $N = mp$. The computations are performed on a linear array V of p virtual processors. A decomposition D of size $N \times N$ is used to control the assignment of array elements and iterations to the virtual processors. The computations are performed by a two dimensional **doall** loop with iteration space I. Elements $A(i,j)$, $B(i,j)$, and $C(i,j)$, and iteration $I(i,j)$ are all mapped to decomposition element $D(i,j)$. Decomposition D is distributed across the processor array in a block row manner.

The sixth program is a complex matrix multiply, implemented as four independent real matrix multiplies followed by two real matrix adds. By using functional parallelism and performing all four multiplications concurrently, a four-fold improvement in exploitable parallelism is obtained as compared to the case where only the loop parallelism in each multiplication is exploited. The matrices are of size $N \times N$, blocked as before with blocks of size $m \times m$, and a linear array V of p virtual processors is used ($N = mp$). The matrix multiplication is parallelized along only one dimension, so the maximum loop parallelism available is p.

The results presented here are speedup curves with respect to the physical number of processors used in the execution of the programs. This number is kept constant during each execution. In both the static and the dynamic scheduling the virtual processors are first assigned cyclically to the physical processors, and time starts to count after this assignment. The difference in the dynamic scheduling is that a processor is allowed to *steal* iterations from virtual processors not assigned to it, whereas in static scheduling it will only execute those iterations mapped to virtual processors assigned to it. For the complex matrix multiply, the static scheme does not exploit functional parallelism.

In the simulations, execution time is computed by counting the work in arithmetic operations (1 time unit), memory reads (1 time unit for cache, 10 for local, 100 for remote), memory writes (1 time unit), and iteration fetches (in the case of dynamic scheduling only, 10 time units for local, 100 or 1000 for remote). Cache line size is 8 words, and a whole line is prefetched when a word is accessed. The cache in each processor is large enough to hold the entire problem set, so there are only cold misses. In the following plots (Figures 3 to 8), the dashed line represents linear speedup. The bullets ($\bullet$) represent static scheduling, the diamonds ($\diamond$) represent dynamic scheduling with a cost of 100 units for remote iteration fetch, and the stars ($\star$) represent dynamic scheduling with a cost of 1000 units for remote iteration fetch. $N = 256, p = 32$, and $m = 8$, except for Figure 8, where $N = 64$, $p = 8$, and $m = 8$.

[4] A threshold size that differentiates between large and small tasks can be computed using machine-specific parameters and other criteria, such as overhead for remote task enqueueing and dequeueing.

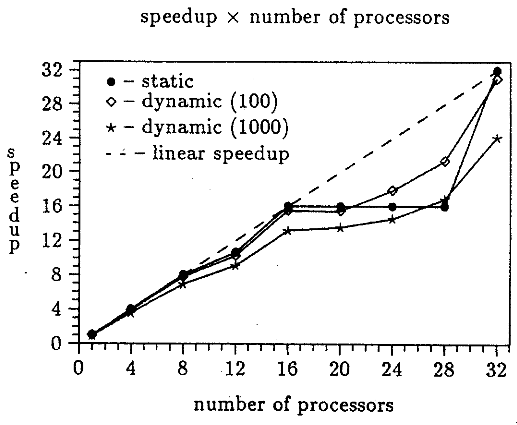


Figure 3: Block matrix add.

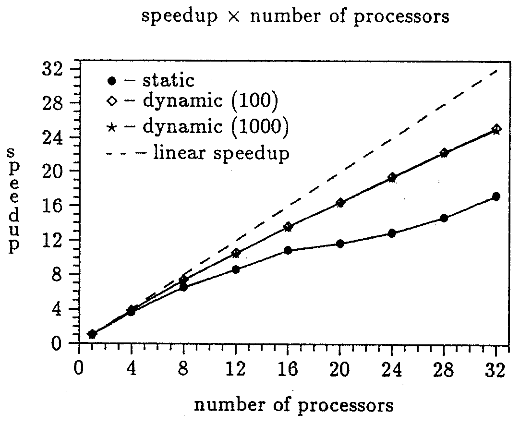


Figure 6: Block triangular matrix multiply.

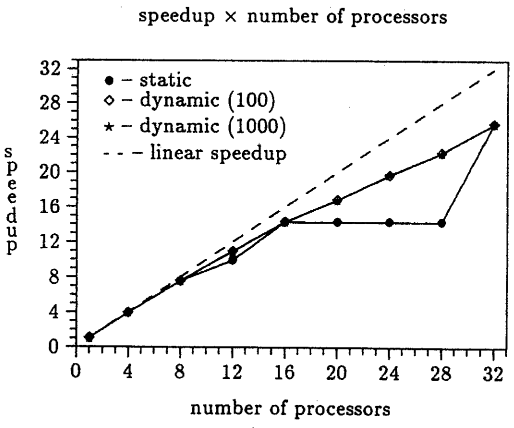


Figure 4: Block matrix multiply.

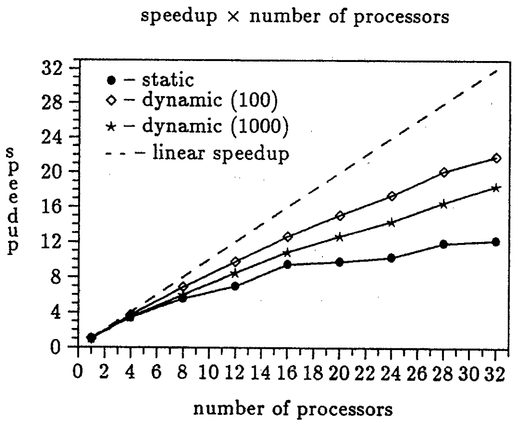


Figure 7: Life.

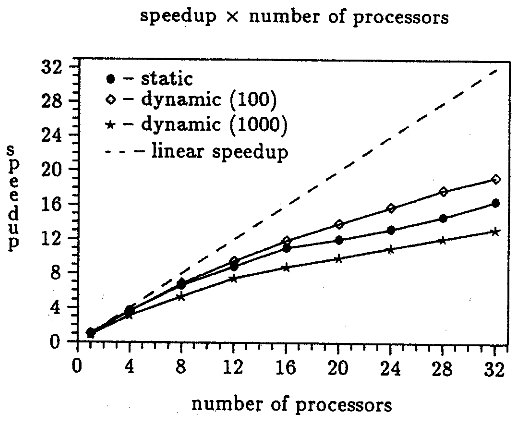


Figure 5: Block triangular matrix add.

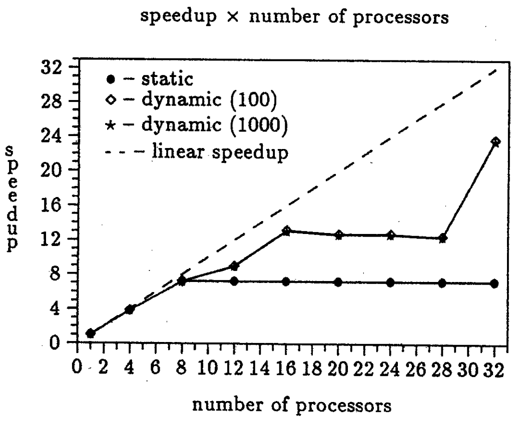


Figure 8: Complex matrix multiply.

8 Related Work

For a survey of distributed shared memory architectures, see [11]. Information on the architecture and programming model of the Cray T3D can be found in [12, 13].

Examples of languages that support the specification of data distribution include Fortran D [3], Vienna Fortran 90 [1], High Performance Fortran [7], and Cray MPP Fortran [13]. Compilation issues for these languages, including the actual implementation of the data distributions, code generation, and communication of distributed data across procedure boundaries, are discussed in [6, 8, 15].

Research is also being conducted on the field of automatic data partitioning techniques [5, 10]. In particular, [10] extends the HTG with information on data accesses, and from there derives data partitioning and processor assignment.

9 Conclusions

We have addressed many of the issues involved on the implementation of autoscheduling in a distributed shared-memory environment: data distribution, load distribution, task queue implementation, enqueueing and dequeueing policies, activation frame allocation, virtual to physical processor mapping, and loop distribution. Our results demonstrate that a dynamic scheduling mechanism such as autoscheduling is very competitive with static scheduling for distributed shared-memory architectures. Autoscheduling supports multiprogramming in a multiprocessor very efficiently because processes can be executed on partitions of time-variant size, allowing a dynamic space partitioning of the resources across the simultaneously executing processes.

References

[1] Siegfried Benkner, Barbara M. Chapman, and Hans P. Zima. Vienna Fortran 90. In *Proceedings of the Scalable High Performance Computing Conference SHPCC-92*, pages 51–59, 1992.

[2] 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.

[3] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koebel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical Report COMP TR90-141, Department of Computer Science, Rice University, December 1990.

[4] 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.

[5] Manish Gupta and Prithviraj Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[6] Mary W. Hall, Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Interprocedural compilation of Fortran D for MIMD distributed memory machines. In *Proceedings of Supercomputing'92*, pages 522–534, 1992.

[7] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 1.0*, May 1993.

[8] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

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

[10] Tsuneo Nakanishi, Kazuki Joe, Hideki Saito, Constantine Polychronopoulos, Akira Fukuda, and Keijiro Araki. The data partitioning graph: Extending data and control dependencies for data partitioning. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing. Ithaca, NY*, 1994.

[11] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, 24(9), August 1991.

[12] Wilfried Oed. The Cray Research Massively Parallel Processor System – CRAY T3D. Technical report, Cray Research GmbH, November 1993.

[13] Douglas M. Pase, Tom MacDonald, and Andrew Meltzer. MPP Fortran programming model. Technical report, Cray Research, Inc., March 1994.

[14] Constantine D. Polychronopoulos. Autoscheduling: Control flow and data flow come together. Technical Report 1058, CSRD, 1990.

[15] Hans P. Zima and Barbara Mary Chapman. Compiling for distributed-memory systems. *Proceedings of the IEEE*, 81(2):264–287, February 1993.