1991

Scatter Scheduling for Problems with Unpredictable Structures

Min-You Wu
Syracuse University, Center for Computational Science ; Syracuse University School of Computer and Information Science

Wei Shu
State University of New York at Buffalo, Computer Science Department

Scatter Scheduling for Problems with Unpredictable Structures

Min-You Wu
Syracuse Center for Computational Science
and School of Computer and Information Science
Syracuse University
Syracuse, NY 13244

Wei Shu
Computer Science Department
State University of New York at Buffalo
Buffalo, NY 14260

Abstract

An extended scatter scheduling was applied to problems with unpredictable, asynchronous structures. It has been found that with this simple scheduling strategy, good load balance can be reached without incurring much runtime overhead. This scheduling algorithm has been implemented on hypercube machines, and its performance is compared with other scheduling strategies.

1 Introduction

To solve an application problem on a parallel computer, the problem must be divided into many parallel actions, which are then scheduled onto individual processing elements (PEs). For the class of problems that can be partitioned with uniform computation and communication patterns, scheduling is relatively easy and straightforward. However, for the class of problems whose computation and communication requirements are unknown prior to execution time, scheduling becomes a non-trivial task. Many issues have to be considered, such as load balancing, minimization of the overhead resulting from scheduling, and communication traffic. The quality of scheduling may have a direct effect on the overall performance of parallel computation.

Scheduling strategies can be either static or dynamic. A static scheduling strategy can generate good load balance without incurring runtime overhead, but it can only schedule problems with static structures. For an irregular problem with a known structure, static scheduling can still be applied, although it may require a sophisticated algorithm. However, for more general application problems, especially those with unpredictable computational structures, a dynamic scheduling strategy becomes inevitable, although a price must be paid in runtime overhead.

Scatter scheduling has been used as a static scheduling strategy. It is a powerful method to balance the load for irregular applications and has been successfully used in many matrix applications [FJL+88]. It has been shown that, in general, scatter scheduling can produce satisfactory load balancing for matrix problems [NS90]. Scatter scheduling has not been used in applications with unpredictable structures, such as event-driven simulation, transaction analysis, computer chess, etc. For these applications, processes must be created at runtime which cannot be scheduled at compile time. They must be scheduled with dynamic scheduling strategies. An adaptive strategy is a good choice, but is usually complicated and hard to implement. In this paper, we show that the scatter scheduling approach can be extended to problems

with unpredictable structures, generating good load balance. The scheduling algorithm can simplify the design of the runtime support system, resulting in low runtime overhead and good performance.

2 Scatter Scheduling

Scatter scheduling has also been called scatter decomposition [FJL+88]. It decomposes a matrix into many pieces and schedules these pieces by interleaving them. As a prerequisite of scatter scheduling, a problem needs to be decomposed into a number of subcomputations, which are labeled using Cartesian coordinates. For example, a matrix of 16×16 can be decomposed into 16 rows, as shown in Fig. 1. Subcomputation i will be scheduled onto PE j by the scatter scheduling function

$$j = MOD(i, P),$$

where P is the number of PEs in the system. As an example, take a system with four PEs, in which rows 0, 4, 8, and 12 are scheduled to PE 0, and rows 1, 5, 9, and 13 are scheduled to PE 1, and so on.

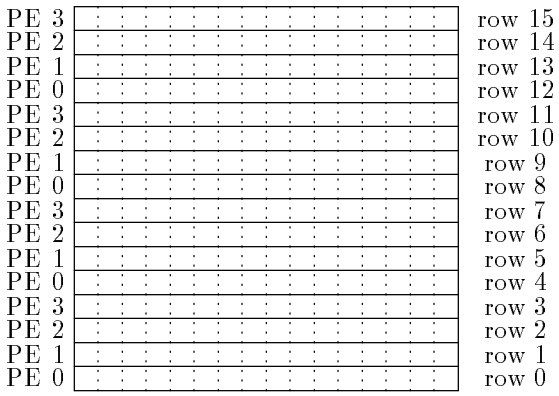

Figure 1: Scatter scheduling for matrices.

Such scatter scheduling can be applied only to problems with static structure. Compared with a regular block decomposition scheme in which a problem is equally divided into P subcomputations where P is the number of PEs, a scatter scheduling scheme is able to balance the load for applications with irregular computation density of each partition and exhibits more flexibility [NS90, Sal88]. Assume that we decompose a problem into K subcomputations, where K is larger than number P. The reason for choosing a large K is to balance the load in the case of computation density in each subcomputation being different. Theoretically, as long as K is large enough, the computation load scheduled onto each PE tends to be the same. This feature enhances scatter scheduling applied to problems with irregular structures.

For a class of problems whose computation patterns are unknown prior to execution time, this scatter scheduling strategy is not applicable, since these subcomputations cannot be labeled at compile time. We extend the scatter scheduling scheme which is presented in the next section.

3 Extended Scatter Scheduling

We extended scatter scheduling for problems with unpredictable structures. In a fully distributed approach, each PE independently participates in scheduling activity, distributing locally generated processes to other PEs.

Since processes are generated at runtime, they cannot be scheduled at compile time. Instead, processes are scheduled in a scatter fashion at runtime. In general, process scheduling is based on the *generation order* of the processes in each PE. That is, processes can be labeled as they are generated. Assume a process is the kth generated process in PE i, it will be labeled as process k in the PE. The process scheduling can be described as a function of k. That is, process k can be scheduled to PE j:

$$j = f(k).$$

The simplest scatter function could be

$$f(k) = MOD(k, P),$$

where P is the number of PEs in the system. With this function, every PE schedules the first generated process to PE 0, the second process to PE 1, and so on. However, processes may be distributed slowly in this way. To distribute the load quickly and more evenly,

we modify the function to

$$f(k) = MOD(i + k, P).$$

That is, every process schedules the first generated process to the next PE, the second process to the second next PE, and so on. For example, in a system with 4 PEs, when PE 1 generates the first four processes, they will be scheduled to PEs 2, 3, 0, and 1, respectively. The corresponding scheduling algorithm is shown in Fig. 2. An example of scheduling is shown in Fig. 3. Note that we assumed any process is scheduled earlier than those in the deeper level of the tree.

```
next = mynode + 1
for each newly generated process
  j = MOD(next, P)
  schedule the process to j
  next = next + 1
```

Figure 2: Scatter scheduling algorithm for unpredictable structures.

4 Implementation

The scatter scheduling algorithm has been implemented on the chare kernel system runing on an Intel iPSC/2 hypercube machine. The chare kernel is a runtime support system that is designed to support machine-independent parallel programming [SK91, Shu90]. The kernel is responsible for dynamically managing and scheduling parallel actions, called *chares*. Programmers use kernel primitives to create instances of chares and send messages between them without concerning themselves with mapping these chares to processors or deciding which chare to execute next. A chare may create other chares or send messages to existing ones.

Random allocation and Adaptive Contracting Within Neighborhood (ACWN) scheduling strategies have been implemented on the chare kernel [Shu90]. Athas and Seitz have proposed a global random allocation strategy [Ath87, AS88]. A random allocation strategy dictates that each PE, when it generates a new process, should send that process to a randomly chosen PE anywhere in the system. One advantage of this system is the simplicity of implementation. Like the scatter scheduling, no local load information needs to be maintained, nor is any load information sent to other PEs. It has also been shown that randomized allocation results in a respectable performance [Gru89].

The ACWN is an adaptive scheduling scheme consisting of *contracting* and *redistributing*. For the contracting phase, a newly created process is contracted m hops, where $0 \leq m \leq d$ and d is the network diameter. The contracting decision is based on the system state of each PE. The number of hops traveled so far for each process p is recorded as $p.hops$. Thus, at each PE k, for a process p, which is either created by PE k or received from other PEs, the following cases exist: if the system is heavily loaded or $p.hops \geq d$, process p will be retained locally and added to the local pool of processes; or if the system is lightly loaded and $p.hops = 0$, PE k will transfer process p to its least-loaded neighbor no matter what its own load is. Otherwise, the process will be transferred conditionally. In this way, the newly generated process p travels along the steepest load gradient to a local minimum. However, load imbalances may appear even though the contracting strategy is applied. Such imbalance may arise either due to limitations of the underlying load contracting scheme (which finds only a local minimum), or due to the different rates of consumption of processes. Moreover, in a heavily loaded state, PEs accumulate processes without sending them to any other PEs. Thus, after a PE leaves the heavily loaded state, it may own many more processes than other PEs do. These processes need to be redistributed to other PEs, as the allocation of new processes alone may not be sufficient to correct the load imbalance. The redistributing is active only when the system is not in its heavily loaded state. In the heavily loaded state, since all the neighbors of the PE have sufficient work to do, it is not necessary to balance the load between them. For further details about the scheduling algorithm, see [SK89, Shu90].

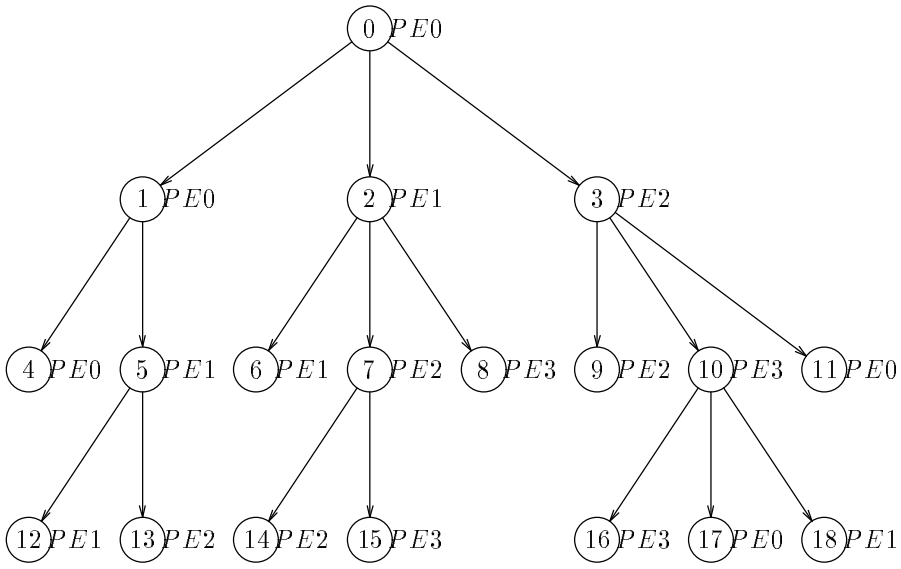

Figure 3: An example of scatter scheduling for the tree structure.

5 Experimental Results

We implemented scatter scheduling on the chare kernel and compared it to random allocation and ACWN scheduling strategies. Four application problems were used to test performance. One instance of each program was chosen for execution, that is, 10-queens, Fibonacci-32, one configuration of 15-puzzle, and the Romberg integration with 14 integrations. In the Queen problem, the grainsize is not equal, since whenever a new queen is placed, the search either successfully continues to the next row or fails. The Fibonacci problem is a regular tree-structured computation. The grainsizes of leaf processes are roughly the same. The 15-puzzle is a good example of an AI search problem. Here the iterative deepening A* algorithm was used [Kor85]. The grainsize may vary substantially, since it depends dynamically on the current estimated cost. Also, synchronization at each iteration reduces the effective parallelism. The performance of this problem was therefore not as good as others. In the Romberg integration, the evaluation of function points at each iteration was performed in parallel.

To best exploit the advantage of the scheduling algorithm, an application problem must be partitioned into processes of proper sizes. Proper granularity can be determined by the user, or by a dynamic partitioner [WS90]. Coarse granularity causes serious load imbalance, and fine granularity leads to large communication overhead and the process-generation overhead. In this experiment, the grainsizes of processes were chosen to be between 1 and 100 milliseconds, resulting from the medium-grained partitioning.

In Fig. 4, for 10-queen on 8 PEs we list the total number of processes scheduled to each PE for different scheduling strategies. We also give the idle time for each scheduling strategy. The idle time is the time in which PEs have no work to do. Comparing the scatter scheduling to the random allocation, the number of processes per PE is more even. Since both scheduling strategies are non-adaptive, even process distribution results in a balanced load, less idle time, and better performance. On the other hand, although scatter scheduling distributes processes to PEs more evenly than ACWN, the latter can balance real load better since it is adaptive to different sizes of processes and has less idle time.

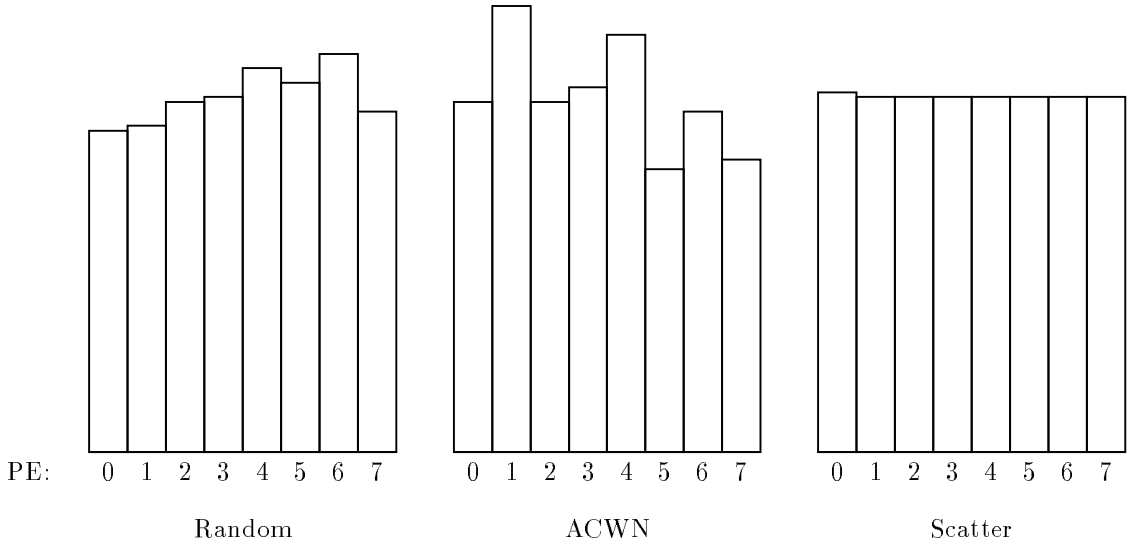

Figure 4: Distribution of processes over PEs.

In Table 1 and Figs. 5 – 8, we give the performance comparison of random allocation, ACWN, and scatter scheduling. Figs. 5 – 8 show the efficiencies for different application problems. The efficiency is defined as

$$\mu = \frac{T_S}{T_P P},$$

where T_S is the sequential execution time; T_P is the parallel execution time; and P is the number of PEs. Scatter scheduling outperforms random allocation for all four programs. In most cases, ACWN obtains better performance than scatter scheduling since it is an adaptive approach and has good locality. However, the efficiency of scatter scheduling is close to that of ACWN except for the Fibonacci problem, which has a small grain size and large communication overhead.

The extended scatter scheduling strategy spreads work quickly. The simple algorithm leads to low scheduling overhead. Scatter scheduling can achieve a good load balance and low overhead simultaneously. Although the performance of scatter scheduling is not as good as ACWN scheduling, it is much easier to implement. For comparison, the scatter scheduling method is implemented in a 10-line code but ACWN is done in a 300-line code. That results in a simple runtime support system with satisfactory performance.

Acknowledgements

The authors thank Ishfaq Ahmad for his comments, and Diane Purser for her editorial efforts. The generous support of the Center for Research on Parallel Computation is gratefully acknowledged. This work was partially supported by the National Science Foundation under Cooperative Agreement No. CCR-8809165.

References

[AS88] W. C. Athas and C. L. Seitz. Multicomputers: Message-passing concurrent computers. *IEEE Computer*, 21(8):9–24, August 1988.

[Ath87] W. C. Athas. *Fine Grain Concurrent Computations*. PhD thesis, Dept. of Computer Science, California Institute of Technology, May 1987.

[FJL+88] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors*, volume I. Prentice-Hall, 1988.

[Gru89] D. C. Grunwald. *Circuit Switched Multicomputers and Heuristic Load Placement.*

Table 1: The Performance of Three Scheduling Strategies on the Intel iPSC/2 Hypercube
Execution Time (*secs*)

		number of PEs						
		seq.	1	2	4	8	16	32
Queen 10	Random	29.5	29.9	15.5	8.41	4.72	2.56	1.69
	ACWN	29.5	29.9	15.2	7.74	4.07	2.24	1.24
	Scatter	29.5	29.8	15.6	7.97	4.19	2.23	1.45
Fibonacci 32	Random	30.0	36.2	21.2	11.4	5.99	3.21	1.73
	ACWN	30.0	36.4	18.4	9.51	4.89	2.52	1.36
	Scatter	30.0	34.9	20.2	10.9	5.68	2.93	1.55
15-puzzle/IDA*	Random	50.2	50.9	29.3	16.2	10.8	8.17	5.17
	ACWN	50.2	50.9	27.2	14.9	8.55	5.52	4.11
	Scatter	50.2	50.9	28.5	15.8	9.58	6.12	4.75
Romberg Integration 14	Random	32.7	34.1	19.5	10.7	6.23	3.81	2.47
	ACWN	32.7	34.0	18.1	9.41	5.16	3.05	1.97
	Scatter	32.7	34.1	17.8	10.1	5.55	3.39	2.38

PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, UIUCDCS-R-89-1514, September 1989.

[Kor85] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, September 1985.

[NS90] D. M. Nicol and J. H. Saltz. An analysis of scatter decomposition. *IEEE Trans. Computers*, C–39(11):1337–1345, November 1990.

[Sal88] J. Salmon. A mathematical analysis of the scattered decomposition. In *The Third Conference on Hypercube Concurrent Computers and Applications, Volume I*, pages 239–240, January 1988.

[Shu90] W. Shu. *Chare Kernel and its Implementation on Multicomputers*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, January 1990.

[SK89] W. Shu and L. V. Kale. A dynamic scheduling strategy for the chare kernel system. In *Supercomputing '89*, pages 389–398, November 1989.

[SK91] W. Shu and L. V. Kale. Chare Kernel — a runtime support system for parallel computations. *Journal of Parallel and Distributed Computing*, 11(3):198–211, March 1991.

[WS90] M. Y. Wu and W. Shu. A dynamic partitioning strategy on distributed memory systems. In *Int'l Conf. on Parallel Processing*, volume I, pages 551–552, August 1990.

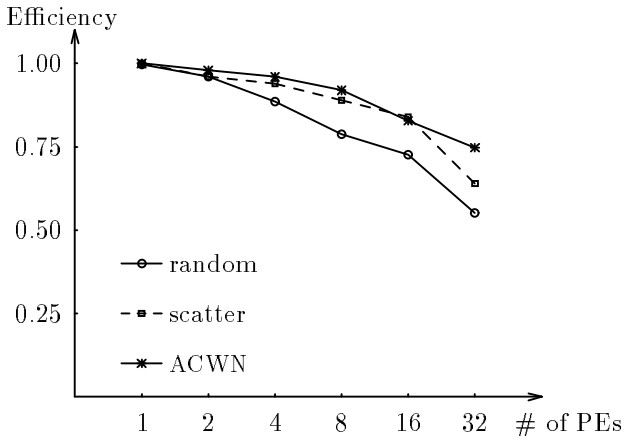

Figure 5: The efficiency comparison for 10-Queen problem.

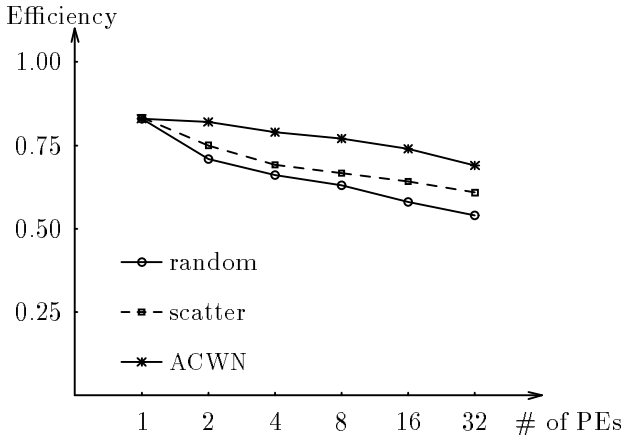

Figure 6: The efficiency comparison for Fibonacci problem.

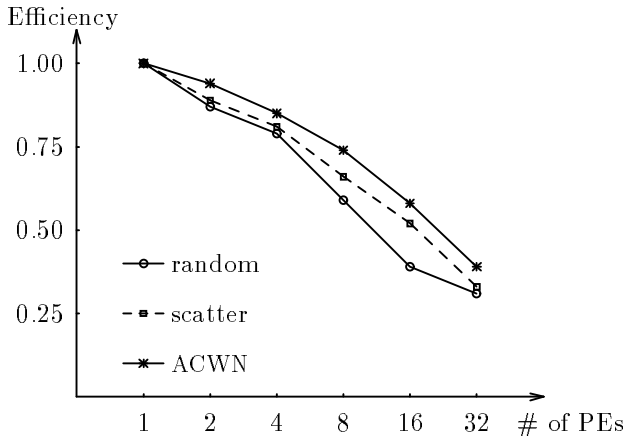

Figure 7: The efficiency comparison for 15-puzzle/IDA* problem.

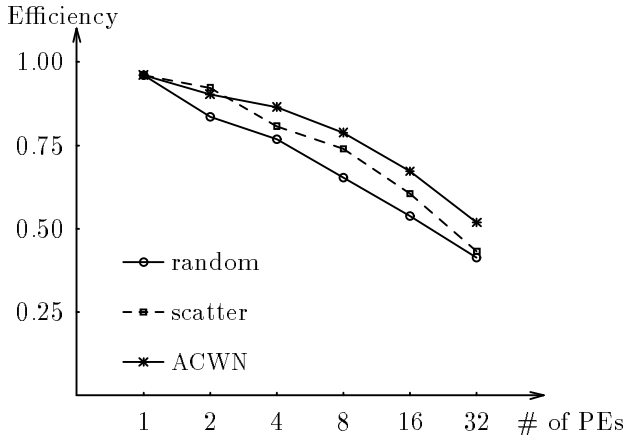

Figure 8: The efficiency comparison for Romberg Integration.