ELSEVIER

The Journal of Systems and Software 44 (1998) 17–29

The Journal of Systems and Software

Classifying and alleviating the communication overheads in matrix computations on large-scale NUMA multiprocessors [1]

Yi-Min Wang [a,*], Hsiao-Hsi Wang [a], Ruei-Chuan Chang [b,c]

[a] *Department of Computer Science and Information Management, Providence University, Taichung, Shalu, Taiwan, ROC*
[b] *Institute of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, ROC*
[c] *Institute of Information Science, Academia Sinica, Nankang, Taipei, Taiwan, ROC*

Received 16 February 1997; received in revised form 25 May 1997; accepted 17 June 1997

Abstract

Large-scale, shared-memory multiprocessors have non-uniform memory access (NUMA) costs. The high communication cost dominates the source of matrix computations' execution. Memory contention and remote memory access are two major communication overheads on large-scale NUMA multiprocessors. However, previous experiments and discussions focus either on reducing the number of remote memory accesses or on alleviating memory contention overhead. In this paper, we propose a simple but effective processor allocation policy, called rectangular processor allocation, to alleviate both overheads at the same time. The policy divides the matrix elements into a certain number of rectangular blocks, and assigns each processor to compute the results of one rectangular block. This methodology may reduce a lot of unnecessary memory accesses to the memory modules. After running many matrix computations under a realistic memory system simulator, we confirmed that at least one-fourth of the communication overhead may be reduced. Therefore, we conclude that rectangular processor allocation policy performs better than other popular policies, and that the combination of rectangular processor allocation policy with software interleaving data allocation policy is a better choice to alleviate communication overhead. © 1998 Elsevier Science Inc. All rights reserved.

1. Introduction

Parallel matrix computations are often used to solve linear algebra, numeric analysis, and graph problems on shared-memory multiprocessors, and the execution of these computations is one of the best ways to evaluate the performance of multiprocessors (Markatos and LeBlanc, 1994; Polychronopoulos and Kuck, 1987; Tzen and Ni, 1993; Wang et al., 1997). Under traditional multiprocessors such as BBN Butterfly I and Sequent Balance 8000, because the cost of memory access is lower than that of computation, the communication effect can be ignored. Therefore, in executing the computations, the main considerations are load balancing and synchronization overhead (Markatos and LeBlanc, 1992). Load balancing means that the workload of every available processor must be as even as possible, without leaving any processor idle. Synchronization operation overhead results from the processors' simultaneous accesses to a set of shared variables which contain the indices of iterations (Tzen and Ni, 1993; Wang and Chang, 1995).

Modern shared-memory multiprocessors have relatively high and non-uniform memory access (NUMA) costs, and the new architecture is an important trend in the development of high performance computer. Much effort has been done to develop large-scale shared-memory NUMA architectures both in academy and in industrial research departments. Toronto HECTOR (Vranesic et al., 1991) and NUMAchine (Vranesic et al., 1995), MIT Alewife (Agarwal et al., 1995), and Stanford Dash (Lenoski et al., 1992) are some examples. The high communication cost dominates the source of parallel matrix computations' execution (Markatos and LeBlanc, 1992, 1994; Wang and Chang, 1995). The high communication cost results in two main overheads, remote memory access and memory contention, and they reduce the performance of parallel applications on large-scale shared-memory NUMA multiprocessors.

The most important feature of shared-memory NUMA machines is that the memory modules, which

[*] Corresponding author. Tel.: +886 4 632 8001x3412; fax: +886 4 632 4045; e-mail: ymwang@simon.pu.edu.tw.

[1] This research work was partially supported by the National Science Council of Republic of China under grant No. NSC86-2621-E126-001-T and NSC87-2213-E126-013.

share one addressing space, are distributed to all nodes. In the system, a memory reference request may be in the cache, in the local memory module, or in the remote memory module. Many compiler loop scheduling techniques such as affinity scheduling algorithm (AFS) and modified AFS (MAFS) have been proposed to reduce the number of remote memory accesses without incurring load imbalance and synchronization overhead (Markatos and LeBlanc, 1994; Wang and Chang, 1995). The goal of these algorithms is to alleviate remote memory access overhead.

Memory contention occurs when multiple processors access the same memory module simultaneously (Harzallah and Sevcik, 1993), especially when all processors must access the same data just modified by a single processor (Bianchini et al., 1994). Memory contention increases the cost of individual memory access. Bianchini et al. (1994) proposed one particular software solution called software interleaving for data allocation, and they confirmed their idea by running some matrix computations on the simulator.

The experiments and discussions described above focus either on reducing the number of remote memory accesses or on alleviating memory contention overhead. However, both overheads always occur together and they affect each other in large-scale NUMA multiprocessors. In this paper, we propose an effective processor allocation methodology, called rectangular processor allocation policy, to alleviate both communication overheads simultaneously. The methodology may achieve good performance for some matrix computations under many popular data allocation policies in large-scale NUMA multiprocessors.

Our motivation is that under matrix computations, if we do not pay attention to processor allocation policies, a processor may access a large area of widespread matrix elements just for computing the results of a rather small number of matrix elements. If an application is run with a fixed number of processors under different processor allocation policies, the total number of memory reference requests for each processor may be about the same. However, these memory reference requests may be in the cache, in the local memory module, or in the remote memory module. Our processor allocation policy may reduce the number of matrix elements needed by each processor because we may increase the number of elements which are repeatedly referenced by the same processor. In this way, the possibility of memory requests in the cache is also increased.

Rectangular processor allocation policy divides the matrix elements into a certain number of rectangular blocks, and assigns each processor to compute the results of one rectangular block. As we know, row major allocation policy, the most popular processor allocation policy, assigns a single processor to compute the results of several consecutive entire rows of matrix. But rectangular processor allocation policy assigns one processor to compute the results of a rectangular block of matrix elements. Compared with row major processor allocation policy, rectangular allocation may reduce the total number of matrix elements needed by each processor and increase the cache hit ratios. High cache hit ratios may reduce the memory references to the memory modules, so the number of remote memory accesses and memory contention overhead will be reduced.

To simulate a NUMA environment, we use an on-line, execution-driven simulator to simulate a large-scale NUMA multiprocessor with up to 128 nodes. The simulator consists of two parts: Mint (Veenstra and Fowler, 1994) and a NUMA machine memory system simulator. Mint calls the memory system simulator on each memory reference, and the memory simulator decides whether the reference is in the cache, in the local memory module, or in the remote memory module. To simulate a NUMA environment more realistically and to capture the communication overheads correctly, we need a realistic memory system simulator. So we modify and enhance the simple cache simulator provided by Mint (Veenstra and Fowler, 1994). The simple cache simulator is a demonstration of user-provided system simulator, but the cache size of the simple cache simulator is infinite, and the coherent protocol is only bus-based. In our modified memory system simulator, each node has a single processor and a finite-size cache which uses directory-based and write-invalidate protocol. The simulator also considers the latency of memory contention and the delays of local and remote memory accesses.

We use all-pairs shortest paths, transitive closure, and Gaussian elimination as matrix applications. First of all, we classify the sources of communication overhead for various data allocation policies under row major processor allocation policy. Then, our simulating results show that under some popular data allocation policies, at least one-fourth of the communication overheads are removed when rectangular processor allocation policy is applied. Therefore, we conclude that under rectangular processor allocation policy, we may achieve better performance, and that the combination of rectangular processor allocation with software interleaving data allocation policy is a better way to alleviate communication overhead.

The rest of this paper is organized as follows: In Section 2, we analyze the effects of both overheads with many matrix computations. Because they often occur together, they must be considered at the same time. In Section 3, rectangular processor allocation policy is illustrated and some simulation results are presented. At last, the conclusion of the paper is given in Section 4.

2. The sources of communication overhead

Remote memory access and memory contention are two major sources of communication overhead on large-scale NUMA machines. In this section, we will show that they substantially reduce the performance of matrix computations on large-scale NUMA multiprocessors, and that because they often occur together, we must consider them at the same time. First we introduce the experimental environments and the benchmarks we use. And then we analyze the effects of both overheads by running various matrix applications under many popular data allocation policies.

Our study is to explore the communication overheads under various parameters, so simulation is appropriate for our experiments (Veenstra and Fowler, 1994). As is described in the previous section, we use a memory system simulator to simulate the shared-memory references realistically. In the experiment, we simulate a large-scale shared-memory NUMA multiprocessors with up to 128 nodes. Each node has a 64 k bytes four way associative cache with 32-byte cache line, and 16 M bytes local memory. The choice of four way associative cache may reduce conflict cache misses (Hennessy and Patterson, 1990). We use 32-byte cache line throughout the experiment; the reason is that large cache line may cause too much false sharing, and that small cache line may increase the number of network transactions required to load data into the cache. For most benchmarks, we may achieve good performance when 32-byte cache line is used (Dubnicki, 1993). We assume that it takes 10 cycles to access the local memory (Hennessy and Patterson, 1990), and assume that one memory module can only process one request at a time. Therefore, if a request arrives when the module is busy, it will be rejected and must be reissued. As to the network latency, we have two assumptions. First we assume there to be 36 cycles of network latency, and this assumption is similar to that made by Bianchini et al. (1994). If a memory access is in the local memory, it takes 10 cycles to complete its work. If a memory access is in the remote memory, it takes 82 cycles to complete its work, but in case the access is rejected, 72 cycles will be wasted. The ratio of remote to local memory access is about 8, and this ratio is conservative for most modern NUMA machines. Secondly, we assume there to be 72 cycles of network latency. In this case a remote access takes 154 cycles, and 144 cycles are wasted if the access is rejected. The ratio of remote to local memory access is about 15 – a more general assumption.

Our matrix applications consist of the following three parallel programs: all-pairs shortest paths, transitive closure, and Gaussian elimination.

The first problem is to compute the all-pairs shortest paths of a graph with 512 vertices, and the graph is represented by a 512×512 matrix A. $A[i][j]$ is the length of the shortest path from vertex i to vertex j for all $0 \leqslant i < 512$ and $0 \leqslant j < 512$. The pseudo code of this problem is shown as follows:

```
for ($k=0$; $k < 512$; $k$++)
    parallel for ($i=0$; $i < 512$, $i$++)
        for ($j=0$; $j < 512$; $j$++)
            $A[i][j] = \min\{A[i][j], A[i][k] + A[k][j]\}$
```

Each element in the matrix occupies 2 bytes, so each cache line may hold 16 elements. It takes 512 phases to complete the work, and we use barrier synchronization among different phases. The matrix is statically partitioned into several parts, and each part is assigned to one processor. Here we use static scheduling algorithm and the most popular processor allocation policy – row major allocation policy. In this policy each processor computes the results of a fixed number of entire rows. For example, as the number of processors is 64, each processor computes the results of 8 entire consecutive rows.

The second problem is to compute the transitive closure of a graph with 512 vertices. The graph is represented by a 512×512 matrix A. For all $0 \leqslant i < 512$ and $0 \leqslant j < 512$, $A[i][j]$ is 1 if there exists a path from vertex i to vertex j, otherwise $A(i,j)$ is 0, and the possibilities of both cases are equal. The pseudo code to solve the problem is shown as follows:

```
for ($k=0$; $k < 512$; $k$++)
    parallel for ($i=0$; $i < 512$, $i$++)
        for ($j=0$; $j < 512$; $j$++)
            if $A[i][k] = A[k][j] =' 1'$ then $A[i][j] =' 1'$
```

Each element in the matrix occupies 1 byte, so each cache line can hold 32 elements. It also takes 512 phases to complete the work, and we use barrier synchronization among different phases. The processor allocation policy is the same as the first problem, so we do not state more details.

The third problem is to perform Gaussian elimination of a 512×512 matrix A. The algorithm can be stated as follows:

```
for ($j=0$; $j < 512$; $j$++)
    parallel for ($i=j+1$; $i < 512$, $i$++)
        {
        $tmp = A[i][j]/A[j][j]$
        for ($k=j$; $k < 512$; $k$++)
            $A[i][k] = A[i][k] - tmp * A[j][k]$
        }
```

Each element in the matrix occupies 4 bytes, so each cache line can only hold 8 elements. It also takes 512 phases to complete the work, and we use barrier synchronization among different phases. Again the processor allocation policy is the same as the first problem, so we do not state more details.

These problems share the same characteristics: The first one is that the iterations in the parallel loop are all independent. The second characteristic is that during the ith phase, all processors compete for the ith memory module, and most of the memory accesses are remote ones.

We simulate three types of data allocation policies – row major, column major, and software interleaving. Row major *(Row)* assigns one entire row to the memory module of a processor. Column major *(Column)*, on the other hand, assigns each element of a row to a different memory module. Software interleaving, proposed by Bianchini et al. (1994), divides each row of the matrix into several cache blocks, and maps the cache blocks of a single row into different memory modules.

We use E_{free} as the ideal execution time of matrix computation. In this case, we assume that all cache-miss memory requests are in the local memory module and that no memory contention occurs. Then the performance reduction, resulting from remote memory access, is defined as follows:

$$(E_{\text{remote}} - E_{\text{free}})/E_{\text{free}} \tag{1}$$

E_{remote} is the execution time when no memory contention occurs but remote memory access delay is considered in the simulation. The performance reduction, resulting from memory contention, is defined as follows:

$$(E_{\text{real}} - E_{\text{remote}})/E_{\text{free}} \tag{2}$$

E_{real} is the real execution time when both remote memory access delay and memory contention overhead are considered.

First we analyze the overhead for all-pairs shortest paths problem when row major processor allocation policy is applied under various data allocation policies. We use two ratios (8 and 15) of remote to local memory access. Figs. 1 and 2 show the performance reduction resulting from memory contention and remote memory access with 64 and 128 processors.

Among these data allocation policies, row major suffers the highest overhead because of memory contention. The reason is that during each phase, all processors must compete for the data of the same row. As the number of processors and the ratio of remote to local memory access increase, the overhead of memory contention is increased. For example, when the number of processors is 128 and the ratio of remote to local memory access is 15, it suffers 0.48 performance reduction resulting from memory contention.

On the other hand, column major policy suffers most overhead because of remote memory access. This policy reduces the memory contention, but causes higher cache miss ratio resulting from the false sharing of cache lines. Thus the number of accesses to memory modules is increased and this leads to higher possibility of remote memory access. The overhead increases as the ratio of remote to local memory reference gets larger. Software interleaving allocation performs better than the other two policies; however, the overhead is still large.

In the same way we may analyze the communication costs under transitive closure and Gaussian elimination. Figs. 3 and 4 present the overheads of transitive closure. Figs. 5 and 6 present the overheads of Gaussian elimination. Transitive closure performs similarly to all-pairs shortest paths problem, but the reduction of performance is not so significant and the differences between various data allocation policies are less obvious. The reason is that each cache line contains more elements in transitive closure than in all-pairs shortest paths. So in transitive closure, the cache miss ratios are smaller. Thus the memory access traffic of transitive closure is lighter than that of all-pairs shortest paths. Again row major policy suffers more memory contention and column major policy suffers more remote memory access.

On the other hand, the performance reduction under Guassian elimination is much larger than under the other two problems. Even when the ratios of remote to local memory access is only 8 and the number of processors is 64, the performance reductions are high. The reason is that in this application, the cache line holds less data and the data updating frequency is too high. So the cache miss ratios are much larger than those of the other two problems, and the memory traffic is very busy.

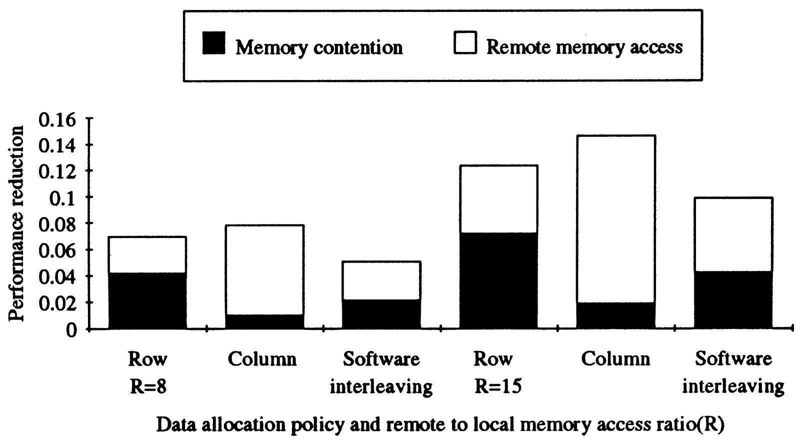

Fig. 1. The sources of overhead under all-pairs shortest paths with 64 processors.

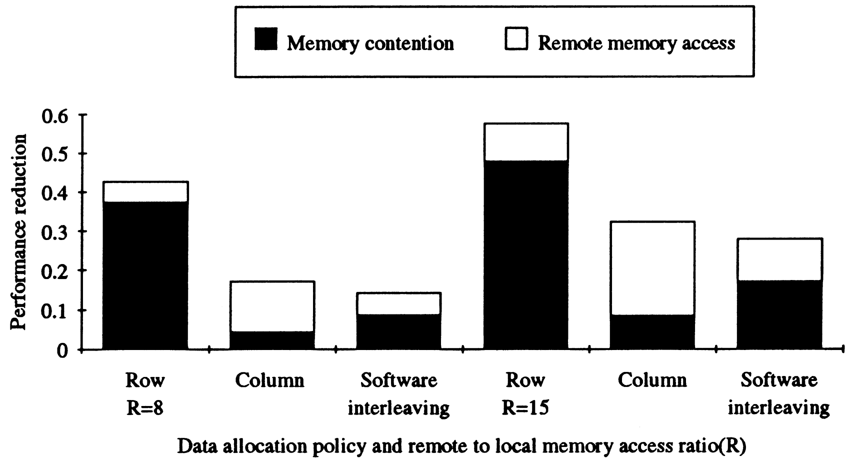

Fig. 2. The sources of overhead under all-pairs shortest paths with 128 processors.

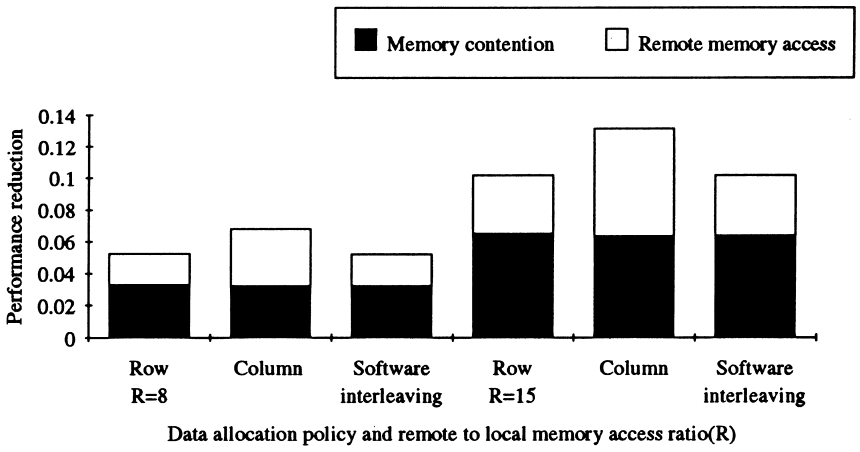

Fig. 3. The sources of overhead under transitive closure with 64 processors.

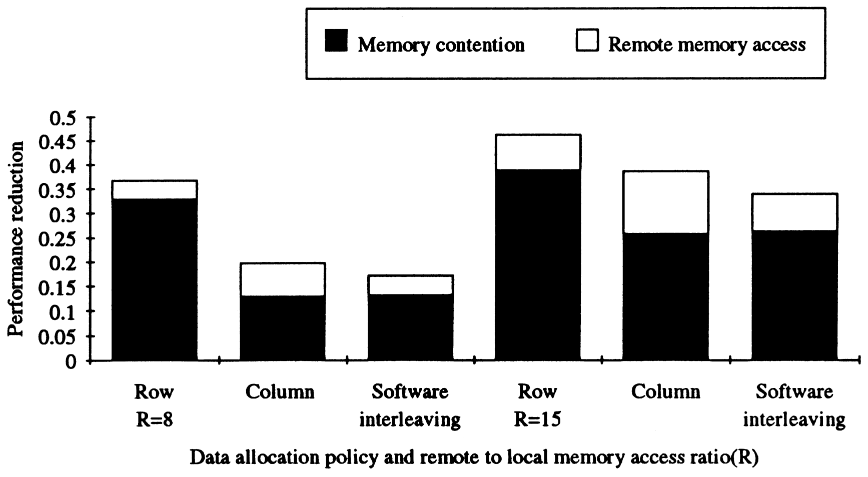

Fig. 4. The sources of overhead under transitive closure with 128 processors.

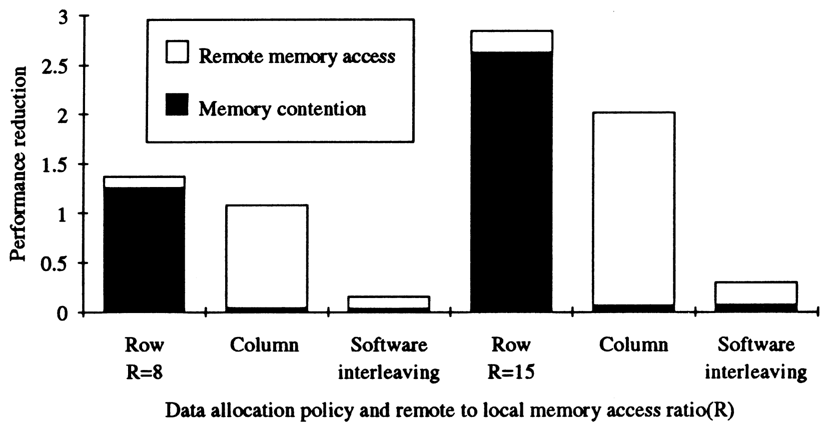

Fig. 5. The sources of overhead under Gaussian elimination with 64 processors.

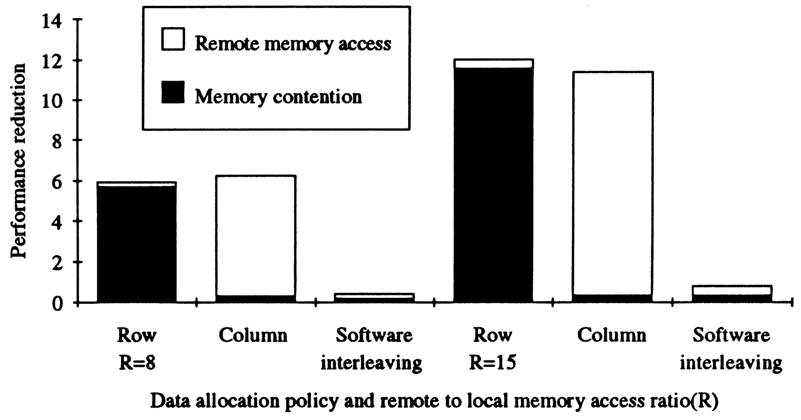

Fig. 6. The sources of overhead under Gaussian elimination with 128 processors.

We may summarize that memory contention and remote memory reference not only occur together but also substantially reduce the performance of parallel matrix applications on NUMA multiprocessors. The effects may differ for various data allocation policies. In the next section, we will argue that the overheads may be alleviated by allocating processors to compute matrices elements in a simple but effective way.

3. Alleviating the communication overheads by rectangular processor allocation

In this section, we illustrate rectangular processor allocation policy in detail. Then we show the effect of this policy by running many matrix computations under various combinations of processor allocation policies with data allocation policies.

First, we give a simple example to illustrate the importance of processor allocation assignment. In many matrix computations, to compute the result of an element of a matrix, it is possible to reference many other rows or/and columns. Take the computation of the all-pairs shortest paths of a matrix A as an example. To compute the result of A_{11}, the entire 1st row and the entire 1st column of matrix A must be referenced, where A_{kj} is the element in row k and column j of matrix A. Now we consider the following two processor allocation policies for the computation of A:

1. Processor i computes the results of the ith row of matrix A.
2. Processor i computes the results of the left half of both ith and the $(i+1)$th rows of matrix A, and processor $(i+1)$ computes the right half of the two rows, where i is an odd number.

Fig. 7 shows the areas of data elements to be accessed and the areas to be computed for processor i under both policies. It shows that in order to compute the results of the same-sized data elements, in the first policy, we must reference the whole matrix of A. But in the second policy, we only need to reference about half. The reason is that the data elements' reusability in the second policy is larger than that in the first one.

We use an effective processor allocation policy, called *rectangular processor allocation* policy, to alleviate the communication bottlenecks for many matrix applica-

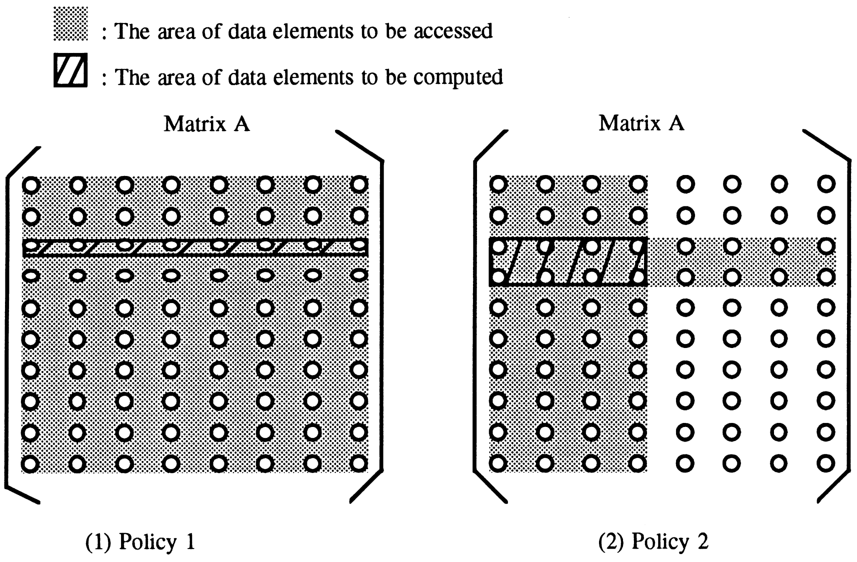
▒ : The area of data elements to be accessed
▨ : The area of data elements to be computed
Matrix A
Matrix A
(1) Policy 1
(2) Policy 2

Fig. 7. The data elements to be computed and the areas to be accessed for two possible processor allocation policies under all-pairs shortest paths.

tions. Rectangular processor allocation policy divides matrix into a certain number of same-sized rectangular blocks, and each processor computes the result of one block. The idea of dividing matrix into a certain number of rectangular blocks is similar to the software interleaving (Bianchini et al., 1994), used in data allocation and is also similar to the tiling technique, used in loop and data transformation (Ramanujam and Sadayappan, 1990). But we use this technique in the assignment of processor allocation. As we know, row major processor allocation policy, the most popular processor allocation policy, assigns several consecutive entire rows to each processor, and each processor computes the results of these rows. But rectangular processor allocation policy assigns each processor to compute the results of a rectangular block of elements. The width (the number of elements) of a rectangular block is the multiple of a cache line size. Fig. 8 shows the areas of data elements to be accessed when we compute the same-sized data under both processor allocation policies for all-pairs shortest paths and transitive closure problems. We may give an investigation of the areas of data elements to be accessed for both policies under these two problems:

If the rank of the matrix is N, and there are P available processors:

1. Under row major allocation policy, for each processor, the total size of elements to be accessed is $N \times N$.

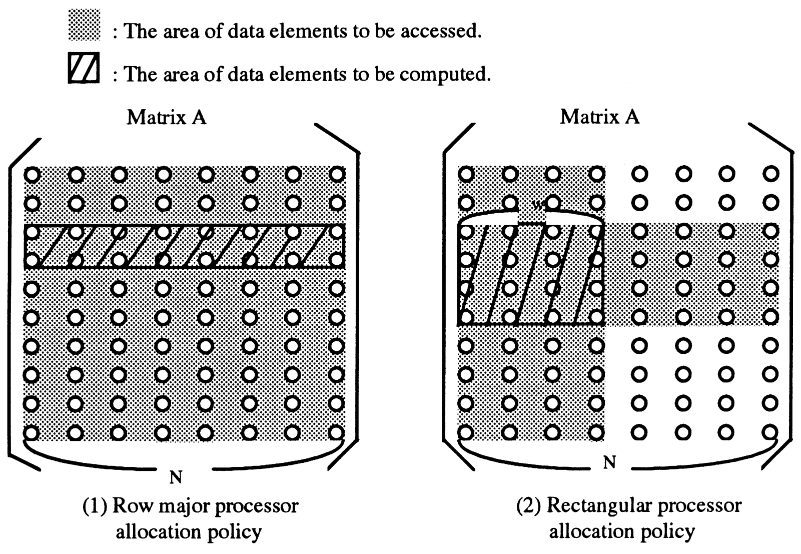
▒ : The area of data elements to be accessed.
▨ : The area of data elements to be computed.
Matrix A
Matrix A
w
N
N
(1) Row major processor allocation policy
(2) Rectangular processor allocation policy

Fig. 8. The data elements to be computed and the areas to be accessed for row major and rectangular allocation policies under all-pairs shortest paths and transitive closure.

2. If the width (the number of elements) of the rectangular block to be computed is w and w is the multiple of a cache line size, then the total size of the data elements to be accessed for each processor is $N \times N \times (p1 + p2 - 1)/P$ under rectangular processor allocation policy, where $p1 = N/w$, and $p2 = P/p1$. Obviously, as $p1 = p2$, the data elements to be accessed is minimized, $N \times N \times (2 \times \sqrt{P} - 1)/P$.

As for Gaussian elimination, the distributions of the data elements to be accessed for both policies are a little different from those of transitive closure and all-pairs shortest paths problems. Nevertheless, the total areas of data elements to be accessed under rectangular processor allocation policy are quite smaller than those under row major processor allocation policy. Fig. 9 shows an example of the data elements to be accessed under both processor allocation policies with same-sized elements to be computed.

To compare the performance of both processor allocation policies, the matrix applications are run for both processor allocation policies under the same simulating environments as in the second section. Table 1 shows the execution times under both processor allocation policies with 64 and 128 processors for all-pairs shortest paths problem. It shows that when rectangular processor allocation policy is used, the execution times in this policy are shorter than those in row major processor allocation policy under any data allocation policy.

To confirm the result of Table 1, we collect the cache miss ratios and analyze the sources of communication overheads for both processor allocation policies. Table 2 shows the cache miss ratios for both processor allocation policies under various data allocation policies. It shows that under rectangular processor allocation policy, the miss ratios are only about one-fourth of those under row major processor allocation policy. Figs. 10 and 11 show the overheads resulting from memory contention and remote memory reference with 64 and 128 processors under various ratios of remote to local memory reference. They show that under any data allocation policy, rectangular processor allocation policy always performs better than row major processor allocation policy. In most cases, it reduces about a half overheads of row major processor allocation policy. Among those combinations, software interleaving data allocation policy and rectangular processor allocation

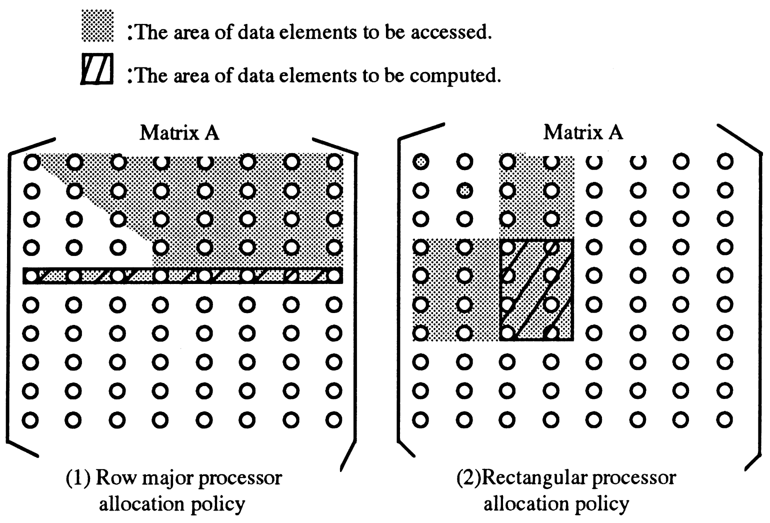
:The area of data elements to be accessed.
:The area of data elements to be computed.
Matrix A
Matrix A
(1) Row major processor allocation policy
(2)Rectangular processor allocation policy

Fig. 9. The data elements to be computed and the areas to be accessed for row major and rectangular allocation policies under Gaussian elimination.

Table 1
The execution times (in millions of cpu cycles) for all-pairs shortest paths under row major and rectangular processor allocation policies

Processor allocation policy	Remote to local memory access ratio = 8 Data allocation policy			Remote to local memory access ratio = 15 Data allocation policy		
	Row major	Column major	Software interleaving	Row major	Column major	Software interleaving
64 processors						
Row major	106	102.3	104.2	111.4	108.8	108.9
Rectangular	103.6	99.3	102.3	107.3	103.1	104.6
128 processors						
Row major	73.9	58.2	59.2	81.7	65.8	66.3
Rectangular	58.3	56.4	56.1	63.9	62.3	59.0

Table 2
The cache miss ratios for all-pairs shortest paths under row major and rectangular processor allocation policies

Processor allocation policy	Data allocation policy		
	Row major (%)	Column major (%)	Software interleaving (%)
64 processors			
Row major	0.26	0.58	0.26
Rectangular	0.08	0.09	0.08
128 processors			
Row major	0.52	1.19	0.52
Rectangular	0.14	0.12	0.13

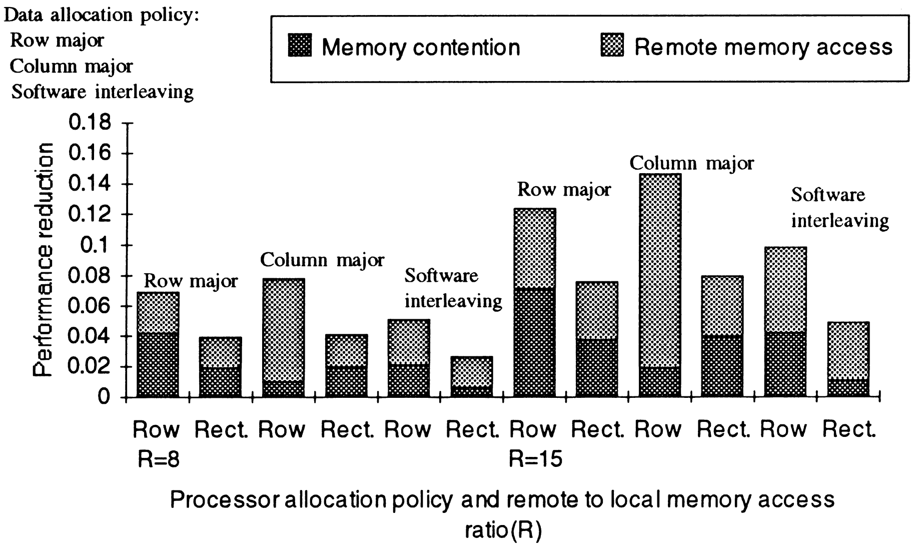

Fig. 10. The sources of overhead for all-pairs shortest paths under row major (Row) and rectangular (Rect.) processor allocation policies with 64 processors.

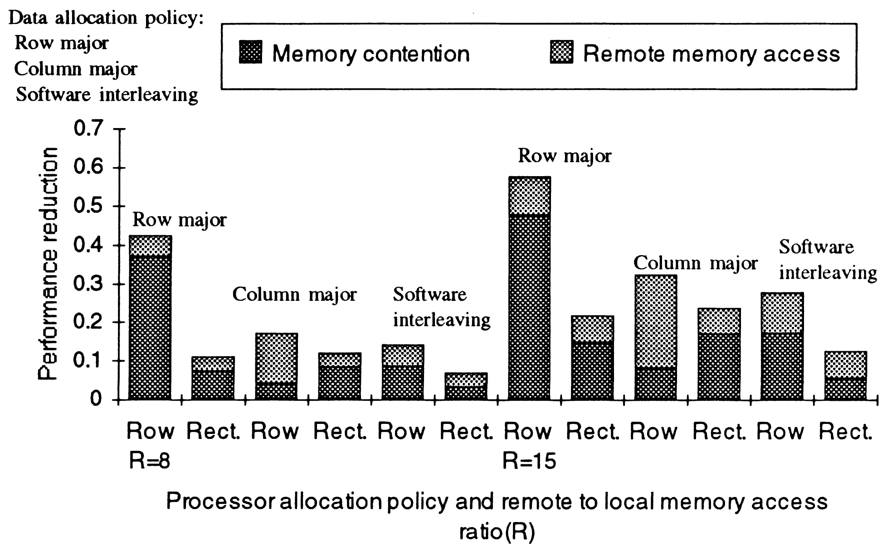

Fig. 11. The sources of overhead for all-pairs shortest paths under row major (Row) and rectangular (Rect.) processor allocation policies with 128 processors.

Table 3
The execution times (in millions of cpu cycles) for transitive closure under row major and rectangular processor allocation policies

Processor allocation policy	Remote to local memory access ratio = 8 Data allocation policy			Remote to local memory access ratio = 15 Data allocation policy		
	Row major	Column major	Software interleaving	Row major	Column major	Software interleaving
64 processors						
Row major	74.4	77.8	74.4	77.9	82.4	77.9
Rectangular	74.2	76.5	73.9	77.0	79.3	76.3
128 processors						
Row major	47.9	43.2	41.1	51.2	50.0	46.9
Rectangular	40.5	41.5	40.5	45.6	46.4	45.6

policy together reduce the largest amount of communication overhead. There is about 0.01 overhead of performance reduction resulting from the extra loop control under rectangular processor allocation policy, and the effect is slight.

In the same way, we may experiment with the results of transitive closure and Gaussian elimination. Tables 3 and 4 show the execution times for these two problems under both processor allocation policies. Once again, we collect the cache miss ratios and analyze the sources of communication cost for transitive closure and Gaussian elimination. The results of cache miss ratios are shown in Tables 5 and 6, and the overheads are shown in Figs. 12–15. Again in the case of transitive closure problem, rectangular processor allocation policy performs better than row major processor allocation policy as in the case of all-pairs shortest paths problem. The rectangular policy reduces about one-fourth to half the overheads compared with row major policy, and the combination of software interleaving data allocation policy and rectangular processor allocation policy remove more overhead than the others. As for Gaussian elimination problem, at least half overheads are reduced under rectangular processor allocation policy, and the combination of software interleaving data allocation policy with rectangular processor allocation policy remove the largest amount of overhead. But in some cases, the execution times of rectangular processor allocation policy are longer than those of row major processor allocation policy. The reason is that Gaussian elimination is an example of load imbalance but the other two problems are examples of load balance. Compared with row major processor allocation policy, rectangular processor allocation policy suffers more performance reduction because of load imbalance. Combined with more complicated dynamic scheduling

Table 4
The execution times (in millions of cpu cycles) for Gaussian elimination under row major and rectangular processor allocation policies

Processor allocation policy	Remote to local memory access ratio = 8 Data allocation policy			Remote to local memory access ratio = 15 Data allocation policy		
	Row major	Column major	Software interleaving	Row major	Column major	Software interleaving
64 processors						
Row major	82.0	78.3	39.3	133.4	114.0	44.2
Rectangular	71.8	88.9	57.3	90.2	116.6	61.5
128 processors						
Row major	114.0	130.7	22.3	215.0	224.2	28.7
Rectangular	52.3	53.3	34.8	74.2	71.9	39.6

Table 5
The cache miss ratios for transitive closure under row major and rectangular processor allocation policies

Processor allocation policy	Data allocation policy		
	Row major (%)	Column major (%)	Software interleaving (%)
64 processors			
Row major	0.20	0.38	0.20
Rectangular	0.05	0.07	0.05
128 processors			
Row major	0.39	0.64	0.39
Rectangular	0.07	0.11	0.07

Table 6
The cache miss ratios for Gaussian elimination under row major and rectangular processor allocation policies

Processor allocation policy	Data allocation policy		
	Row major (%)	Column major (%)	Software interleaving (%)
64 processors			
Row major	0.60	16.97	0.60
Rectangular	0.40	7.96	0.40
128 processors			
Row major	1.16	81.87	1.16
Rectangular	0.55	8.16	0.55

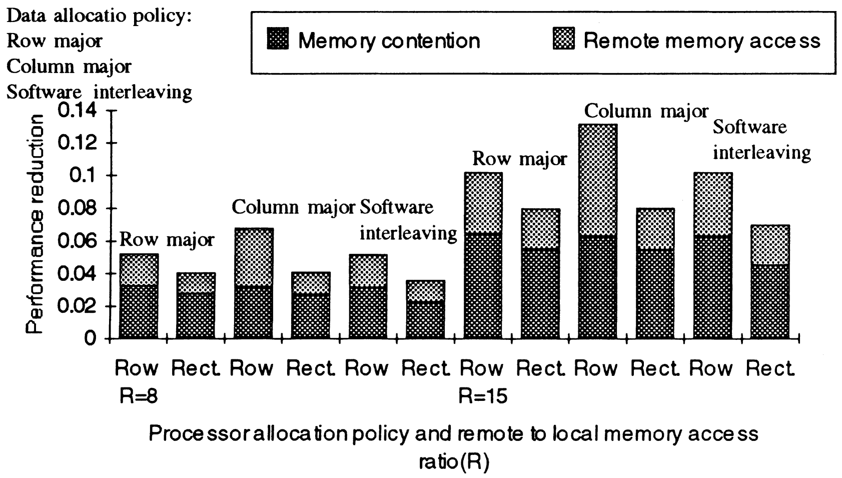

Fig. 12. The sources of overhead for transitive closure under row major (Row) and rectangular (Rect.) processor allocation policies with 64 processors.

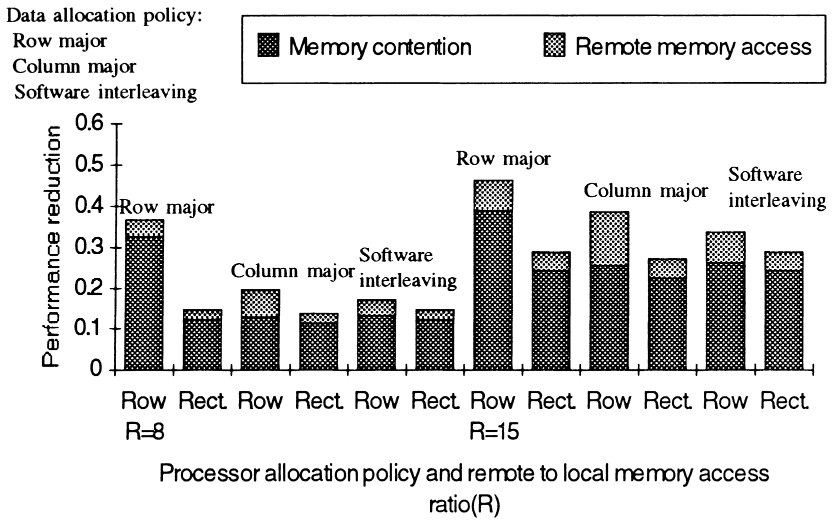

Fig. 13. The sources of overhead for transitive closure under row major (Row) and rectangular (Rect.) processor allocation policies with 128 processors.

algorithms, such as AFS (Markatos and LeBlanc, 1994), LDS (Li et al., 1993), and MAFS (Wang and Chang, 1995), we may alleviate load imbalance to improve the performance.

We may summarize that by using simple but effective processor allocation policy, we may increase the cache hit ratios. So both memory contention and remote memory reference overheads will be alleviated. And the

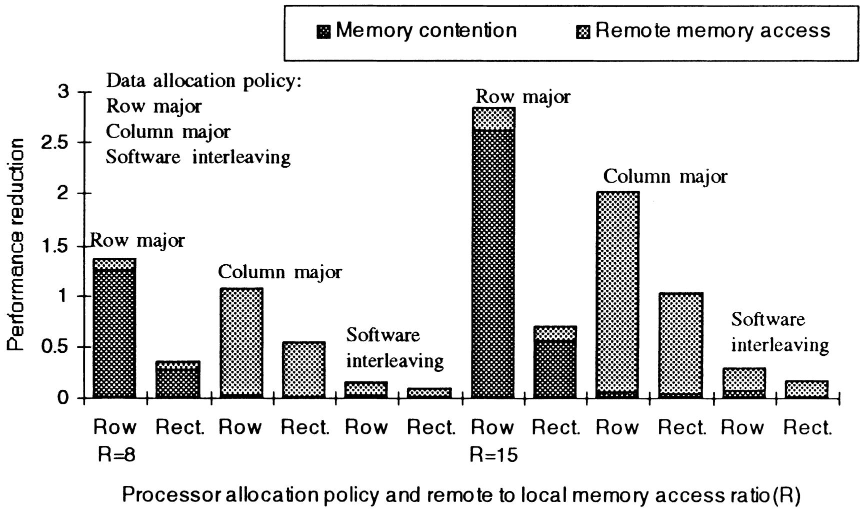

Fig. 14. The sources of overhead for Gaussian elimination under row major (Row) and rectangular (Rect.) processor allocation policies with 64 processors.

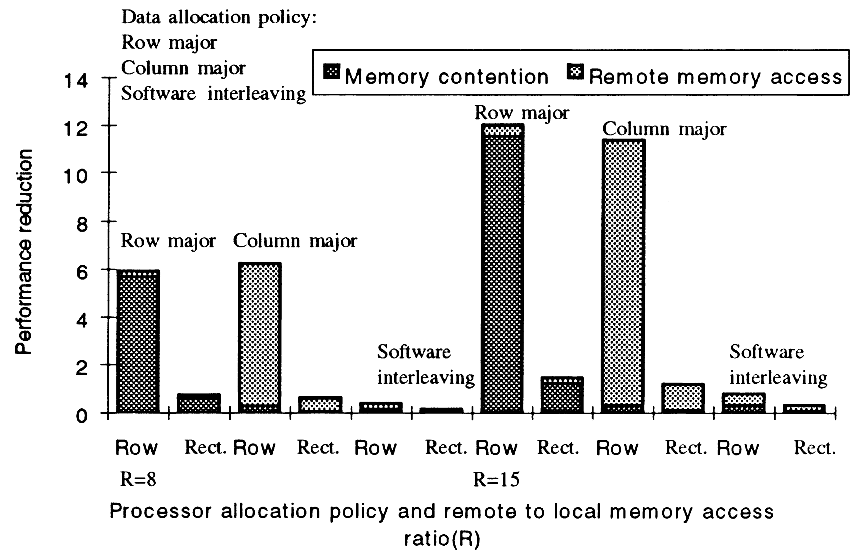

Fig. 15. The sources of overhead for Gaussian elimination under row major (Row) and rectangular (Rect.) processor allocation policies with 128 processors.

combination of rectangular processor allocation policy with software interleaving data allocation policy may reduce the largest amount of overhead. Thus the performance of multiprocessor will be improved for many matrix computations.

4. Conclusion

As we know, modern large-scale, shared-memory multiprocessors have non-uniform memory access costs. The ratio of memory access cost to computation cost has increased significantly during these years. So the communication cost has become the third dimension and gradually dominates the source of parallel matrix applications' execution. Communication cost results in two main overheads, remote memory access and memory contention, and they reduce the performance of matrix computations on NUMA multiprocessors. Because these two overheads always occur together, neither of them can be ignored in the development of parallel environments.

In this paper, we propose a simple but effective processor allocation policy to alleviate both communication overheads at the same time. This methodology may reduce a lot of unnecessary accesses to local or remote modules. By running many matrix operations under a realistic memory system simulator, we confirm that at least one-quarter of the communication overhead may be removed when rectangular processor allocation policy is applied. Therefore, we conclude that under rectangular processor allocation policy, we may achieve better performance under any data allocation policy, and that the combination of rectangular processor allocation with software interleaving data allocation policy is a better choice to alleviate communication overhead.

Since the processor allocation policy plays an important role in the execution of NUMA multiprocessor, it is an interesting topic to combine more complicated loop scheduling algorithms with processor allocation policies so as to improve the performance of NUMA environment.

References

Agarwal, A., Bianchini, R., Chaiken, D., Johnson, K.L., Kranz, K., Kubiatowicz, J., Lim, B.H., Mackenzie, K., Yeung, D., 1995. The MIT Alewife machine: Architecture and performance. Proceedings of the 22nd International Symposium on Computer Architecture. pp. 2–13.

Bianchini, R., Crovella, M.E., Kontothanassis, L., LeBlanc, T.J., 1994. Software interleaving. Proceedings of the 1994 Symposium on Parallel and Distributed Processing, pp. 56–65.

Dubnicki, C., 1993. The effects of block size on the performance of coherent caches in shared-memory multiprocessors. Ph.D. Thesis, University of Rochester, Computer Science Department.

Harzallah, K., Sevcik, K.C., 1993. Hot spot analysis in large scale shared memory multiprocessors. Proceedings of the Supercomputing '93 Conference, pp. 895–905.

Hennessy, J.L., Patterson, D.A., 1990. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, Los Altos, CA.

Lenoski, D., Laudon, J., Joe, T., Nakahira, D., Stevens, L., Gupta, A., and Hennessy, J., 1992. The Dash prototype: Implementation and performance. The 19th Annual International Symposium on Computer Architecture. pp. 92–103.

Li, H., Tandri, S., Stumn, M., Sevcik, K.C., 1993. Locality and loop scheduling on NUMA multiprocessors. International Conference on Parallel Processing. pp. 140–147.

Markatos, E.P., LeBlanc, T.J., 1992. Shared-memory multiprocessors trends and the implications for parallel program performance. Technical Report 420. Computer Science Department, University of Rochester.

Markatos, E.P., LeBlanc, T.J., 1994. Using processor affinity in loop scheduling on shared-memory multiprocessors. IEEE Trans. on Parallel and Distributed Systems 5 (4), 379–400.

Polychronopoulos, C.D., Kuck, D.J., 1987. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. IEEE Trans. on Computers C/36 (12), 1425–1439.

Ramanujam, J., Sadayappan, P., 1990. Tiling of iteration spaces for multiprocessors. International Conference on Parallel Processing. pp. 178–186.

Tzen, T.H., Ni, L.M., 1993. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. IEEE Trans. on Parallel and Distributed Systems 4 (1), 87–98.

Veenstra, J.E., Fowler, R.J., 1994. MINT Tutorial and User Manual. Technical Report 452. Computer Science Department, University of Rochester.

Vranesic, Z.G., Stumm, M., Lewis, D.M., White, R., 1991. Hector: A hierarchically structured shared-memory multiprocessor. IEEE Computer 24 (1), 72–80.

Vranesic, Z., Brown, S., Stumm, S., Caranci, S., Grbic, A., Grindley, R., Gusat, M., Krieger, O., Lemieux, G., Loveless, K., Manjikian, N., Zilic, Z., Abdelrahman, T., Gamsa, B., Pereira, P., Sevcik, K., Elkateeb, A., Srbljic, S., 1995. The NUMAchine Multiprocessor. Technical Report CSRI-324. Toronto University, Computer Systems Research Institute.

Wang, Y.M., Chang, R.C., 1995. A minimal synchronization overhead affinity scheduling algorithm for shared-memory multiprocessor. International Journal of High Speed Computing 7 (2), 231–249.

Wang, Y.M., Wang, H.H., Chang, R.C., 1997. Clustered affinity scheduling on large-scale NUMA multiprocessors. The Journal of Systems and Software 39 (1), 61–70.

YI-MIN WANG received the B.S. degree in Computer and Information Science from Chiao Tung University, Hsinchu, Taiwan, ROC in 1984, the M.S. degree in Computer and Decision Science from Tsing Hua University, Hsinchu, Taiwan, ROC in 1986, and the Ph.D. degree in Computer and Information Science from Chiao Tung University, Hsinchu, Taiwan, ROC in 1996. From 1986 to 1989 he was an engineer at Chung Shan Institute of Science and Technology, Taiwan, ROC, and from 1989 to 1992 he was an engineer at the Institute of Information Industry, Taipei, Twiwan, ROC. Now he is an Associate Professor of the Department of Computer Science and Information Management, Providence University, Taichung, Shalu,Taiwan, ROC. His research interests include operating systems, parallel processing, and computer architecture.

HSIAO-HIS WANG received the B.S. degree in Computer and Information Science, Ph.D. degree in Computer Science and Information Engineering from National Chiao Tung University, Hsinchu, Taiwan, ROC. Now he is an Associate Professor of the Department of Computer Science and Information Management, Providence University, Taichung, Shalu, Taiwan, ROC. The current research interests of Dr. Wang include operating systems, parallel processing, distribution systems, and algorithm design.

RUEI-CHUAN CHANG received the B.S. degree in 1979, the M.S. degree in 1981, and the Ph.D. degree in 1984, all in computer engineering from National Chiao Tung University, Hsinchu, Taiwan, ROC. In August 1983, he joined the department of Computer and Information Science at National Chiao Tung University as a Lecturer. Now he is a Professor of the Department of Computer and Information Science. He is also an Associate Research Fellow at the Institute of Information Science, Academia Sinica, Taipei, Taiwan, ROC. The current research interests of Dr. Chang include system softwares, distributed systems, design and analysis of algorithms, and computer graphics.