ELSEVIER

The Journal of Systems and Software 55 (2000) 33–44

The Journal of Systems and Software

www.elsevier.com/locate/jss

Hierarchical loop scheduling for clustered NUMA machines

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

[a] Department of Computer Science and Information Management, Providence University, Taichung, Shalu 433, Taiwan, ROC
[b] Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, ROC

Received 11 January 1999; received in revised form 18 May 1999; accepted 10 August 1999

Abstract

Loop scheduling is an important issue in the development of high performance multiprocessors. As modern multiprocessors have high and non-uniform memory access (NUMA) costs, the communication costs dominate the execution of parallel programs. Previous affinity algorithms perform better than dynamic algorithms under non-clustered NUMA multiprocessors, but they suffer heavy overheads when migrating work load under clustered NUMA machines. In this paper, we propose a new loop scheduling policy, hierarchical policy, to improve various affinity scheduling algorithms (AFSs) for clustered NUMA machines. We cyclically distribute the iteration chunks to clusters. When imbalance occurs, the migration of iterations is carried on hierarchically. We use hierarchical policy to improve AFS and modified AFS (MAFS), and we call them Hierarchical AFS (HAFS) and Hierarchical MAFS (HMAFS), respectively. AFS uses a deterministic assignment policy to assign repeated executions of loop iteration to the same processor. MAFS modifies the migration policy of AFS, and reduces the number of synchronization operations. We confirm our idea by running many applications under a clustered NUMA simulator. Our experimental result shows that hierarchical policy reduces the inter-cluster remote memory accesses, decreases the locks to the queues, and effectively balances the work load. We also show that HMAFS is the best choice among these algorithms in most cases.

1. Introduction

Clustered shared-memory multiprocessors are the trends of modern multiprocessors. As these machines have large numbers of processors, the system is often divided into clusters, each containing a small number of processors. These clusters are connected by hierarchical network to form the system. Since the memory modules are distributed across the system, yet share one addressing space, processors may access local or remote memory modules in the system. So these machines have non-uniform memory access (NUMA) costs. Moreover, as the accesses are to the inter-cluster memory, the costs are high. Toronto HECTOR (Vranesic et al., 1991; Stumm et al., 1992) and NUMAchine (Vranesic et al., 1995), MIT Alewife (Agarwal et al., 1995), and Stanford Dash (Lenoski et al., 1992) are examples of clustered NUMA machines.

On the other hand, loop scheduling is an important issue to design the system software for multiprocessor systems. Under uniform-memory access (UMA) multiprocessors, static and dynamic algorithms have been extensively studied. The main considerations are load balance and synchronization overhead (Polychronopoulos and Kuck, 1987; Tzen and Ni, 1993; Hummel et al., 1992; Kruskal and Weiss, 1985; Markatos and LeBlanc, 1992). After the evolution of multiprocessor architecture, the communication cost becomes the bottleneck and dominates the performance of parallel program executions in NUMA multiprocessors. To reduce the communication cost, some affinity scheduling algorithms (AFSs) have been proposed for NUMA multiprocessors. AFSs deterministically partition and schedule the loop iterations to the processors. The data for an iteration are placed on the cache of some dedicated processor to be used again and again. Markatos and LeBlanc derived the first one called AFS (Markatos and LeBlanc, 1994). Li et al. (1993) proposed LDS with data placement taken into consideration. Wang and Chang (1995) proposed modified AFS (MAFS) to combine the advantages of both AFS (Markatos and LeBlanc, 1994) and guided self-scheduling (GSS) (Polychronopoulos and Kuck, 1987). Subramaniam and Eager (1994) proposed dynamic partitioned affinity scheduling and wrapped partitioned affinity scheduling for iterations with widely varying execution times. Wang

* Corresponding author. Tel.: +886-4-632-8001x3412; fax: +886-4-632-4045.
E-mail address: ymwang@pu.edu.tw (Y.-M. Wang).

0164-1212/00/$ - see front matter
PII: S0164-1212(00)00045-5

et al. (1997) proposed CAFS for larger NUMA multiprocessors.

Although the above affinity algorithms perform well on UMA and non-clustered NUMA machines, these algorithms cannot be directly applied to clustered NUMA machines. As we will show, when load imbalance occurs, iteration migration and data move are needed. Affinity algorithms waste time in accessing inter-cluster memory. Furthermore, as the number of processors is increased, the synchronization overhead will be increased.

In this paper some well-known loop scheduling algorithms are experimented on clustered NUMA machines by simulation. We propose a simple affinity scheduling policy, hierarchical policy, to improve the performance of various affinity algorithms. Our policy may be easily applied to many AFSs. For example, it can be used to improve the performance of AFS and MAFS in most cases, as we will show. We call the new ones as Hierarchical AFS (HAFS) and Hierarchical MAFS (HMAFS). Our policy reduces the number of inter-cluster remote memory accesses, decreases the synchronous operations to the work queues, and effectively balances the work load.

The idea of our policy is that when imbalance occurs, the migration of iterations is carried on hierarchically. As an example, when executing parallel applications on Hector (Vranesic et al., 1991; Stumm et al., 1992) with 4 stations (each station contains 4 processors), the idle processor first migrates a fraction of iterations from the most loaded processor in its local-cluster (station), before searching other clusters. Only if all of the processors' work queues in the local cluster are empty, the idle processor does search and migrate from the other clusters. Since the idle processor first searches the work queues in its own cluster for the iteration indices, the number of remote accesses and that of synchronous operations are reduced. Moreover, because the idle processor migrates most of the work only from these processors in local-cluster, the inter-cluster memory accesses are reduced.

We use an on-line, execution-driven simulator to simulate a clustered NUMA multiprocessor with 16 clusters, and each cluster contains 4 nodes. The simulator consists of two parts: Mint (Veenstra and Fowler, 1994) and a clustered NUMA memory system simulator. Mint is a software package, top of which multiprocessor memory system simulators can be constructed. We modify and enhance the simple cache simulator provided by Mint, and make it a clusterd NUMA memory system simulator. Mint calls the memory system simulator on each memory reference, and the memory system simulator must decide the locations of the memory reference. Each node in our memory simulator has a processor and a finite-size cache. The caches use a write-invalidate protocol that is directory based. The simulator also takes into consideration the synchronization operations and the non-uniform remote memory access latency.

The applications we choose include Gaussian elimination, all-pairs shortest paths, adjoint convolution, reverse adjoint convolution, and two synthetic parallel programs. By running various applications on the simulator, we characterize the execution times, synchronization overhead, and inter-cluster memory accesses for various scheduling algorithms. We implement static, GSS, AFS, CAFS, MAFS, HAFS, and HMAFS on the simulator. Compared with AFS and MAFS, our policy may remove many synchronous operations to the work queues and reduce a lot of inter-cluster memory accesses in most cases. As a result, the hierarchical policy may shorten the execution times of these applications.

The organization of this paper is as follows: in Section 2, some popular loop scheduling algorithms are briefly described. Then hierarchical policy is described in Section 3. In Section 4, we describe our experimental environment. In Section 5, we show the results of simulations under various loop scheduling algorithms. Finally, the conclusion is given in Section 6.

2. Loop scheduling algorithms

Loops are the main source of parallelism in most programs (Subramaniam and Eager, 1994). To shorten the execution of programs on multiprocessors, the independent iterations may be executed in parallel on the multiprocessors. Loop scheduling algorithms partition and schedule the loop iterations on the processors in the hope that the multiprocessors will complete the work as soon as possible.

A static scheduling algorithm would assign a fixed number of iterations to each processor at compile time. It would divide the N iterations into P chunks, and assign $\lceil N/P \rceil$ consecutive iterations to each processor, where N is the number of total iterations, and P is the number of processors. This approach performs well if the work loads of all chunks are equal. However, in practice the work loads of iteration distributions may be non-uniformed. So static algorithms always achieve poor speed-up.

Dynamic scheduling algorithms, such as fixed-size chunking (Tzen and Ni, 1993), GSS (Polychronopoulos and Kuck, 1987), factoring scheduling (Hummel et al., 1992), and trapezoid self-scheduling (TSS) (Tzen and Ni, 1993), share the same characteristics. They always maintain a global queue containing indices of iterations. At run time, when a processor is idle, it issues synchronous operations to the global queue and fetches some iterations for execution. Dynamic scheduling usually provides an advantage in terms of load balance. However, it also requires more frequent exclusive

accesses to the global work queue. So dynamic algorithms result in more synchronization overhead. Among these dynamic algorithms, GSS (Polychronopoulos and Kuck, 1987) is the first algorithm to dynamically adjust the granularity of task at run time. Each idle processor fetches $\lceil 1/P \rceil$ of unscheduled iterations from the global queue for execution, where P is the number of processors. With this method, at the start of computation, the idle processor will have larger number of iterations to execute. At the end of computation, the idle processor will have only one or two iterations to execute. So the work load will be balanced at most cases. However, GSS does not perform well for NUMA machines. The reason is that GSS does not exploit affinity effect.

Markatos and LeBlanc (1993, 1994) proposed the first affinity loop scheduling algorithm that called AFS. AFS uses a deterministic assignment policy to assign repeated executions of loop iteration to the same processor. The progress of AFS includes initialization phase and execution phase. In the initialization phase, AFS assigns each processor about N/P iterations, where N is the total number of iterations and P is the number of processors. Instead of using a global queue to store all iterations' indices, AFS uses distributed local queues to store the work. The execution phase of AFS follows this rule. Every processor fetches $(1/P)$ of the unscheduled iterations from its local queue for execution again and again until the local queue is empty. If no imbalance occurs before all the iterations are completed, no migration is needed. But if imbalance occurs, some work must be migrated. The idle processor then searches among the other processors for the most loaded one, and migrates $(1/P)$ of the iterations remaining in that work queue to itself for execution. AFS ensures that most of the data accesses are to the cache or to the local memory. AFS also alleviates the contention to global queue for accessing the indices of unscheduled iterations. However, as load imbalance occurs, only a fraction $(1/P)$ of iterations are migrated from the most heavily loaded processor. AFS uses conservative migration policy to avoid load imbalance, but it causes unnecessary synchronization operations (Wang and Chang, 1995).

To reduce the number of synchronization operations, Wang and Chang (1995) proposed a modified migration policy for AFS called MAFS. The main difference between AFS and MAFS is the migration quantum. MAFS divides the iterations remained in the most loaded processor (N^{most} iterations) into two parts: one part contains $N1$ iterations and another one contains $N2$ iterations. $N1$ equals $1/P$ of the total number of iterations in all processors, and $N2$ equals $N^{\text{most}} - N1$. MAFS migrates the minor of the two parts to the idle processor, and the other part is left to the most loaded processor. With this method, MAFS not only migrates a very reasonable load to the idle processor in order to balance the load but also avoids the unnecessary data movement during migration.

However, when programs are executed on clustered NUMA machines, AFS and MAFS have the following disadvantages. First of all, when load imbalance occurs, the data and iterations have to be migrated. However, as the iterations are migrated, affinity algorithms do not consider the locations of the moved data. As we know, the cost of inter-cluster memory access is larger than that of local-cluster memory, some time will be wasted in accessing inter-cluster memory. Furthermore, because the idle processor must search the most loaded processors for migrating work. As the number of processors gets larger, the searching procedure will take many inter-cluster remote accesses to some processors' work queues for the iteration indices. So the synchronization overhead is increased.

3. The hierarchical policy

Like AFS and MAFS, the progress of hierarchical policy is divided into the initialization phase and the execution phase. During the initialization phase, N parallel iterations are also divided into P chunks, and each chunk contains about N/P consecutive iterations, where P is the number of processors. But unlike AFS and MAFS, consecutive chunks are assigned to adjacent processors, instead, they are cyclically distributed to different clusters. By grouping consecutive iterations into chunks and cyclically distributing those chunks into different clusters, we may retain data locality and effectively balance the work load. Fig. 1 shows an example that cyclically distributing 16 chunks (numbered from Chunk1 to Chunk16) of iterations into 4 clusters, and each cluster contains 4 processors. In this example, Cluster1 contains Processor1, Processor2, Processor3, and Processor4, and Chunk1, 5, 9, 13 are assigned to these processors, respectively.

During the execution phase of hierarchical policy, if no imbalance occurs, no migration is needed. But if imbalance occurs, some work must be migrated from heavy loaded processors to the idle ones. We retain the basic skeleton of AFS and MAFS, but make some modifications to the migration policies of AFS and MAFS. The idea of hierarchical policy is that when imbalance occurs, the migration of iterations is carried on hierarchically. The idle processor first migrates a fraction of iterations from the most loaded processor in its local-cluster, before searching other clusters. Only if all of the processors' work queues in the local-cluster are empty, the idle processor does search and migrate work from the other clusters.

Now we apply the hierarchical policy to AFS under the example shown in Fig. 1, we call the modified al-

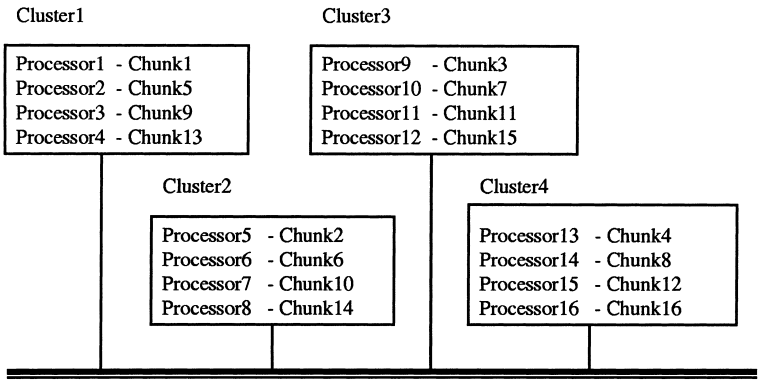

Fig. 1. Distributing 16 chunks of iterations into 4 clusters under hierarchical policy.

gorithm as HAFS. The migration policy of HAFS can be stated as follows:

- During the first stage of migration, the idle processor migrates $1/P_{\text{local-cluster}}$ of the iterations from the most loaded processor within its local-cluster, where $P_{\text{local-cluster}}(=4)$ is the number of processors in its local-cluster.
- If all of the processors' work queues in the idle processor's local-cluster are empty, the idle processor then searches for the most loaded processor among the idle processor's super-cluster. If the idle processor finds the most loaded processor, then it migrates $1/P_{\text{super-cluster}}$ of the iterations from that processor for execution. Since there are only two levels in the system, the value of $P_{\text{super-cluster}}$ is the total number of processors in the whole system ($=16$).

The execution phase of Hierarchical MAFS (HMAFS) can be stated as follows:

- In the first stage of migration, each time the idle processor migrates $\min(N^1, N^2)$ iterations from the most loaded processor in its local-cluster for execution. N^1 equals $N_{\text{local-cluster}}/P_{\text{local-cluster}}$, and N^2 equals $N^{\text{local-most-one}} - N^1$, where $P_{\text{local-cluster}}(=4)$ is the number of processors in the local-cluster, $N^{\text{local-most-one}}$ the number of iterations in the most loaded processor's work queue, and $N_{\text{local-cluster}}$ is the total number of iterations left in the queues of the processors within the local-cluster.
- If all of the processors' work queues in the idle processor's cluster are empty, the idle processor searches the most loaded processor among the idle processor's super-cluster. Then it migrates $\min(N^1, N^2)$ iterations to itself for executing from the most loaded processor. N^1 equals $N_{\text{super-cluster}}/P_{\text{super-cluster}}$, and N^2 equals $N^{\text{super-most-one}} - N^1$, where the value of $P_{\text{super-cluster}}$ is also 16 (the total number of processors in the whole system), $N^{\text{super-most-one}}$ is the number of iterations in the most loaded processor's work queue, and $N_{\text{super-cluster}}$ is the total number of iterations left in the work queues of the processors in the super-cluster.

The hierarchical policy can be easily applied to the other loop scheduling algorithms and we do not describe them here.

4. Experimental environment

To compare the performance of various loop scheduling algorithms under clustered NUMA machines, some effects on the performance must be carefully and correctly characterized. These effects include execution time, remote memory access overhead, and synchronization operation. By the use of simulator, we may easily set the values of architecture parameters into simulation. These parameters include memory access latency, the number of processors, network latency, ..., and so on. Simulation is appropriate for our experiments. In this section, we introduce the clustered NUMA machine simulator we used, and then present our test programs and the characteristics of these programs.

As described in Section 1, we use an on-line, execution-driven simulator to simulate a clustered NUMA multiprocessor with 64 nodes. There are 16 clusters in the system, and each cluster contains 4 nodes. The simulator consists of two parts: Mint (Veenstra and Fowler, 1994) and a clustered NUMA memory system simulator. The applications are input to Mint, and Mint calls the memory system simulator on each memory reference. The memory simulator decides whether the reference is in the cache, in the local-cluster memory, or in the remote cluster memory. We modify and enhance the simple cache simulator provided by Mint (Veenstra and Fowler, 1994). In the modified memory system simulator, each node has a single processor and a finite-size cache which uses a write-invalidate protocol that is directory-based. The simulator also takes into consideration the remote memory access cost and the synchronization overhead.

Each node in the simulator has a 64KB four way set-associative cache with 32B cache line, and the processors in one cluster share 64MB local-cluster memory. We

assume that it takes 1 cycle to access the cache and 25 cycles to access the local-cluster memory (Lenoski et al., 1992; Hennessy and Patterson, 1990). As for inter-cluster memory access, we assume there to be 125 cycles of latency. The ratio of remote to local-cluster memory access is about 5.

We choose the following parallel programs as test suites: Gaussian elimination, all-pairs shortest paths, adjoint convolution, reverse adjoint convolution, and synthetic programs with decreasing and increasing work load. We use barrier synchronization among outer sequential loops under all applications.

The first problem is to perform Gaussian elimination of a 480×480 matrix A. The algorithm to solve the problem can be stated as follows:

```
for (j = 0; j < 480; j++){
   parallel for (i = 0; i < 480, i++){
        if (i <= j) continue;
        tmp = A[i][j]/A[j][j]
        for (k = j; k < 480; k++)
        A[i][k] = A[i][k] - tmp * A[j][k]
   }
}
```

It takes 480 phases to complete the work, and there are 480 parallel iterations for each phase. During the jth phase, the first j parallel iterations have little work to do (each iteration needs just an *if* instruction), but the other $(480 - j)$ parallel iterations have $(480 - j)$ sequential loops to complete its work. Load imbalance will occur in this case. The ith iteration of the 480 parallel iterations always accesses the ith row of matrix. Thus both load imbalance and affinity effects must be studied for this case.

The second program is to compute the all-pairs shortest paths of a graph with 600 vertices, and the graph is represented by a 600×600 matrix A. The pseudo-code to solve the problem is shown as follows:

```
for (k = 0; k < 600; k++){
   parallel for (i = 0; i < 600, i++){
        if (A[i][k] has path)
        for (j = 0; j < 600; j++)
        A[i][j] = min{A[i][j], A[i][k] + A[k][j]}
   }
}
```

For all $0 \leqslant i < 600$ and $0 \leqslant j < 600$, if there exists a path from vertex i to j, $A[i][j]$ equals the value randomly chosen from 1 to 15, or there is no path, and the possibilities of both cases (with path and without path) are equal. The work load of the ith iteration of the 600 parallel iterations depends on $A[i][k]$, and it takes $O(1)$ or $O(N)$ work to complete. As each processor's work queue initially contains about N/P consecutive iterations, the total load for all processors is about the same, it is not necessary to study the imbalance effect. The ith iteration always accesses the ith row of the matrix, so affinity effect must be studied.

The third program is adjoint convolution and the pseudo-code can be stated as follows:

```
parallel for (i = 0; i < 120 * 120; (i++){
      for (j = i; j < 120 * 120, j++)
      A[i] = A[i] + X * B[j] * C[i - j]
}
```

The problem is a case of decreasing work load, but no affinity effect needs to be studied.

The fourth program is reverse adjoint convolution and the pseudo-code can be stated as follows:

```
parallel for (i = 0; i < 120 * 120; i++){
      for (j = 1; j < i, j++)
      A[i] = A[i] + X * B[j] * C[i - j]
}
```

The problem is a case of increasing work load, and no affinity effect needs to be studied.

The fifth and sixth programs are synthetic programs with decreasing and increasing work load (Subramaniam and Eager, 1994). The two problems are cases of load imbalance, and we must take care the affinity effect for them. The size of matrix A is 9600×32, and the pseudo-codes are shown as follows:

```
for (k = 0 ; k < 10, k++) {
   parallel for (i = 0 ; i < 9600 ; i++) {
        for (j = i ; j < 9600, j = j + 32) {
        A[i][j%32] = 1;
        }
   }
}

for (k = 0 ; k < 10, k++) {
   parallel for (i = 0 ; i < 9600 ; i++) {
        for (j = 1 ; j < i, j = j + 32) {
        A[i][j%32] = 1;
        }
   }
}
```

5. Experimental results

We implement static, GSS, AFS, HAFS, MAFS, HMAFS, and CAFS algorithms and run them on the simulator, then we evaluate their performance by running various applications. The metrics of our experiment are execution times, the number of locks to the queues for updating iteration indices, and the number of inter-cluster remote memory accesses.

In our experiment, we assume that each cluster contains 4 processors, and there are at most 16 clusters in the system. As for the distribution of chunks to clusters, as described in Section 3, HAFS and HMAFS algo-

rithms group consecutive iterations into chunks and cyclically distribute those chunks into different clusters. CAFS also cyclically distributes those chunks into different clusters. However, as imbalance occurs, migrations are carried only on within the idle processor's cluster, there are no migrations among clusters. AFS and MAFS distribute those consecutive chunks into adjacent processors, so most consecutive chunks are assigned to the same cluster. To study the impact of different distribution of chunks to clusters, we implement a modified version of AFS algorithm called CD_AFS. The migration policy of CD_AFS is the same as AFS. But unlike AFS and MAFS, consecutive chunks are assigned to adjacent processors, instead, consecutive chunks are cyclically distributed to different clusters.

Fig. 2 shows the execution times for Gaussian elimination problem with 4–24 processors running under various scheduling algorithms. This problem exploits affinity and shows load imbalance effect. AFSs perform better than the GSS and static algorithms. GSS suffers heavy load in accessing inter-cluster memory and the static algorithm suffers from load imbalance. There are no significant differences among AFS, CAFS, MAFS, and CD_AFS. Most important of all, the figure shows that HAFS performs better than AFS and that HMAFS performs better than MAFS. In fact, HMAFS is the best among these algorithms.

To confirm the results described above, we collect the number of inter-cluster memory accesses and the number of locks to the work queues under various algorithms, and the results are shown in Figs. 3 and 4, respectively. Obviously the inter-cluster memory accesses of GSS are the largest among these algorithms, and the accesses of hierarchical algorithms (HAFS and HMAFS) are smaller than the original ones (AFS and MAFS). The results also show that hierarchical policy may reduce the number of locks to the work queues by about half.

Fig. 5 shows the execution times for all-pairs shortest paths problem with 8–24 processors running under various scheduling algorithms. This problem only exploits affinity effect, so GSS is the worst one among all

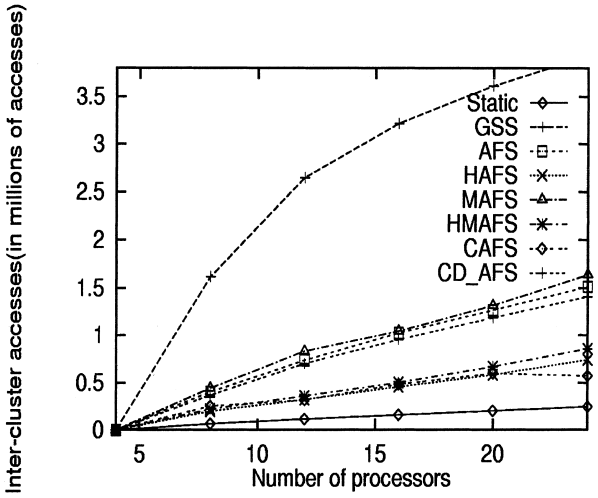

Fig. 3. Inter-cluster remote accesses for Gaussian elimination.

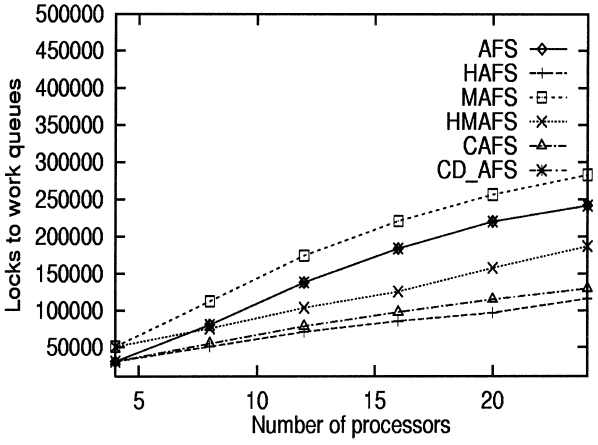

Fig. 4. Locks to the work queues for Gaussian elimination.

algorithms. As migrations rarely occur in this case, there are no significant differences among various affinity algorithms. Static algorithm is the best one among these algorithms, because the synchronization overhead is not needed under this algorithm. However, static algorithm and affinity algorithms show little difference. Fig. 6 shows the inter-cluster memory accesses for these algo-

Loop scheduling algorithms

Number of processors	Static	GSS	AFS	CD_AFS	CAFS	HAFS	MAFS	HMAFS
4	490.0	381.2	346.9	346.9	346.9	346.9	344.6	344.6
8	258.2	256.1	196.1	196.6	204.3	191.1	197.4	190.6
12	178.5	198.0	145.1	146.1	147.0	137.6	147.2	137.0
16	138.3	160.6	119.2	119.1	118.5	110.7	118.7	109.5
20	114.3	136.2	102.5	103.2	100.1	94.4	102.5	93.7
24	98.1	119.2	92.5	92.1	87.0	84.1	92.0	83.7

Execution times (in billions of cycles)

Fig. 2. Execution times for Gaussian elimination.

Loop scheduling algorithms

Number of processors	Static	GSS	AFS	CD_AFS	CAFS	HAFS	MAFS	HMAFS
8	1.091	1.131	1.093	1.092	1.092	1.092	1.093	1.092
12	0.738	0.781	0.741	0.740	0.741	0.740	0.741	0.740
20	0.456	0.495	0.460	0.459	0.459	0.460	0.460	0.460
24	0.386	0.422	0.390	0.389	0.389	0.390	0.391	0.390

Execution times (in billions of cycles)

Fig. 5. Execution times for all-pairs shortest paths.

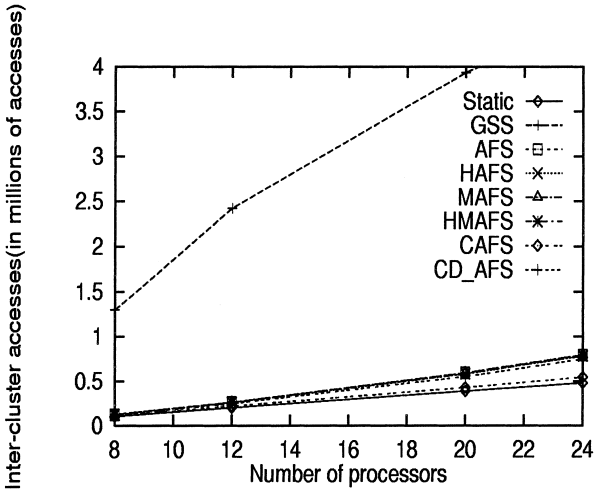

Fig. 6. Inter-cluster remote accesses for all-pairs shortest paths.

rithms. As we have discussed above, GSS suffers heavy inter-cluster accesses, and there are slight differences among various AFSs.

Fig. 7 shows the execution times of the adjoint convolution problem with 4–40 processors running under various scheduling algorithms. The case is an example of decreasing work load, but no affinity effect can be exploited. As the figure shows, affinity scheduling algorithms perform better than GSS and static algorithm. The reason is that affinity algorithms well balance the work load at run time. But GSS assigns too much work to the processors at the start of computation, and static algorithm does not balance the work load at run time. This figure also shows that CD_AFS and AFS are worse than the other affinity algorithms, and that both MAFS and HMAFS are the better ones among these algorithms. The reason is that under CD_AFS and AFS policies, the idle processor migrates too much work from inter-cluster processors as the number of processor increases. As Fig. 8 shows, AFS and CD_AFS have the largest number of inter-cluster memory accesses among these algorithms. Fig. 9 shows the number of locks to the work queues. As the figure shows, hierarchical algorithms reduce at least $1/2$ of the locks compared with the original ones.

Fig. 10 shows the execution times of the reverse adjoint convolution problem with 4–40 processors running under various algorithms. The case is an example of increasing work load, but no affinity effect can be exploited. The result is similar to that of adjoint convolution except that GSS performs almost as good as both MAFS and HMAFS. The reason is that with increasing work load, the first a few iterations have light work load but the last a few ones have heavy work load. According to the policy of GSS, at the beginning of loops, chunks with large number of iterations are assigned to the processors, but at the end of loops, single iteration is assigned. In this way, variance of work load among iterations will not dominate the performance of GSS. So GSS will well balance the work load. As for the number of inter-cluster memory accesses and the locks to the work queues, Figs. 11 and 12 shows the results under

Loop scheduling algorithms

Number of processors	Static	GSS	AFS	CD_AFS	CAFS	HAFS	MAFS	HMAFS
4	1.861	1.861	1.067	1.067	1.067	1.067	1.065	1.065
8	0.998	0.998	0.542	0.547	0.541	0.540	0.538	0.537
12	0.680	0.680	0.373	0.376	0.368	0.364	0.361	0.361
20	0.416	0.416	0.242	0.249	0.243	0.230	0.218	0.220
24	0.348	0.348	0.215	0.218	0.202	0.197	0.184	0.186
40	0.211	0.211	0.161	0.165	0.145	0.125	0.113	0.115

Execution times (in billions of cycles)

Fig. 7. Execution times for adjoint convolution.

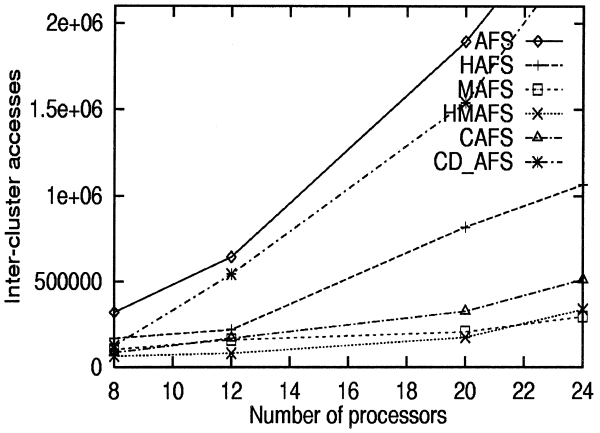

Fig. 8. Inter-cluster remote accesses for adjoint convolution.

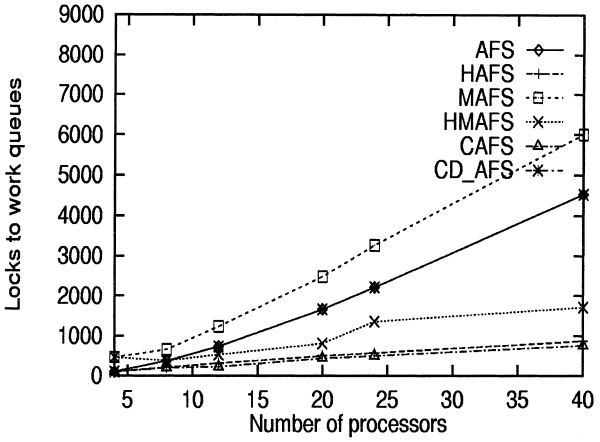

Fig. 9. Locks to the work queues for adjoint convolution.

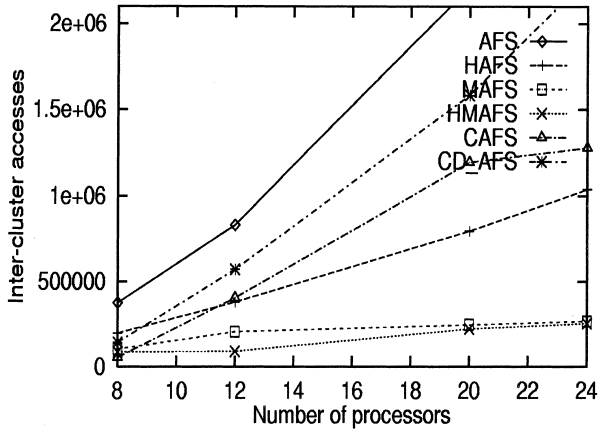

Fig. 11. Inter-cluster remote accesses for reverse adjoint convolution.

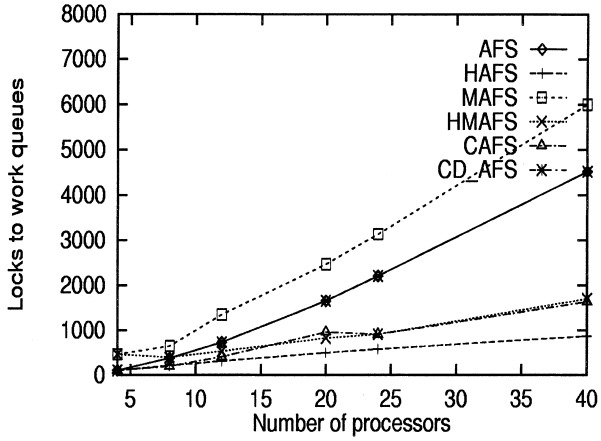

Fig. 12. Locks to the work queues for reverse adjoint convolution.

various AFSs, and the results are similar to those of adjoint convolution.

The execution times of the fifth problem, synthetic problem with decreasing work load, is shown in Fig. 13. In this case, we must consider load imbalance and affinity effects. But the load imbalance effect is lighter than that of adjoint convolution, and affinity effect is also lighter than that of Gaussian elimination. As the figure shows, AFSs perform better than static algorithm and GSS. The reason is that affinity algorithms not only exploit affinity but also well balance the work load at run time. But static algorithm does not balance the work load at run time. GSS cannot exploit affinity effect and suffers load imbalance because of the heavy load for the first a few iterations. The figure also shows that HAFS is better than AFS, and that HMAFS is better than MAFS. Again, because AFS suffers too many inter-cluster memory accesses, AFS is the worst among these

Loop scheduling algorithms

Number of processors	Static	GSS	AFS	CD_AFS	CAFS	HAFS	MAFS	HMAFS
4	1.952	1.118	1.067	1.067	1.067	1.067	1.065	1.065
8	1.046	0.563	0.571	0.569	0.565	0.571	0.563	0.563
12	0.713	0.379	0.392	0.394	0.389	0.386	0.379	0.377
20	0.436	0.233	0.257	0.260	0.247	0.241	0.231	0.230
24	0.365	0.196	0.223	0.230	0.207	0.204	0.194	0.193
40	0.222	0.123	0.167	0.170	0.150	0.130	0.118	0.121

Execution times (in billions of cycles)

Fig. 10. Execution times for reverse adjoint convolution.

Loop scheduling algorithms

Number of processors	Static	GSS	AFS	CD_AFS	CAFS	HAFS	MAFS	HMAFS
4	145.94	145.99	84.16	84.16	84.16	84.16	84.13	84.13
12	53.37	53.67	28.85	28.65	28.70	28.58	28.60	28.58
16	40.50	40.74	22.00	21.75	21.78	21.61	21.63	21.55
24	27.35	27.50	15.22	14.95	14.74	14.72	14.71	14.65
40	16.62	16.71	9.92	9.63	10.37	9.23	9.30	9.15

Execution times (in millions of cycles)

Fig. 13. Execution times for synthetic problem with decreasing work load.

affinity algorithms. HMAFS is the best one because it reduces both inter-cluster memory accesses and locks to the work queues. Figs. 14 and 15 show the number of inter-cluster memory accesses and the number of locks to the work queues. The figures show that hierarchical policy may reduce a lot of inter-cluster memory accesses and locks to the work queues.

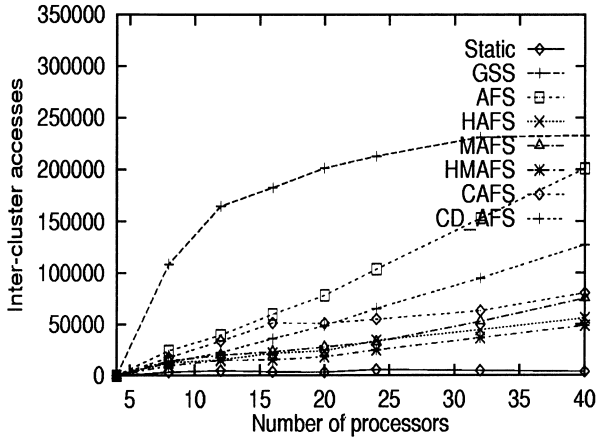

Fig. 14. Inter-cluster remote accesses for synthetic problem with decreasing work load.

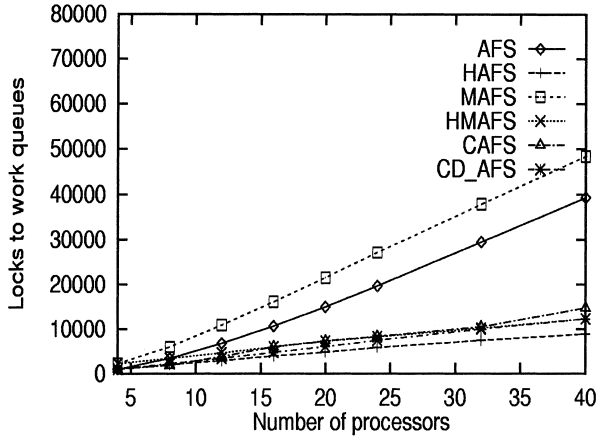

Fig. 15. Locks to the work queues for synthetic problem with decreasing work load.

The execution times of the last problem, synthetic problem with increasing work load, is shown in Fig. 16. The same as the previous problem, this is a case of load imbalance and affinity. The result is similar to that of the previous one, except that as the number of processor increases, GSS performs as good as AFS. The reason is that non-affinity will reduce the performance of GSS, but the iteration assigned policy of GSS well balances the work load. As the number of processor increases, the affinity effect will become insignificant. As is shown in Fig. 17, if the number of processors increases, the differences between GSS and affinity algorithms decrease. Fig. 17 also shows that the numbers of inter-cluster memory accesses under hierarchical algorithms are smaller than those under original ones. The result of the locks to the work queues is shown in Fig. 18, and it is similar to that of the previous problem.

It is interesting to study the impact of memory latency on loop scheduling algorithms on NUMA machines. We select Gaussian elimination as test application, and implement static, GSS, AFS, and HAFS on the simulator. As to the local/remote memory latency, we have the following four assumptions:

- *Case 1*: We assume that all local/inter-cluster memory accesses take no time to complete, it is the ideal case.
- *Case 2*: We assume that all memory accesses are to the local memory, and they take 25 cycles to complete.
- *Case 3*: This case is the same as the previous experiments in this section, that is, it takes 25 cycles to access the local memory, and it takes 125 cycles to access the inter-cluster memory.
- *Case 4*: The case is similar to case 3 except that it takes 225 cycles to access the inter-cluster memory.

Fig. 19 shows the executions for Gaussian problem with 12 and 24 processors running under various assumptions of memory latency. The results show that:

- The impact of memory latency on GSS algorithm is more significant than that of the other algorithms. The reason is that the inter-cluster memory accesses of GSS are the largest among these algorithms. The execution times increase significantly as the cost of inter-cluster memory access gets larger.

Loop scheduling algorithms

Number of processors	Static	GSS	AFS	CD_AFS	CAFS	HAFS	MAFS	HMAFS
4	158.55	93.29	91.43	91.43	91.43	91.43	91.41	91.41
12	58.98	33.19	31.35	31.11	31.18	30.97	31.07	31.10
16	44.00	25.20	23.85	23.63	23.63	23.43	23.48	23.38
24	29.70	17.01	16.45	16.19	16.01	15.93	15.98	15.87
40	18.04	10.49	10.69	10.36	11.16	9.95	10.07	9.86

Execution times (in millions of cycles)

Fig. 16. Execution times for synthetic problem with increasing work load.

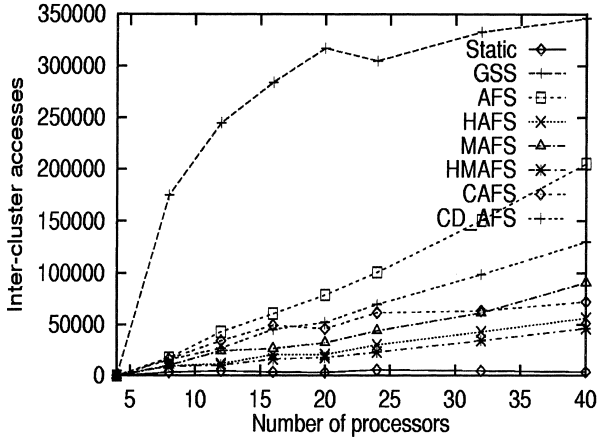

Fig. 17. Inter-cluster remote accesses for synthetic problem with increasing work load.

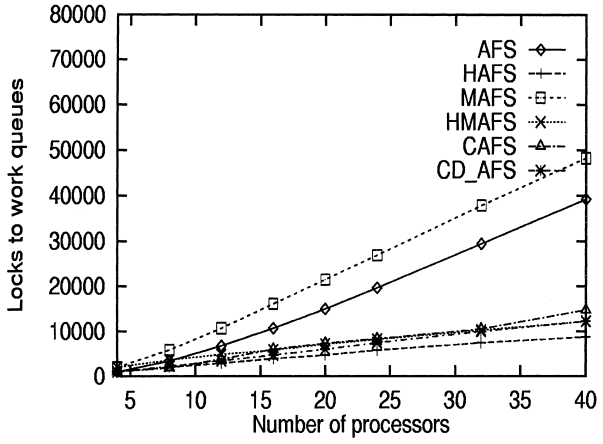

Fig. 18. Locks to the work queues for synthetic problem with increasing work load.

- The impact of memory latency on static algorithm is the least significant among all scheduling algorithms. The execution times do not increase significantly under the influence of high inter-cluster memory cost. However, as static algorithm suffers from load imbalance, static algorithm does not perform well.
- Among these algorithms, HAFS performs better than the other algorithm. The reason is that HAFS not only well balances the work load but also reduces the number of inter-cluster accesses.

6. Conclusion

As modern large-scale multiprocessors are clustered, have high speed processors and hierarchical non-uniformed slow memory access time, communication becomes the most important consideration in the development of high performance multiprocessors (Markatos and LeBlanc, 1992; Markatos and LeBlanc, 1994; Crovella et al., 1991). Though affinity algorithms as AFS (Markatos and LeBlanc, 1994), MAFS (Wang and Chang, 1995), and LDS (Li et al., 1993) perform well for non-clustered NUMA multiprocessors, they do not run efficiently on clustered NUMA machines. There are two major reasons. As load imbalance occurs, iterations migration and data movements are needed. However, these affinity algorithms do not guarantee that most of the remote memory accesses are as close to the processor as possible. As the locations of memory accesses are ignored, too many inter-cluster memory accesses occur. The other is that the migration overhead becomes heavy as the number of processors increases. The overhead includes more remote accesses to the work queues for the iteration indices and more locks to update the iteration indices.

In this paper, a new scheduling policy, called hierarchical scheduling policy, is proposed to improve various affinity algorithms under clustered NUMA machines. Under this policy, migrations are carried on hierarchically. The policy may reduce the number of inter-cluster memory accesses, decrease the number of locks to the work queues, and well balance the work load. We apply the new policy to AFS and MAFS, and make them HAFS and HMAFS, respectively. We confirm our idea by running various applications under a realistic clustered NUMA simulator. Our experimental results show that:

- Affinity algorithms perform better than GSS and static algorithm. The reason is that affinity algorithms not only retain affinity but also balance the work load at run time.

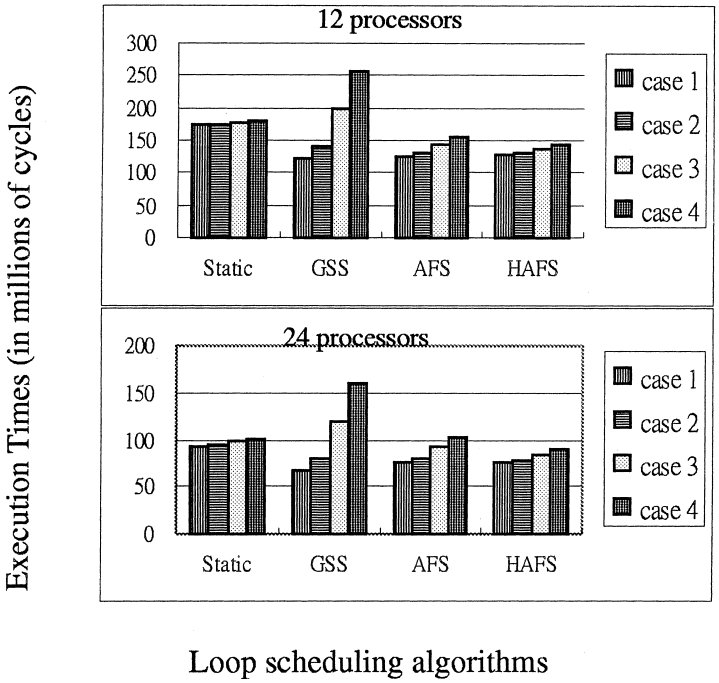

Fig. 19. Execution times for Gaussian elimination problem with various memory latencies.

- Hierarchical affinity algorithms perform better than the original ones because they reduce many inter-cluster memory accesses as well as a lot of locks to the work queues. For example, HAFS is better than AFS and HMAFS is better than MAFS.
- HMAFS is the best choice among these algorithms in most cases.

Acknowledgements

This research work was partially supported by the National Science Council of Republic of China under grant No. NSC88-2213-E126-004 and NSC89-2213-E-126-009.

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. In: Proceedings of the 22nd International Symposium on Computer Architecture, pp. 2–13.

Crovella, M., Das, P., Dubnicki, C., Markatos, E.P., LeBlanc, T.J., 1991. Multiprogramming on multiprocessors. In: Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing, pp. 590–597.

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

Hummel, S.F., Schonberg, E., Flynn, L.E., 1992. Factoring: a practical and robust method for scheduling parallel loops. Communications of the ACM 35 (8), 90–101.

Kruskal, C.P., Weiss, A., 1985. Allocating independent subtasks on parallel processors. IEEE Transactions on Software Engineering SE-11 (10), 1001–1016.

Lenoski, D., Laudon, J., Nakahira, D., Stevens, L., Gupta, A., Hennessy, J., 1992. The dash prototype: implementation and performance. In: 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. In: 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. University of Rochester, Computer Science Department, 1992.

Markatos, E.P., LeBlanc, T.J., 1993. Scheduling for locality in shared-memory multiprocessors. Ph.D. Thesis. University of Rochester, Computer Science Department.

Markatos, E.P., LeBlanc, T.J., 1994. Using processor affinity in loop scheduling on shared-memory multiprocessors. IEEE Transactions 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 Transactions on Computers C-36 (12), 1425–1439.

Stumm, M., Vranesic, Z., White, R., Unrau, R., Farkas, K., 1992. Experiences with the hector multiprocessor. Technical Report CSRI-276. University of Toronto.

Subramaniam, S., Eager, D.L., 1994. Affinity scheduling of unbalanced work loads. Supercomputing'94, 214–226.

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

Veenstra, J.E., Fowler, R.J., 1994. Mint: a front end for efficient simulation of shared-memory multiprocessors. In: Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, pp. 201–207.

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.

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

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 NUMA shared-memory 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 wass 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, Taiwan, 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-Hsi 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, distributed systems, and algorithm design.

Ruei-Chaun 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 an 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.