NORTH-HOLLAND

# Clustered Affinity Scheduling on Large-Scale NUMA Multiprocessors*

Yi-Min Wang

*Institute of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, R.O.C.*

Hsiao-Hsi Wang

*Department of Computer Science and Information Management, Providence University, Taichung, Shalu, Taiwan, R.O.C.*

Ruei-Chuan Chang

*Institute of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, R.O.C., and Institute of Information Science, Academia Sinica, Nankang, Taipei, Taiwan, R.O.C.*

Modern shared-memory multiprocessors have high and non-uniform memory access (NUMA) costs. The communication cost gradually dominates the source of parallel applications' execution. Algorithms based on affinity, like affinity scheduling algorithm (AFS), perform better than dynamic algorithms, such as guided self-scheduling (GSS) and trapezoid self-scheduling (TSS). However, as the number of processors increases, AFS suffers heavy overheads for migrating workload. The overheads include remote reads to the queues for the indices information, synchronous writes to the queues for migrating iterations, and the time in loading data into cache. In this paper, we propose a new loop scheduling algorithm, clustered affinity scheduling (CAFS), to improve affinity scheduling algorithm. We distribute the processors into several clusters, and cluster-based migrations are carried on when imbalance occurs. We confirm our idea by running many applications under a realistic hierarchy memory simulator. Our results show that CAFS reduces at least 1/3 of both remote reads and synchronous writes to the queues under most applications. CAFS also improves the cache hit ratios, and balances the workload. Therefore, we conclude that under large NUMA multiprocessor, CAFS is a better choice among loop scheduling algorithms. ©1997 Elsevier Science Inc.

## 1. INTRODUCTION

Parallel loops execution is one of the best ways to evaluate the performance of multiprocessors. By carefully scheduling the loops on multiprocessors, we may shorten the execution time of parallel applications. Two basic loop scheduling algorithms, static algorithms and dynamic algorithms, are studied extensively under traditional multiprocessors [Tzen and Ni, 1993; Hummel et al., 1992; Kruskal and Weiss, 1985; Polychronopoulos and Kuck, 1987]. Because the memory access costs are low for all processors, the communication effect can be ignored under these scheduling algorithms. The main considerations of these algorithms are load balance and synchronization overhead.

Modern shared-memory multiprocessors have relatively high and/or non-uniform memory access costs, and the new architecture is an important trend in the design of high performance computers. Much effort has been done to develop non-uniform memory access (NUMA) architectures both in academy and in industrial research departments. Toronto

*Address correspondence to Ruei-Chuan Chang, Institute of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan, R.O.C. E-mail: (rc@iss.sinica.edu.tw). Phone no.: 886-35-712121 ext. 52801, 56656 Fax: 886-35-721490.*

* *This research work was partially supported by the National Science Council of the Republic of China under grant No. NSC83-0408-E009-005.*

J. SYSTEMS SOFTWARE 1997; 39:61–70
© 1997 Elsevier Science Inc. All rights reserved.
655 Avenue of the Americas, New York, NY 10010

0164-1212/97/$17.00
PII S0164-1212(96)00163-X

HECTOR [Vranesic et al., 1991], MIT Alewife [Agarwal et al., 1995], and Stanford Dash [Lenoski et al., 1992] are some examples. After the evolution of multiprocessor architecture, loop scheduling algorithms running on old multiprocessors can not run efficiently on shared-memory NUMA machines. The relatively high and non-uniform remote memory access cost in NUMA machines becomes the third dimension and gradually dominates the performance of multiprocessors [Crovella et al., 1991; Squillante and Lazowska, 1993; Markatos and LeBlanc, 1992b; Markatos and LeBlanc, 1994].

To alleviate remote memory access cost, it is necessary to ensure that most memory accesses are in the local memory or in the cache. Markatos and LeBlanc [Markatos and LeBlanc, 1994] proposed a dynamic scheduling algorithm called affinity scheduling algorithm (AFS). Initially, AFS deterministically schedules the loop iterations to all processors so that each processor has the same number of iterations. The same iterations are assigned to the same processors again and again so that the affinity effect may be retained. Each processor maintains a local queue to store the iteration indices of its own remaining work. During the execution phase, each processor fetches a constant ratio of the remaining work from its own queue until the queue is empty. If any processor is idle, it searches among the other $(p - 1)$ processors for the one with the largest number of iterations and migrates $\lceil 1/p \rceil$ of the iterations from that processor to itself for execution, where $p$ is the number of processors. The experimental results confirm that AFS performs better than the other dynamic scheduling algorithms.

However, AFS will suffer heavy overhead in migrating work as the number of processors gets larger. When a processor is idle, it searches for the processor with the largest number of iterations. The searching procedure takes a lot of remote reads to the other $(p - 1)$ processors' local queues for the iteration indices. Moreover, the migration quantum of AFS may be insufficient so that further migrations are needed. These migrations result in further synchronous writes which update the iteration indices and waste time in loading data into cache. In this paper we propose a new affinity scheduling algorithm called clustered affinity scheduling algorithm (CAFS) under large NUMA machines. CAFS reduces both remote reads and synchronous writes to the queues. It also increases the cache hit ratios, and balances the workload.

The initialization phase of CAFS, the same with that of AFS, deterministically and evenly distributes loop iterations to all processors. We distribute the processors into several clusters, and each processor belongs to a dedicated cluster. When migrating iterations, the idle processor migrates a fraction of iterations from the most loaded processor in its own cluster. Since the idle processor only accesses a dedicated cluster's local queues for the iteration indices, the remote reads and the synchronous writes to the local queues will be reduced. Moreover, because the idle processor migrates work only from some dedicated processors, the cache will not be polluted severely and the possibility of data reusability will be increased. Thus the cache affinity will be retained and data movement traffic will be alleviated. By distributing processors into several clusters in a special order, we may reduce load imbalance between the clusters.

We use an on-line, execution-driven simulator to simulate a scalable NUMA multiprocessor with 128 nodes. The simulator consists of two parts: Mint [Veenstra and Fowler, 1994] and a NUMA hierarchy memory simulator. Mint calls the hierarchy memory simulator on each memory reference, and the memory simulator must decide whether the reference is in the cache, in the local memory, or in the remote memory. To simulate a NUMA environment more realistically and to capture the communication overheads correctly, we modify and enhance the simple cache simulator provided by Mint [Veenstra and Fowler, 1994]. The simple cache simulator is a demonstration of a user-provided system simulator, but the cache size of the simple cache simulator is infinite, and the coherent protocol is bus-based. Each node in our modified hierarchy memory simulator has a processor and a finite-size cache. The caches use directory-based and write-invalidate protocol. The simulator also takes into consideration the latency of memory contention, synchronization operations, and the remote memory access cost.

We carefully chose Gaussian elimination, all-pairs shortest paths, adjoint convolution, and a synthetic program as applications. By running various applications on the simulator, we characterize the execution times, synchronization overhead, and cache behaviors for various scheduling algorithms. Compared with AFS, CAFS may reduce at least 1/3 of the synchronization operations and improve the cache hit ratios as the number of processors gets larger. Our results confirm that the execution times of CAFS are shorter than those of AFS and guided self-scheduling (GSS). Therefore, we conclude that under large NUMA multiprocessors, CAFS is a better choice among loop scheduling algorithms.

The organization of this paper is as follows: In section 2, the clustered AFS algorithm is described;

in section 3, we describe our experimental environment; in section 4, we show the results of simulations under various loop scheduling algorithms; finally, the conclusion is given in section 5.

## 2. THE ALGORITHM

In this section, first we describe the original AFS algorithm in detail. Then we describe the main idea of CAFS and compare it with AFS.

AFS consists of two phases: initialization phase and execution phase. To make full use of affinity, AFS deterministically assigns a chunk of iterations to the same processor again and again in initialization phase. During this phase, AFS assigns each processor about $N/p$ iterations, where $N$ is the total number of iterations and $p$ is the number of processors. That is, $N$ iterations are divided into $P$ chunks, and the $i$th chunk is assigned to the $i$th processor deterministically. If no imbalance occurs before all the iterations are completed, then migration is not needed. But if imbalance occurs, some iterations must be migrated from loaded processors to the idle one.

The execution phase of AFS follows this rule: Every processor fetches $\lceil 1/k \rceil$ (in general, we assume $k = p$) of the remaining iterations from its local queue for execution again and again until the local queue is empty. The idle processor then searches among the other processors for the work queue with the largest number of iterations and migrates $\lceil 1/p \rceil$ of the iterations remaining in that queue to itself. The searching procedure requires many remote memory accesses to the other processors' local queues. AFS performs well under small-scale machines, but can not run efficiently under large NUMA machines. For one thing, during the searching procedure, it wastes much effort in accessing the distributed local queues. For another, as $p$ is large, the migration quantum will be insufficient so that further migrations are needed.

The initialization phase of CAFS is the same as that of AFS. But we make the following modifications to the execution phase of AFS:

- We divide the processors into $C$ clusters, and each cluster contains about $S = p/C$ processors, where the value of $C$ is about $\lceil \sqrt{p} \rceil$ and $p$ is the number of processors. *Processor*$1, 2, \ldots,$ and $C$ are assigned to *cluster*$1, 2, \ldots,$ and $C$ in sequence. But *processor*$(C+1), C+2, \ldots,$ and $2C$ are assigned to the clusters in reverse order,..., and so on. Each processor belongs to one dedicated cluster. During the initialization phase, $N$ iterations are divided into $P$ chunks, and the $i$th chunk is deterministically assigned to the $i$th processor, where $N$ is the number of the total iterations and $P$ is the number of processors. It is possible that processors have various execution times under such cases as increasing or decreasing work loads. The order we use may evenly distribute the work load between the clusters so as to reduce the load imbalance between the clusters. The experimental results in section 4 will show that load imbalance between the clusters is slight so that migration between the clusters is unnecessary in most cases.
- Each time, a processor gets $\lceil 1/S \rceil$ of the remaining iterations from its local queue for execution, and this is done again and again until the local queue is empty. If no imbalance occurs before all the iterations are completed, then migration is not needed.
- When imbalance occurs, CAFS migrates $\lceil 1/S \rceil$ iterations from the processor with the largest number of iterations to the idle one. Instead of searching the other $(p-1)$ processors, CAFS searches only the other processors in its cluster.

The main difference between AFS and CAFS is that under CAFS, we simply need to search the local queues of one cluster. Thus remote reads and synchronization writes to the queues are reduced. We also alleviate the contention to the queues for the indices of iterations. Another advantage of CAFS is that the pollution of caches will be reduced, and thus the cache hit ratios are increased. The processors are divided into several clusters, and the idle processor can not migrate work from the other clusters.

Figure 1 shows an example of CAFS in which 16 processors are distributed into 4 clusters and 16 chunks are assigned to those processors. In the example, we assume that the work loads of the chunks are linearly increasing and that the load of *Chunki* is $i$. The example shows that the load imbalance between the clusters is slight because those clusters have almost the same amount of work load. Therefore, if any processor in *Cluster*0 is idle, it only searches for the most loaded processor in *Cluster*0 and migrates some work from that processor for execution.

## 3. EXPERIMENTAL ENVIRONMENT

Since our studies focus on the evaluations of various loop scheduling algorithms under NUMA machines, some effects on the performance must be carefully and correctly characterized. These effects include

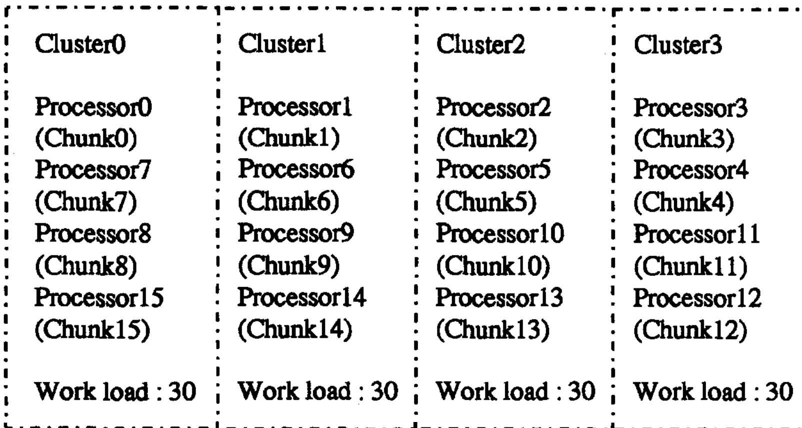


**Figure 1.** Distributing 16 processors into 4 clusters of CAFS.

cache behaviors, remote memory access overhead, synchronization operations, memory contention and load balance (execution time). Simulation is appropriate for our experiments. In this section, we will introduce the realistic NUMA machine simulator we use, and then present the applications we choose.

As is described in the first section, we use an on-line, execution-driven simulator to simulate a scalable NUMA multiprocessor with 128 nodes. The simulator consists of two parts: Mint [Veenstra and Fowler, 1994] and a NUMA machine hierarchy memory simulator. The real applications are input to Mint, and Mint calls the hierarchy memory simulator in each memory reference. The memory simulator decides whether the reference is in the cache, in the local memory, or in the remote memory. We modify and enhance the simple cache simulator provided by Mint [Veenstra and Fowler, 1994]. In the modified hierarchy memory simulator, each node has a single processor and a finite-size cache which uses directory-based and write-invalidate protocol. The simulator also takes into consideration the latency of memory contention, the remote memory access cost, and the synchronization overhead.

Each node in the simulator has a $64KB$ four way set-associative cache with 32-byte cache line, and $16MB$ local memory. We assume that it takes 1 cycle to access the cache and 10 cycles to access the local memory [Hennessy and Patterson, 1990], and 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. We assume there to be 25 cycles of network latency, and network collisions are ignored. This assumption is similar to what Bianchini et al. did in [Bianchini et al., 1994], but the value is smaller than that they used. 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 60 cycles to complete its work, but in case the access is rejected, 50 cycles will be wasted. The ratio of remote to local memory access is about 6. The communication overheads under modern NUMA machines will be larger than those under our simulation. We use an optimistic experimental environment during the simulation because we believe that if CAFS performs well under our simulation, it must be a good choice under NUMA machines in the near future.

Our applications consist of the following parallel programs: Gaussian elimination, all-pairs shortest paths, adjoint convolution, and a synthetic program.

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 = j + 1; i < 480, i + + ){
            tmp = A[i][j]/A[j][j]
            for (k = j; k < 480; k + + )
            A[i][k] = A[i][k] - tmp * A[j][k]
    }
}
```

Each element in the matrix occupies 4 bytes. It takes 480 phases to complete the work, and we use barrier synchronization among different phases. Load imbalance will not occur in this case, and the $i$th iteration of parallel loop always accesses the $i$th row of matrix. Thus affinity is the only effect to be exploited.

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 $A$. For all $0 \le i < 600$ and $0 \le j < 600$, if there exists a path from vertex $i$ to $j$, $A[i][j]$ equals the value randomly chosen from 5 to 9. But there may be no path, and both cases share equal possibilities. The pseudo

J. SYSTEMS SOFTWARE 1997; 39:61–70



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]}
    }
}
```

Each element in the matrix occupies 2 bytes, and each cache line may hold 16 elements. It takes 600 phases to complete the work, and we use barrier synchronization among different phases. The work load of the $i$th iteration of parallel loop depends on $A[i][k]$, and it takes $O(1)$ or $O(N)$ times to complete the work. The $i$th iteration in parallel loop always accesses the $i$th row of the matrix. So the application is to exploit both load imbalance and affinity effects.

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]
}
```

Each element in the matrix occupies 4 bytes, and it only takes 1 phase to complete the work. The application is a case of load imbalance, but no affinity effect need be considered. Thus the only effect to be considered is load imbalance.

The fourth program is a synthetic one [Subramaniam and Eager, 1994], and the pseudo is shown as follows:

```
for (k = 0; k < 25, k + + ){
    parallel for (i = 0; i < 9600; i + + ){
            for (j = i; j < 9600, j = j + 8){
            A[i + 1][j%32] = A[i + 1][j%32] + 1;
            A[i][j%32] = A[i][j%32] + 1;
            A[i - 1][j%32] = A[i - 1][j%32] + 1;
            }
    }
}
```

The size of matrix $A$ is $9600*32$, and each element of $A$ occupies 2 bytes. It takes 25 phases to complete the work, and each phase contains 9600 parallel iterations. This application is a case of decreasing load and it shows a little affinity effect.

## 4. EXPERIMENTAL RESULTS

To compare the performance of various scheduling algorithms, we implement GSS, AFS, and CAFS on the simulator. Then we evaluate the performance of these algorithms by running various applications on the simulator. The metrics of our experiment are the execution times, the number of remote reads to the local queues for iteration indices information, the synchronous writes to update iteration indices for migrations, and the cache miss ratios.

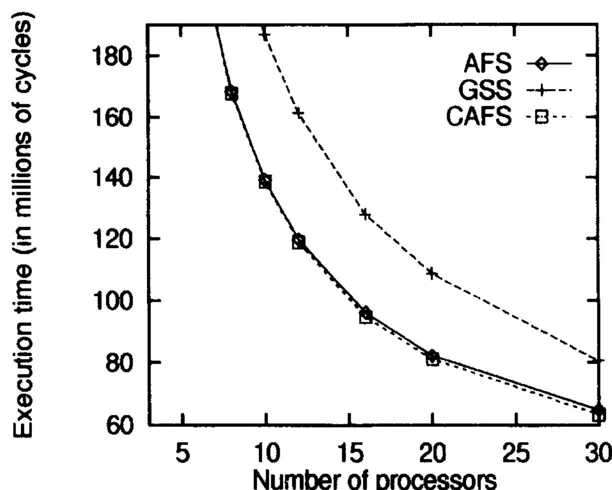


**Figure 2.** Execution times for Gaussian elimination.

Figure 2 shows the execution times for Gaussian elimination problem under various scheduling algorithms. This problem is an example to exploit affinity, but it is a case of load balance. Thus both affinity algorithms perform better than GSS because of retaining affinity. Because migrations rarely occur under the problem, the differences in execution times between AFS and CAFS are not significant. Figure 3 shows the cache miss ratios for these algorithms. Obviously the cache miss ratios of GSS are the largest among the three algorithms, and the ratios of CAFS are slightly better than those of AFS.

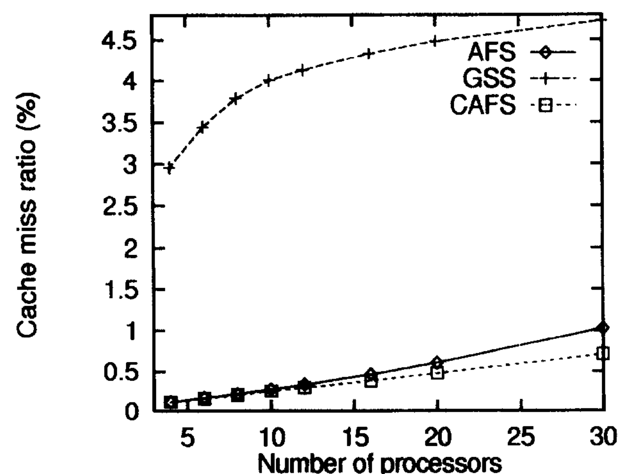


**Figure 3.** Cache miss ratios for Gaussian elimination.

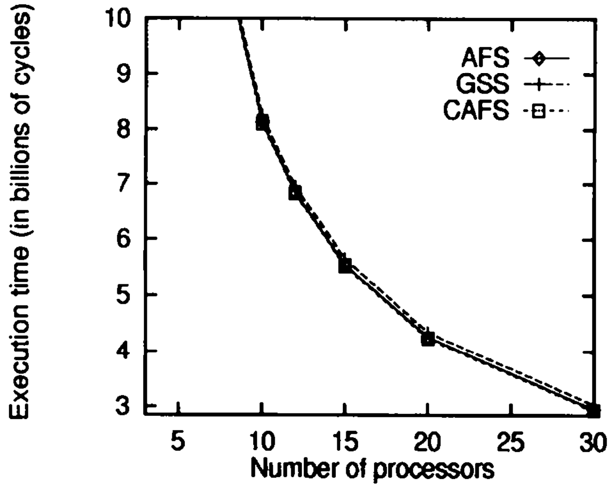


Figure 4. Execution times for all-pairs shortest paths.

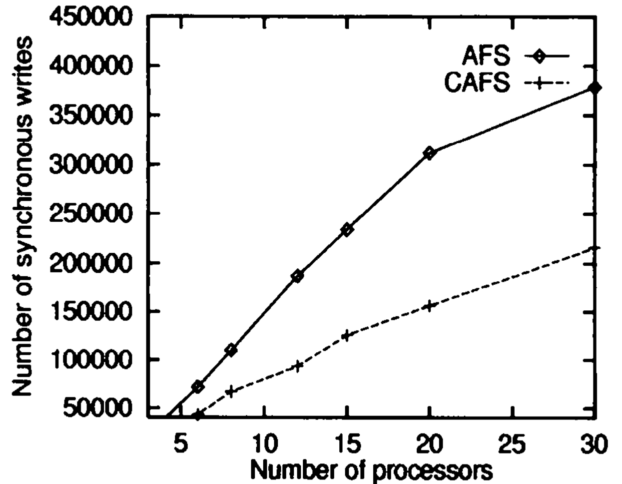


Figure 6. Synchronous writes to queues for all-pairs shortest paths.

Figure 4 presents the execution times for all-pairs shortest paths problem. It shows that AFS and CAFS are still better than GSS, and CAFS performs slightly better than AFS. All-pairs shortest paths problem is another example of exploiting affinity, but each cache line in this case can hold more data elements ($=16$) than in the case of Gaussian elimination problem ($=8$). Moreover, the frequency of data updating is much lower under all-pairs shortest paths problem. So the affinity effect is lighter and the cache miss ratios for those algorithms are also lower. Figure 5 shows the cache miss ratios for various algorithms. The cache miss ratios of GSS are still the largest among these algorithms, and the ratios of CAFS are lower than those of AFS. Because the case is an input dependent case of load imbalance, some migrations are needed during execution phase. CAFS may reduce a lot of remote reads and synchronous writes to the local queues. Figure 6 and Figure 7 show the synchronous writes and remote reads to local queues for CAFS and AFS by running 6 to 30 processors. They show that CAFS may reduce about $1/2$ of the synchronous writes and about $1/3$ of the remote reads to local queues.

The results of Gaussian elimination and all-pairs shortest paths confirm that affinity scheduling algorithms perform better than GSS. CAFS performs a little better than AFS as the work load is in balance or a little imbalance. The reason is that some migration overhead is reduced, but the difference is not of great significance.

Figure 8 shows the execution times of the adjoint convolution problem by running 12 to 60 processors.

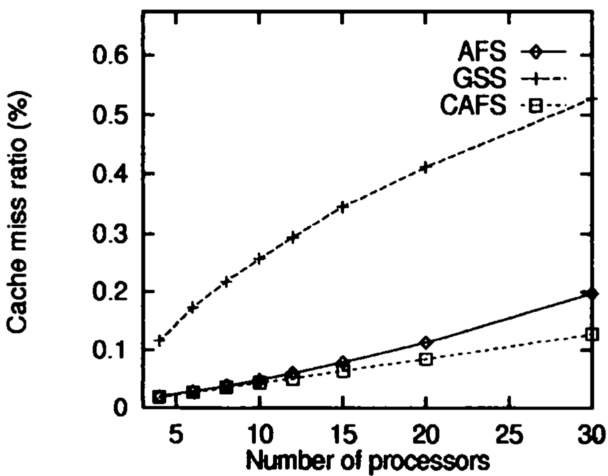


Figure 5. Cache miss ratios for all-pairs shortest paths.

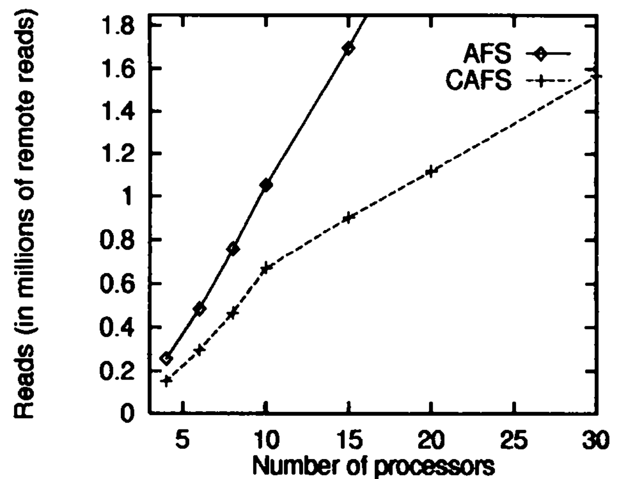


Figure 7. Remote reads to queues for all-pairs shortest paths.

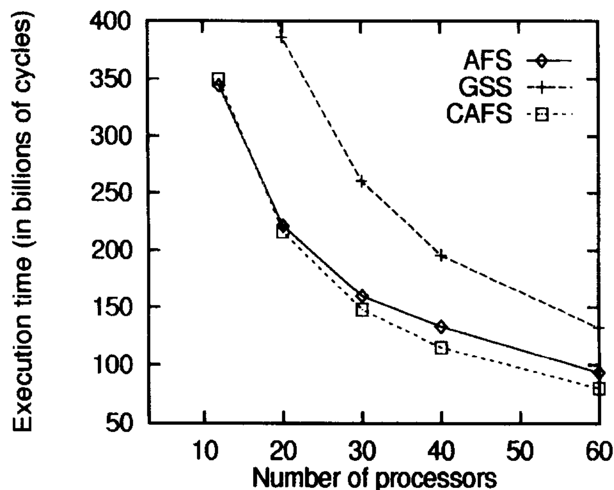


**Figure 8.** Execution times for adjoint convolution.

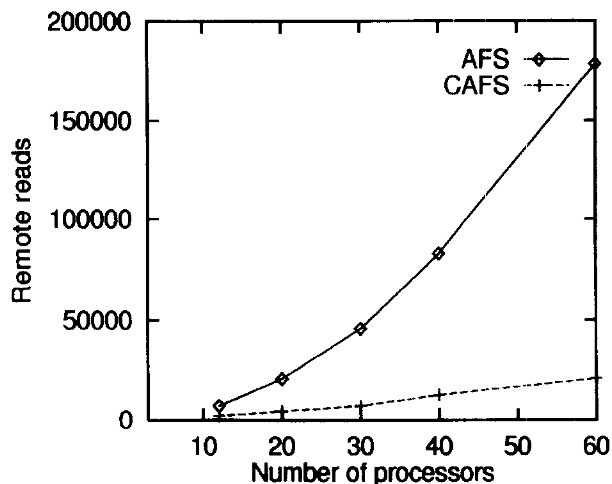


**Figure 10.** Remote reads to queues for adjoint convolution.

The case is an example of decreasing work load, but no affinity effect needs to be exploited. As the figure shows, GSS is the worst of these algorithms. The reason is that GSS assigns too much load to the processors with the first few iterations. CAFS performs better than AFS except in the case of running fewer than 12 processors. The reason is that as the number of processors is small, CAFS will suffer a little load imbalance. CAFS divides the processors into several clusters, and work load can not be migrated among processors in different clusters. However, as the number of processors gets larger, CAFS performs better than AFS because it reduces a lot of remote reads and synchronous writes. As the number of processors gets larger, the difference between CAFS and AFS is more significant. Figure 9 and Figure 10 show the synchronous writes and remote reads to the local queues for AFS and CAFS under adjoint convolution. They show that CAFS reduces 2/3 to 3/4 of the synchronous writes, and CAFS also eliminates a lot of remote reads to local queues for iteration indices.

The execution times of the last problem, synthetic problem with decreasing work load, is shown in Figure 11. The problem is a case of load imbalance and a little affinity. So both cache miss ratios and synchronization overheads must be considered. Again, GSS is the worst of these algorithms. CAFS is better than AFS except under small-scale environments. As the number of processors increases, CAFS

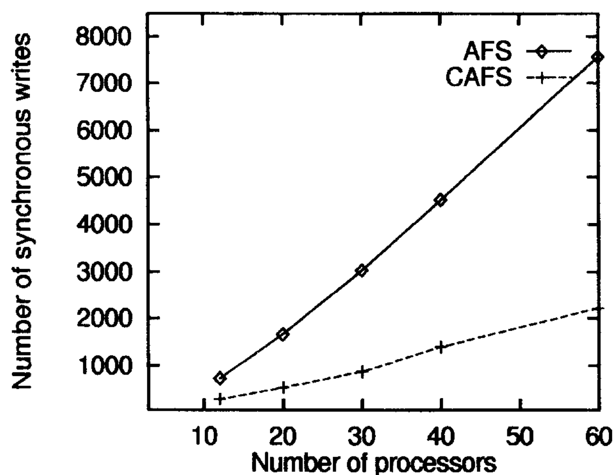


**Figure 9.** Synchronous writes to queues for adjoint convolution.

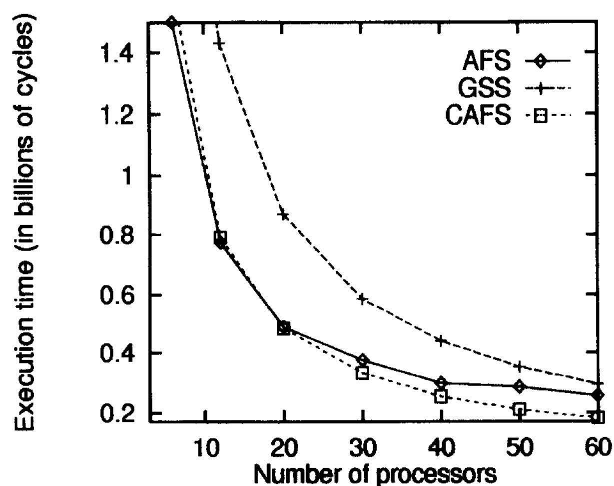


**Figure 11.** Execution times for synthetic problem with decreasing work load.

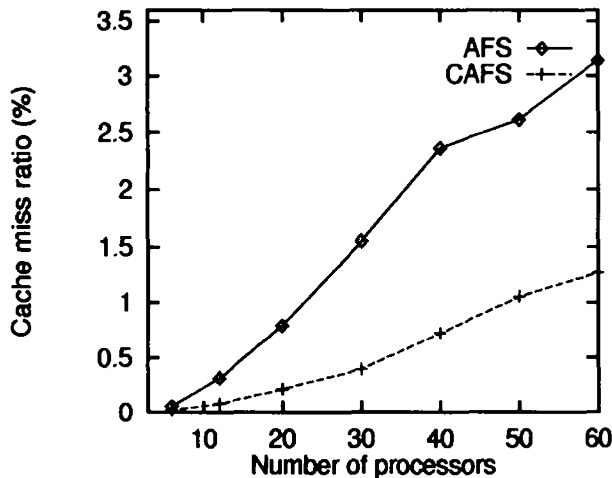


Figure 12. Cache miss ratios for synthetic problem with decreasing work load.

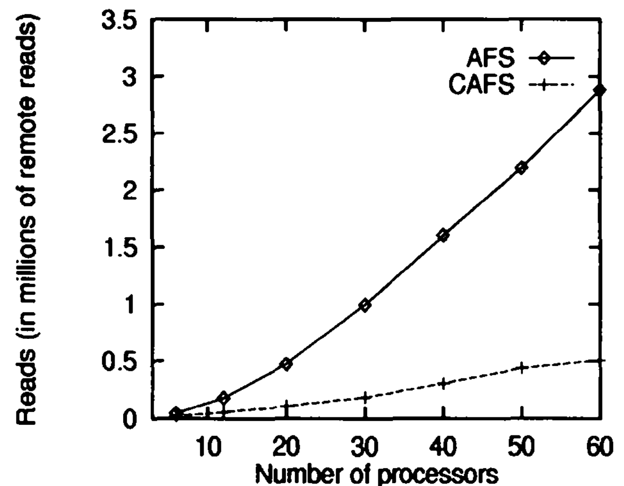


Figure 14. Remote reads to queues for synthetic problem with decreasing work load.

performs much better than AFS. Figure 12, Figure 13, and Figure 14 show the cache miss ratios, the synchronization writes, and remote reads under AFS and CAFS. As is shown in these figures, the cache miss ratios of CAFS are much lower than those of AFS, and the synchronization operations of CAFS are also lower than those of AFS.

To characterize the effect of load imbalance between the clusters, we implement a loop scheduling algorithm called CAFS with cluster migration. The initialization and execution phases of this policy are the same as those of CAFS, but migration between the clusters is performed as load imbalance occurs. As all of the local queues in a cluster are empty, the idle processor will migrate some work from the most loaded processor in the other clusters. However, migration between the clusters will waste time in accessing remote work queues and in loading data from remote memory to the cache. Figure 15 shows the execution times of CAFS and CAFS with cluster migration. The figure of various applications shows that CAFS performs better than CAFS with cluster migration except in the case of adjoint convolution. The exception, adjoint convolution, is an example of decreasing work load, but no affinity effect needs to be exploited. Therefore, we conclude that load imbalance between the clusters is slight, and that the migration between the clusters is unnecessary in most cases.

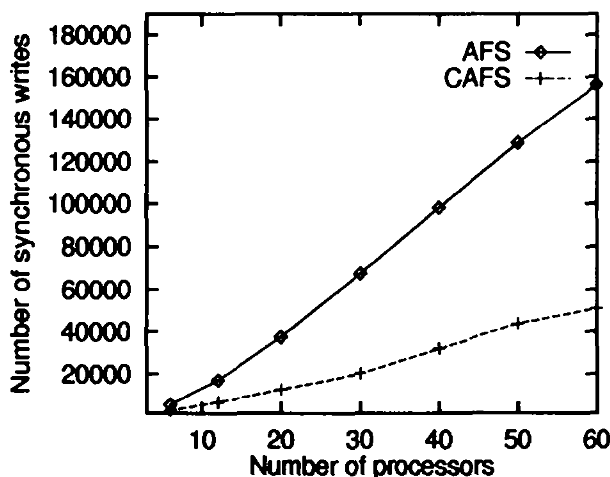


Figure 13. Synchronous writes to queues for synthetic problem with decreasing work load.

## 5. CONCLUSION

As we know, modern multiprocessors have high speed processors and relatively slow memory. In addition to load balance and synchronization overhead, affinity is an important consideration for loop scheduling algorithms [Markatos and LeBlanc, 1992a; Markatos and LeBlanc, 1994; Markatos and LeBlanc, 1992b; Crovella et al., 1991]. Algorithms based on affinity, such as AFS, indeed perform better than other dynamic algorithms such as GSS and TSS. However, AFS can not run efficiently on large NUMA machines. When imbalance occurs, it takes a lot of remote reads for AFS to search for the most loaded processor. Moreover, the migration quantum of AFS is conservative so that further migrations are needed. These migrations result in heavy synchronous operations to the queues and the waste of time of loading data into cache.

J. SYSTEMS SOFTWARE 1997; 39:61-70


| Number of processors | 8 | 10 | 12 | 16 | 20 | 30 |
|---|---|---|---|---|---|---|
| CAFS | 167.9 | 138.6 | 118.9 | 94.77 | 81.05 | 63.42 |
| CAFS with cluster migration | 168.1 | 139.1 | 119.4 | 95.68 | 82.00 | 64.84 |

**Gaussian Elimination Problem (in millions of cycles)**

| Number of processors | 6 | 10 | 12 | 15 | 20 | 30 |
|---|---|---|---|---|---|---|
| CAFS | 13.3 | 8.12 | 6.82 | 5.52 | 4.23 | 2.94 |
| CAFS with cluster migration | 13.3 | 8.12 | 6.82 | 5.53 | 4.24 | 2.95 |

**All-pair shortest paths problem (in billions of cycles)**

| Number of processors | 12 | 20 | 30 | 40 | 60 |
|---|---|---|---|---|---|
| CAFS | 350 | 216 | 148 | 115 | 79.9 |
| CAFS with cluster migration | 339 | 208 | 143 | 112 | 78.6 |

**Adjoint convolution problem (in billions of cycles)**

| Number of processors | 12 | 20 | 30 | 40 | 50 | 60 |
|---|---|---|---|---|---|---|
| CAFS | .791 | .483 | .333 | .255 | .211 | .182 |
| CAFS with cluster migration | .791 | .483 | .333 | .255 | .211 | .184 |

**Synthetic problem with decreasing work load (in billions of cycles)**

**Figure 15.** Execution times for CAFS and CAFS with cluster migration.

In this paper, we propose a new affinity algorithm, called clustered affinity scheduling algorithm (CAFS). CAFS distributes a large number of processors into several clusters. When migrating iterations, the idle processor migrates iterations from its own cluster. In addition to retaining the advantage of AFS, the new method reduces the synchronization operations, increases the cache hit ratios, and well balances the work load. We confirm our idea by running various real and synthetic applications on realistic NUMA simulator. Our results show that CAFS is a better choice among loop scheduling algorithms under large NUMA machines.

Since the migration policy plays an important role in loop scheduling algorithms, it is an interesting topic to use profile data to improve the performance of loop scheduling algorithm under large NUMA machines.

## REFERENCES


Agarwal A., et al., The MIT Alewife Machine: Architecture and Performance. *Proceedings of the 22nd International Symposium on Computer Architecture* (June 1995).

Bianchini, R., Crovella, M. E., Kontothanassis L., and LeBlanc, T. J., Software Interleaving. *Proceedings of the 1994 Symposium on Parallel and Distributed Processing*, 56–65 (October 1994).

Crovella, M., Das, P., Dubnicki, C., Markatos, E. P., and LeBlanc, T. J., Multiprogramming on Multiprocessors. *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, 590–597 (December 1991).

Hennessy, J. L. and Patterson, D. A., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers Inc., 1990.

Hummel, S. F., Schonberg, E., and Flynn, L. E., Factoring: A Practical and Robust Method for Scheduling Parallel Loops. *Communications of the ACM* Vol. 35 No. 8, 90–101 (August 1992).

Kruskal, C. P. and Weiss, A. Allocating Independent Subtasks on Parallel Processors. *IEEE Transactions on Software Engineering* Vol. SE-11 No. 10, 1001–1016 (Oct. 1985).

Lenoski et al., The Dash Prototype: Implementation and Performance, *The 19th Annual International Symposium on Computer Architecture*, 92–103 (May 1992).

Markatos, E. P. and LeBlanc, T. J., Using Memory (or Cache) Affinity in Loop Scheduling on Shared-Memory Multiprocessors, Technical Report 410, University of Rochester, Computer Science Department, 1992.

Markatos E. P. and LeBlanc, T. J., Shared-Memory Multiprocessors Trends and the Implications for Parallel Program Performance, Technical Report 420, University of Rochester, Computer Science Department, 1992.

Markatos, E. P. and LeBlanc, T. J., Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. *IEEE Trans. on Parallel and Distributed Systems* Vol. 5 No. 4, 379–400 (April 1994).

Polychronpoulos, C. D. and Kuck, D. J., Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Trans. on Computers* Vol. C-36 No. 12, 1425–1439 (December 1987).

Squillante, M. S. and Lazowska, E. D., Using Processor Cache Affinity Information in Shared Memory Multiprocessor Scheduling. *IEEE Trans. on Parallel and Distributed Systems* Vol. 4 No. 2, 131–143 (February 1993).

Subramaniam, S. and Eager, D. L., Affinity Scheduling of Unbalanced Workloads. *Supercomputing'94*, 214–226 (November 1994).

Tzen, T. H. and Ni, L. M., Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers. *IEEE Trans. on Parallel and Distributed Systems* Vol. 4 No. 1, 87–98 (January 1993).

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

Vranesic, Z. G., et al., Hector: A Hierarchically Structured Shared-Memory Multiprocessor. *IEEE Computer* 24-1, 72–80 (January 1991).