Knowledge-Based Adaptive Self-Scheduling

Yizhuo Wang, Weixing Ji, Feng Shi, Qi Zuo, Ning Deng

Knowledge-based Adaptive Self-Scheduling[*]

Yizhuo Wang, Weixing Ji, Feng Shi, Qi Zuo and Ning Deng

School of Computer Science and Technology, Beijing Institute of Technology

{frankwyz,jwx,sfbit,zql127,dunning}@bit.edu.cn

Abstract. Loop scheduling scheme plays a critical role in the efficient execution of programs, especially loop dominated applications. This paper presents KASS, a knowledge-based adaptive loop scheduling scheme. KASS consists of two phases: static partitioning and dynamic scheduling. To balance the workload, the knowledge of loop features and the capabilities of processors are both taken into account using a heuristic approach in static partitioning phase. In dynamic scheduling phase, an adaptive self-scheduling algorithm is applied, in which two tuning parameters are set to control chunk sizes, aiming at load balancing and minimizing synchronization overhead. In addition, we extend KASS to apply on loop nests and adjust the chunk sizes at runtime. The experimental results show that KASS performs 4.8% to 16.9% better than the existing self-scheduling schemes, and up to 21% better than the affinity scheduling scheme.

Keywords: loop scheduling; self-scheduling; multiprocessor system

1 Introduction

Loops are the dominant source of parallelism for many applications [1]. In general, loops fall into two categories: DOALL loops and non-DOALL loops. DOALL loops, also known as parallel loops, do have not any loop-carried dependence. Hence, different iterations of a DOALL loop can be easily executed on concurrent threads. In this paper, we focus on DOALL loops to exploit loop level parallelism (LLP). The main problem encountered with LLP is on partitioning and allocating loop iterations among threads in a multiprocessor environment. Loop scheduling schemes have been extensively studied to deal with this problem.

Existing loop scheduling schemes can be classified as static and dynamic. Static scheduling schemes partition loop iterations at compile time and statically assign iterations to processors. The static scheduling scheme results in a low runtime scheduling overhead but poor load balancing. Dynamic scheduling determines the division of iterations among processors at runtime. It leads to runtime overhead, but achieves dynamic load balancing.

Self-scheduling schemes are the most successful and widely used dynamic scheduling schemes. They partition loop iterations into chunks before execution. A chunk is

[*] This work was supported by the National Natural Science Foundation of China (60973010).

assigned to an idle processor at each scheduling step at runtime. Thus, the time spent in determining how many iterations should be scheduled is saved. Most self-scheduling algorithms have decreasing size chunks, which is a result of the trade-off between load balancing and scheduling overhead.

Some features of the loop and runtime environment affect the execution time of chunks. On one hand, different loops have different types of workload distribution. Non-uniform workload results in different execution times for same size chunks. On the other hand, processor speed and usage are important runtime features that determine the execution time of a chunk. Existing self-scheduling schemes are oblivious to these features. In this paper, we propose an adaptive self-scheduling scheme that utilizes the knowledge of the workload and the runtime environment with the aim of reducing synchronization costs, improving load balancing, and exploiting locality.

The rest of the paper is organized as follows. Section 2 reviews loop scheduling strategies and discusses related work. Section 3 presents the knowledge-based adaptive self-scheduling scheme in detail. Experimental results are presented in section 4, and the conclusion is presented in Section 5.

2 Background and Related Work

The simplest of all scheduling algorithms is static scheduling, which assigns an even number of loop iterations to each processor. It keeps the scheduling overhead to a minimum, but does not balance the load dynamically compared with the other schemes. The first self-scheduling scheme [2] is another extreme. It assigns one iteration to an idle processor each time. Hence, processors finish at nearly the same time and the workload is well balanced. However, the scheduling overhead maybe unacceptable because of the large number of scheduling steps. Self-scheduling with fixed size chunks [3] is a tradeoff between these two rival techniques. It assigns a chunk, which consists of successive iterations, to an idle processor each time. This procedure reduces the number of scheduling steps needed, which therefore, reduces the scheduling overhead.

Some self-scheduling schemes with decreasing chunk sizes were proposed to achieve better load balancing than fixed size chunking self-scheduling. They schedule large chunks at the beginning to ensure locality and reduce overhead, while scheduling small chunks at the end to balance the workload. In these self-scheduling schemes, guided self-scheduling (GSS, [1]), factoring self-scheduling (FSS, [4]), and trapezoid self-scheduling (TSS, [5]) are the most successful and widely used. The difference among them is the computation process for chunk sizes. None of the three schemes takes the characters of the loop and runtime factors into account. Our self-scheduling scheme quantifies these factors and uses them in calculating chunk sizes.

Some dynamic self-scheduling schemes [6, 7] adjust chunk sizes at run time or exploit processor affinity when mapping chunks to processors. Affinity scheduling proposed by Markatos et al. [8], employs per-processor work queues, which is similar with our scheme. However, the size of each work queue is not determined based on the knowledge of loop and runtime environment in affinity scheduling. Some groups [9, 10] have undertaken self-scheduling studies on particular architectures, considering the features of the system architecture. Our technique could be easily extended to these architectures.

3 Knowledge-based Adaptive Self-Scheduling

In this section, we describe the details of the knowledge-based adaptive self-scheduling scheme (KASS). The problem to be considered is the scheduling, across p processors P_1, P_2. . . , and P_p, of N iterations on a parallel loop. KASS has two phases:

Static partitioning phase: A knowledge-based approach is used to partition iterations of the parallel loop into local queues of processors, which makes the total workload, not the number of iterations, equally distributed onto processors approximately.

Dynamic scheduling phase: Based on self-scheduling rule, every local work queue is partitioned into chunks with decreasing sizes. Each processor allocates a chunk from its local queue to execute. A processor steals a chunk from another processor to execute when it finishes the execution of all the chunks in its local queue.

3.1 Static Partitioning

The execution time of each iteration can be obtained via loop profiling. Let t_i denote the execution time of ith iteration. The mean execution time is μ_t and the variance is σ_t. Let l_j and u_j denote the lower and upper bounds of the local queue assigned to processor j. Assuming that all the processors have the same capacity to execute the loop, static partitioning will only relate to the workload distribution of the loop. Thus, the bounds $\{(l_j, u_j) \mid j = 1, 2, ..., p\}$ can be calculated by

$$\sum_{i=l_j}^{u_j} t_i \approx \frac{N\mu_t}{p} \qquad (1)$$

Note that

$$l_1 = 1; \quad l_{j+1} = u_j + 1, \quad j = 1, 2, ..., p-1; \quad u_p = N. \qquad (2)$$

Parallel loop normally has regular workloads. Thus, t_i could be well estimated by profiling [11]. In addition, prior knowledge about the loop only contributes to the static load balancing. The dynamic scheduling will further balance the workload in our scheme. Thus, prior information need not be accurate. For example, if the workload distribution of the loop is almost uniform, no profiling is needed, and t_i (i=1, 2…, N) could just be set to any fixed number.

In addition to loop features, differences in processor speed, load running, and architecture can significantly impact performance. We use a_i to represent the capacity of processor P_i to execute the loop. In a simple case, let s_i denote the speed of P_i and b_i denote how much of P_i's capacity is used in the execution of this application. Then, $a_i = s_i b_i$. We normalize a_i with a_1. For instance, a_1=1 and a_2=2, which means that the execution time of the same workload on P_1 is twice as that on P_2. Subsequently, the loop bounds of the local queues can be calculated using the following equations under the assumption that the loop is uniform.

$$l_1 = 1;\ \ l_{j+1} = u_j + 1;\ \ u_j = \left\lceil \left(\sum_{i=1}^{j} a_i \middle/ \sum_{i=1}^{p} a_i \right) N \right\rceil, j = 1, 2, ..., p-1;\ \ u_p = N. \quad (3)$$

Considering the aspects of loop workload distribution and processor capacity, the bounds should be determined using

$$\frac{1}{a_j}\sum_{i=l_j}^{u_j} t_i \approx \sum_{j=1}^{p}\left(\frac{1}{a_j}\sum_{i=l_j}^{u_j} t_i\right)\Bigg/ p \quad (4)$$

where the discrete function of t_i is approximately equally partitioned among p processors. Unfortunately, the approximation of (4) does not provide a simple computation. We propose a heuristic method to calculate the loop bounds $\{(l_j, u_j) \mid j = 1, 2, ..., p\}$. Let l'_j and u'_j denote the bounds calculated using equation (1). Let l''_j and u''_j denote the bounds calculated using equation (3). We initialize the bounds as follows:

$$l_1 = 1;\ \ l_{j+1} = u_j + 1;\ \ u_j = \left\lfloor \frac{u'_j + u''_j}{2} \right\rfloor, j = 1, 2, ..., p-1.\ \ u_p = N; \quad (5)$$

Consequently, the execution time of iterations from l_j to u_j on processor P_j is calculated as follows:

$$T_j = \frac{1}{a_j}\sum_{i=l_j}^{u_j} t_i, j = 1, 2, ..., p. \quad (6)$$

The mean of T_j is μ_T and the variance is σ_T. Our goal is to make all T_j as equal as possible, i.e., all the processors finish the execution at approximately the same time. For a processor P_j, if T_j is greater than μ_T, the processor has much workload, thus, we need to decrease the number of iterations assigned to P_j that results in the adjustment of the bounds, and vice versa. The change is defined as $(\mu_T - T_j)/\bar{t}$, where $\bar{t}$ is the mean execution time of iterations, which is defined as $\bar{t} = \sum_{j=1}^{p} T_j \Big/ N$.

The above procedure is repeated to adjust the bounds until one of the following conditions is met:

1) The maximum number of steps, which is user inputted, is reached.
2) The coefficient of variation (c.o.v.) of T_j (σ_T/μ_T) becomes less than a threshold value, which is set to 0.1 in our experiments.
3) The variance (σ_T) in the current step is greater than that in the last step.

To summarize, we determine the loop bounds of the local queues in the static partitioning phase using the following rules:

1) If the c.o.v of the execution time of iterations (σ_t / μ_t) is less than 0.1, the bounds are calculated using equation (3). Since the loop has nearly uniform workload distribution in this case, we just take the capacities of the processors into account.
2) If the c.o.v of the capacities of the processors (σ_a / μ_a) is less than 0.1, the bounds are calculated using equation (1). The processors have almost same capacity in this case, thus, we just take the workload distribution into account.
3) In other cases, the heuristic method introduced above is used.

3.2 Dynamic Scheduling

In the dynamic scheduling phase, each local iteration queue is partitioned into chunks. A self-scheduling algorithm is then applied to assign a chunk to an idle processor in each scheduling step. A chunk size tuning parameter k_i is set for processor P_i (i=1,2,...,p) to balance between data locality and parallelism. Each processor always removes k_i of the remaining iterations in its local queue for execution. P_i turns to help other heavily loaded processors after completing the execution of the iterations in its local queue. In our implementation, P_i allocates k_i of the remaining iterations from the next unfinished work queue for execution. Considering synchronization cost, a chunk should not be too small. Thus, another parameter α, which is the minimal number of iterations in a chunk, is identified. The sketch of KASS is presented in Fig. 1.

Algorithm 1 Knowledge-based Adaptive Self-Scheduling

```
Master:
    for ($i$ = 1; $i \le p$; $i$++)        // initial partition
        assign_iterations($i$, $l_i$, $u_i$);   // assign iterations $l_i$ to $u_i$ to processor $P_i$.
    for ($i$ = 1; $i \le p$; $i$++) {      // create workers
        create_thread($T_i$);
        Set $T_i$ running on processor $P_i$;
    }
    Wait for all worker threads to complete.
Worker:
    get_iterations($l$, $u$, $k$){   // get a chunk from a work queue.
        if ($u-l < 2\alpha$){ // ensure chunk size is larger than $\alpha$.
            chunk.begin = $l$;   chunk.end = $u$;
        }else{
            chunk.begin = $l$;   chunk.end = $l + (u-l)k$;
        }
        $l$ = chunk.end + 1; // update the lower bound.
    }
    while (true) {
        // get a chunk in the local work queue of current processor $P_i$.
        chunk = get_iterations($l_i$, $u_i$, $k_i$)
        if(chunk = NULL){
            Lock( );
            $j$ = find_unfinished_workqueue( );
            // processor $P_j$ has remaining iterations.
            if( $j$ = 0) // all the iterations have been scheduled.
            { Unlock( );    thread_exit( ); }
            chunk = get_iterations($l_j$, $u_j$, $k_i$)   // steal a chunk from $P_j$.
            Unlock( );
        }
        execute_loop(chunk);
    }
```

Fig. 1. The sketch of KASS

Next, we present the identification of tuning parameters k_i and α. Previous work in [12] shows that the chunk size should range from $R/2p$ to R/p in a self-scheduling scheme with a central work queue to have reasonable load imbalance and synchronization overhead. R is the number of the remaining iterations. The chunk size would range from 0.5 R to R when the same principle is applied to per-processor work queues.

Therefore, the range of k_i should be [0.5, 1]. The selection of k_i relates to two factors: the "*error*" of static partitioning and the dynamic changes of runtime environment. We use c.o.v (σ/μ) to represent the *error* of static partitioning. Three cases in static partition phase were noted, as described in section 3.1. We use σ_a/μ_a when $\sigma_t/\mu_t < 0.1$; use σ_t/μ_t when $\sigma_a/\mu_a < 0.1$; use σ_T/μ_T for other cases. The range of σ/μ is [0, 0.1]. Let Δ denote the dynamic changes in the runtime environment, which ranges from 0 to 0.4 to enforce k_i ranging from 0.5 to 1. The chunk size tuning parameter k_i is defined as

$$k_i = 1 - \mu / \sigma - \Delta, \qquad i = 1, 2, ..., p. \tag{7}$$

The value of Δ is user inputted. Our experiment results suggest that it is optimal to set Δ to 0.1 when the system is relatively stable. Thus, the value of k_i is 0.8.

The tuning parameter α limits the minimal size of a chunk. If the time that a processor spends to steal a chunk from another processor is longer than the execution time of this chunk, it is clear that chunk stealing need not be done. Performance penalty of work stealing in the present comes from two aspects. One is the synchronization overhead, which refers to the time cost of the critical sections in shared memory multiprocessor systems. Another is the loss of data locality, which is mainly the cache miss penalty in shared memory systems, and is considered as communication cost in distributed memory systems. The execution time of the critical section used to allocate a chunk via profiling is obtained. Let T_{cs} denote it. The minimal number of iterations of a chunk is defined as $\alpha = 2T_{cs}/\mu_t$, where, μ_t is the mean execution time via profiling. The above equation does not take the effect of locality into account because locality is hard to quantify. Hence, we enlarged the synchronization cost to fill up the loss.

3.3 KASS for Loop Nests

For loop nests with sequential outer loop and parallel inner loop, we improve the KASS algorithm by adaptively changing k_i for each processor. A counter C_i is set for processor P_i. C_i is increased by one each time the processor P_i removes a chunk from another processor P_j. Accordingly, C_j is decreased by one. The counters are initially set to zero at the beginning of every step in the outer loop. At the end of the outer loop step, the processors are classified into three types based on the value of the counters:

- Lightly loaded: the C_i value of the processor is greater than a positive integer θ, which is a threshold set by the user. We set θ to 1 in our experiments.
- Normally loaded: the C_i value of the processor is within the range of [$-\theta$, θ].
- Heavily loaded: the C_i value of the processor is less than $-\theta$.

If P_i is a heavily loaded processor, less iterations should be assigned to it at each scheduling step so that more iterations remaining in the heavily loaded processor can be executed by the lightly loaded processors. k_i is decreased to realize the adjustment. On the contrary, k_i is increased for lightly loaded processors so that these processors can finish the chunks in their local work queue as soon as possible, and then start to help heavily loaded processors. The value of k_i is adjusted to k_i' as follows:

$$k_i' = \begin{cases} \min(0.9, k_i + 0.1) & \text{if } C_i > \theta \\ k_i & \text{if } \theta \geq C_i \geq -\theta \\ \max(0.5, k_i - 0.1) & \text{if } C_i < -\theta \end{cases} \tag{8}$$

The KASS algorithm for loop nests is similar to the affinity scheduling proposed by Markatos et al. [8]. Both of these algorithms use per-processor work queues and utilize work-stealing to balance the workload. Three important differences between KASS and affinity scheduling exist. First, static partitioning in KASS is knowledge-based, whereas affinity scheduling makes equivalence partitioning. Second, chunk sizes are adaptively changed by the adjustment of k_i in KASS. In affinity scheduling, chunk sizes are fixed during multiple execution times of the inner DOALL loop. Third, KASS limits the minimal chunk size, whereas affinity scheduling does not. Excessive partitioning as in some cases in affinity scheduling causes harmful effects on data locality and overall performance.

4 Experiments

In this section, we present our experimental setup and results. KASS is compared with other popular loop scheduling schemes in two cases studies: one for outer most parallel loops; another for loop nests.

For the case study on outermost loops, we extracted several kernels from SPEC CPU2000/2006 benchmarks. The detail of the kernel set is presented in Table 1. For the case study on loop nests, the selected application kernels are Successive Over-Relaxation (SOR), Jacobi Iteration (JI), and Transitive Closure (TC) [8]. The detailed experimental setup is provided in Table 2.

4.1 Study on Outermost Parallel Loops

GSS, FSS, and TSS are widely used loop scheduling schemes in practice. For each kernel in Table 1, we compare execution times obtained with static scheduling, GSS [1], FSS [4], TSS [5], and KASS. Experiments were run by varying the number of threads from 2 to 16. All worker threads are bound on different cores.

Several artificial loads are added to processors P_1, P_3, P_5,..., and P_{15} when testing. Thus, $a_i = 1$ (i = 1, 3, ..., 15) and $a_i = 2$ (i = 2, 4, …, 16) in the static partition phase of KASS. L1 to L9 use equation (3) to calculate the bounds of local partitions. L10 has obvious non-uniform workloads. Hence, the heuristic method is applied to obtain the bounds. k_i is set to 0.8 because there are no other unknown loads running in the system. The parameter α is determined via profiling for each kernel.

We report the speedups over sequential versions of the codes for each scheduling scheme in Fig. 2. The execution time of the sequential versions is the average value of the execution times in loaded and unloaded environments on one core. We observed that for all ten kernels, both KASS and the classic self-scheduling schemes (GSS, FSS, and TSS) show significant improvement over static scheduling. Although static scheduling has better data locality than other schemes of scheduling and has no synchronization cost, poor load balancing made it much worse than self-scheduling.

Table 1. Kernel Set

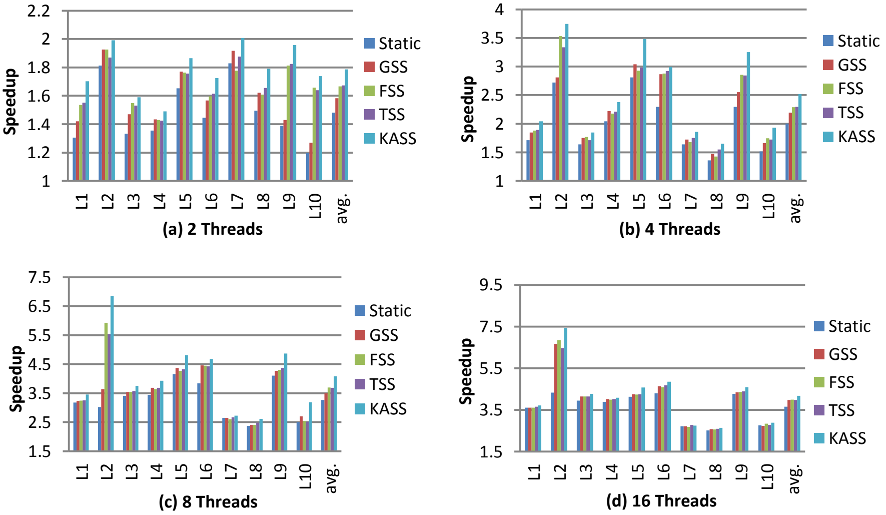
Kernel	Suite	Benchmark	File, line
L1	SPEC2000	179.art	scanner.c, 317
L2	SPEC2000	188.ammp	rectmm.c, 405
L3	SPEC2000	183.equake	quake.c, 462
L4	SPEC2006	470.lbm	lbm.c, 186
L5	SPEC2006	433.milc	quark_stuff.c, 1523
L6	SPEC2006	462.libquantum	gates.c, 89
L7	SPEC2006	464.h264ref	mv-search.c, 394
L8	SPEC2006	482.sphinx3	vector.c, 512
L9	Matrix Multiplication		mm.c
L10	Matrix Transpose		mt.c

Table 2. Experimental Setup

Processor	4 X Intel®Xeon™ X7350 (4 cores/chip) @ 2.93GHz
L1 Data Cache	32 KB
L2 Cache	2 X 4 MB
Memory	8 GB
Compiler	gcc 4.2.4
Compiler Flags	-O2 -lpthread -lrt -lm
Thread Library	NPTL 2.7
OS	Linux ubuntu 2.6.22.14

Figure 2 shows that KASS is the most effective self-scheduling scheme. Comparing KASS with GSS, FSS and TSS, we observe that KASS is 16.9% faster than GSS on average with 8 threads, which is the best case, and 4.8% faster than FSS on average with 16 threads, which is the worst case. Again, we notice that GSS is worse than other self-scheduling schemes on average due to the large chunk size allocated in the first scheduling step.

The performance gains obtained from KASS can be attributed to load balancing, synchronization overhead and data locality. KASS only needs synchronization during work stealing. In the ideal case, the number of locks can be as low as the number of threads if the workload was balanced perfectly in the static partition phase. For the other three self-scheduling schemes, the number of locks equals the number of chunks, and the value never changes during numerous executions times.



(a) 2 Threads

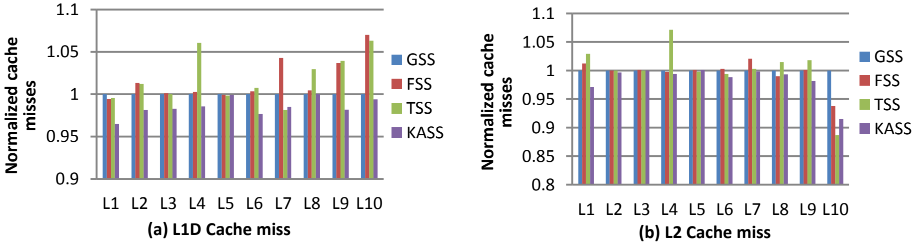
(b) 4 Threads

(c) 8 Threads

(d) 16 Threads

Fig. 2. Speedup over sequential execution

Aside from synchronization overhead, data locality is another significant benefit of using distributed work queues. For the outmost parallel loop, spatial locality is improved because most chunks in the local queue are executed successively by the local processor. For nested loops, distributed work queues enforce processor affinity with the data set. Therefore, temporal locality is improved. To gain better insight into the performance issues, we collected L1 and L2 cache misses with Pfmon2 for each kernel and each self-scheduling scheme. Figure 3 shows the results when 4 threads are used. Cache misses are normalized against cache misses in GSS. KASS has less L1 data cache misses than others. L2 cache misses decrease slightly except for L10. The matrix size in L10 is 3200 x 3200, thus, it cannot be loaded into the L2 cache entirely. Moreover, matrix transposition has decreasing workload distribution. Therefore, L10 presents much variation in cache misses.



(a) L1D Cache miss

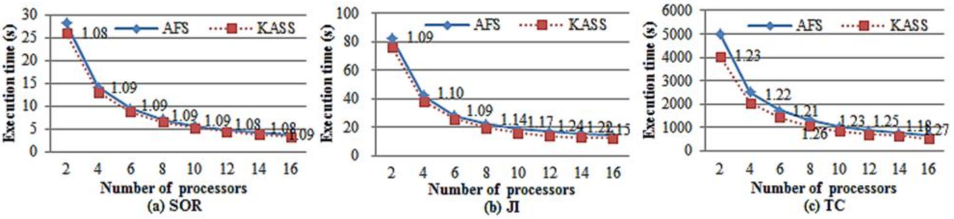
(b) L2 Cache miss

Fig. 3. Cache misses when 4 Threads used

4.2 Study on Loop Nests

To the best of our knowledge, affinity scheduling is most similar with our technique for loop nests. We implemented KASS algorithm as presented in Section 3, and the affinity scheduling algorithm (AFS) in [8]. Three applications (SOR, JI, and TC) were run on 2 cores to 16 cores with N = 10000. The average execution times are reported in Fig. 4. KASS performs better than AFS in all cases. The speedups of KASS over AFS in the figure are also labeled. The maximal speedup is 1.27, which makes KASS 21% faster. As discussed in Section 3, the attained performance is mainly attributed to knowledge-based static partition and adaptive adjustment with tuning parameters.

Both KASS and AFS exploited processor affinity with distributed work queues. Work stealing happens when load imbalance arises between initial partitions. A significant disadvantage with the AFS scheduling scheme exist where work stealing has dramatically increased scheduling overhead when more processors are used. Therefore, KASS should achieve greater speedup when more processors are used because of relatively balanced initial partition and adaptive adjustments, which result in less work stealing than AFS. This trend has been noticed in Fig. 4(b).



(a) SOR

(b) JI

(c) TC

Fig. 4. Execution time of SOR (a), JI (b) and TC (c)

5 Conclusion

A knowledge-based adaptive self-scheduling (KASS) algorithm for parallel loops has been proposed in this paper. An experimental study was performed to compare the KASS algorithm with classic self-scheduling algorithms (GSS, TSS, and FSS), static scheduling, and affinity scheduling algorithm. The major conclusions from the study are: Dynamic scheduling schemes perform well for all kernels. KASS performs better than classic self-scheduling and static scheduling for the outermost loops. For loop nests, KASS not only exploits processor affinity, but also adaptively adjusts chunk partitions. Therefore, KASS achieved better performance compared with affinity scheduling. Future work would be geared towards implementing KASS on heterogeneous computing systems.

References

1. C. D. Polychronopoulos, D. J. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Trans. Computers* 36(12):1425-1439, 1987.
2. B. J. Smith. Architecture and Application of the HEP Multiprocessor Computer System. *Real Time Signal Processing IV*, vol. 298, 1981.
3. P. Tang and P. C. Yew. Processor self-scheduling for multiple nested parallel loops. In *ICPP*, pages 528-535, 1986.
4. S. Flynn-Hummel, E. Schonberg, and L. E. Flynn. Factoring: A method for scheduling parallel loops. *Communications of the ACM* 35(8):90-101, 1992.
5. T. H. Tzen, L. M. Ni. Trapezoid self-scheduling: a practical scheduling scheme for parallel computers. *IEEE Transactions on Parallel and Distributed Systems* 4(1):87-98, 1993.
6. T. Tabirca, L. Freeman, S. Tabirca, and L. T. Yang. 2004. Feedback guided dynamic loop scheduling: convergence of the continuous case. *J. Supercomput* 30(2): 151-178, 2004.
7. R. L. Cariño, Ioana Banicescu. Dynamic load balancing with adaptive factoring methods in scientific applications. *J. Supercomput* 44(1):41-63, 2008.
8. E. P. Markatos, T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.* 5(4):379-400, 1994.
9. S. Srivastava, I. Banicescu, F. M. Ciorba: Investigating the robustness of adaptive Dynamic Loop Scheduling on heterogeneous computing systems. In *IPDPS Workshops,* pages 1-8, 2010.
10. C. Yang, C. Wu, and J. Chang. Performance-based parallel loop self-scheduling using hybrid OpenMP and MPI programming on multicore SMP clusters. *Concurrency Computation Practice and Experience* 23(8): 721-744, 2011
11. A. Kejariwal, A. Nicolau, and C. D. Polychronopoulos. History-aware Self-Scheduling. In *ICPP*, pages 185-192, Columbus, Ohio, USA, 2006.
12. J. Liu, V. A. Saletore, and T. G. Lewis. Safe Self-Scheduling: A Parallel Loop Scheduling Scheme for Shared-Memory Multiprocessors. *Int. J. Parallel Program* 22(6):589-616, 1994.