A New Carried-Dependence Self-scheduling Algorithm

Hyun Cheol Kim

Dept. of Computer & Information, Jaineung College, 122, Songlim-Dong, Dong-Gu,
Incheon 401-714, Republic of Korea
hckim@mail.jnc.ac.kr

Abstract. In this paper we present an analysis on a shared memory system of five self-scheduling algorithms running on top of the threads programming model to schedule the loop with cross-iteration dependence. Four of them are well-known: self-scheduling (SS), chunked self-scheduling (CSS), guided self-scheduling (GSS) and factoring. Because these schemes are all for loops without cross-iteration dependence, we study the modification of these schemes to schedule the loop with cross-iteration dependence. The fifth is our proposal: carried-dependence self-scheduling (CDSS). The experiments conducted in varying parameters clearly show that CDSS outperforms other modified self-scheduling approaches in a number of simulations. CDSS, modified SS, factoring, GSS and CSS are executed efficiently in order of execution time.

1 Introduction

In many scientific applications, loops are the richest source of parallelism. Therefore, many loop scheduling schemes were proposed to exploit parallelism. In general, there are two major types of parallel constructs that are provided in all parallel languages: *Doall loops* (also called a parallel loop) and *Doacross loops* [1],[2][3]. However, most previous work for loop scheduling focused on *Doall loops* without cross-iteration dependence.

The dependence constraint among different iterations, called cross-iteration dependence, is our major concern. A cross-iteration dependence occurs if some data computed in one iteration is also used by another iteration. Data dependence analysis gives information about underlying data flow of a loop. To preserve those cross-iteration dependences, additional a code must be added to ensure proper synchronization between the processors executing different iterations [1],[2],[3].

This paper proposes a new self-scheduling method for parallel processing of a loop with cross-iteration dependence on shared memory systems. Also, we study the modification of several self-scheduling schemes using central queue in order to schedule the loop with cross-iteration dependence. Our scheme assigns loops efficiently in three-level considering the dependence distance of the loops. To adapt the proposed scheduling and modified self-scheduling schemes into various platforms, including a uni-processor system, we use Java thread for implementation and performance evaluation of five scheduling methods. A series of simulation results corresponding to various parameter changes are presented in this paper.

This paper is organized into the following sections. Section 2 revisits some well known loop scheduling schemes for shared memory multiprocessors. Then, carried-

O. Gervasi et al. (Eds.): ICCSA 2005, LNCS 3480, pp. 281 – 290, 2005.
© Springer-Verlag Berlin Heidelberg 2005

dependence self-scheduling (*CDSS*) scheme is introduced in Section 3 with an explanation of its working principles. Next, discussions on the their implementation and simulation results are presented in Section 4, and followed by a conclusion in Section 5.

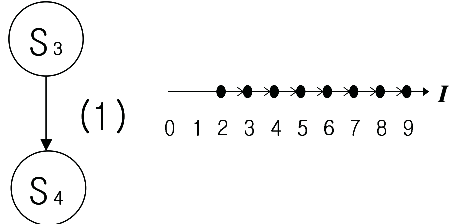
2 Related Works

In this section we look at some of the dynamic loop scheduling algorithms, which have been proposed in the literature. These algorithms fall into two distinct classes: *central queue based algorithms* and *distributed queue based ones* according to the organization of take queue. According to the presence of a dedicated scheduler, algorithms can be classified the *central* and *self-scheduling method*.

A special case of scheduling through distributed control units is *self-scheduling* [4],[5],[6]. As implied by the term, there is no single control unit that makes global decisions for allocating processors, but rather the processors themselves are responsible for determining what task to execute next. In central queue based self-scheduling algorithms, such as *self-scheduling* (*SS*), *guided self-scheduling* (*GSS*), *factoring* and *chunked self-scheduling* (*CSS*), iterations of a parallel loop are all stored in a shared central queue and each processor exclusively seizes some iterations from the central queue to execution. The major advantage of using a central queue is the possibility of optimally balancing the load. While keeping a good load balance, the central queue based algorithms differ in the way they reduce synchronization and loop allocation overheads [4],[5],[6]. In *SS* [4],[5] algorithms, each processor repeatedly executes iterations of the loop until all iterations are executed. *SS* achieves almost perfect load balancing. Unfortunately, this method incurs significant synchronization overhead. *CSS* [5],[6] reduces synchronization overhead by having each processor take k iterations instead of one, resulting in less synchronization overhead but load balancing is not as efficient as *SS*. *GSS* [7] changes the size of chunks at run-time. By allocating large chunks of iterations at the beginning of loops to processor, synchronization overhead can be reduced. In addition, allocating small chunks at the end of the loops gives rise to workload balance. Under *GSS* method, each processor is allocated to l/p iterations, where p is the number of processors and l is the number of remaining iterations. In *factoring* [8] algorithm, allocation of loop iterations to processors proceeds in phases. During each phase, only a subset of the remaining loop iterations is divided equally to processors. It balances workload better than *GSS* when the computation times of loop iterations are considerable. Other schemes in self scheduling models are *adaptive guided self-scheduling* [9], which includes a random back-off to avoid contention for the task queue, and assigns iterations in and interleaved fashion to avoid imbalance; *trapezoidal self-scheduling* [10], which linearly decreases the number of iterations allocated to each processor; *tapering* [11], which is suitable for irregular loops and uses execution profiles to select a chunk size that minimizes the load imbalance; *safe self-scheduling* [12], which uses a static phase where each processor is allocated a chunk of iterations and a dynamic phase during which the processors are self scheduled to even out the load imbalances; and *affinity scheduling* [13],[14][15], which takes processor affinity into account while scheduling. Under this scheme, all the processors are initially assigned an equal chunk taking into account data reuse and locality. Previous approaches to loop scheduling have been attempted to achieve the

minimum completion time by distributing the workload as even as possible and to minimize required synchronization overhead. However, all these previous works focused on loops without cross-iteration dependence. In Section 3, we explain our scheduling scheme for loops with cross-iteration dependence.

```
  Do I=2 to 9
S1 A[I]=I*10
S2 B[I]=A[I]+2
S3 C[I]=A[I]*B[I]
S4 D[I]=C[I-1]+100
S5 E[I]=(D[I]+C[I])*B[I]
End Do
```



(a) An Example Program (b) DDG (c) Iteration Space Dependence Graph

Fig. 1. An Example Program and its Dependence Graph

3 The Proposed Algorithm

An example program of the loop with cross-iteration is Figure 1(a). Here, the value assigned to C in each iteration of S_3 is fetched by S_4 in the next iteration of the loop. Since some instances of S_4 depend on some instances of S_3 , we write $S_3\delta^f S_4$. There are many instances of each statement in the loop, but the *data dependence graph* (DDG) [1],[2],[3] contains only one node for each statement, as shown in Figure 1(b). This is a loop-carried, lexically forward flow dependence relation. The (1) annotation in Figure 1(b) represents the dependence distance of the loop. The dependence relations is represented by *iteration space dependence graph* [1][2][3], which contains one point for each iteration of the loop and the dependence is represented by an edge from the source iteration to the target iteration as shown in Figure 1(c). Also, the dependence relation between iterations is represented by loop dependence graph (LDG) [1],[2],[3].

Figure 2 illustrates the proposed carried-dependence self-scheduling (*CDSS*) algorithm where, *d* indicates dependence distance of the loops. The basic idea of *CDSS* is that it distributes the loop iterations into processors in three-level considering the dependence distance of the loops that have dependences between iterations while minimizing the loop allocation overhead (i.e. synchronization overhead to access exclusive shared variables). Also, *CDSS* uses a central task queue for scheduling of the loops. The data structures used in the algorithm are as follows. The central task queue, *TaskQueue*, has the role of identifier for the loops and *CrossDep* is an array that has *n* bits, each bit related dependency information corresponding to iterations of loops. It determines the execution of iteration j, which is dependent on iteration. If the value of a corresponding bit equals one, then the iteration can be executed. The initial values of the element of *CrossDep* array are set at 1 when the iterations have no incoming dependency arcs. Therefore, an iteration that has a value 1 of *CrossDep* would be executed

immediately because it is an independent iteration. The remaining bits of *CrossDep* are set at 0. The shared variables, *CrossDep* and central task queue, which correspond to critical sections, are serialized by locking. The local variable Temp is used to reduce the access count to shared variables and the variable Rest represents the number of remaining iterations currently.

```
Entry Block :
/* get_routine */
1.  lock(TaskQueue)
2.    Temp =TaskQueue
3.  unlock(TaskQueue)
4.  if (Temp[front] == 1) then
/* get an iteration from front of queue */
5.    i = get_from_queue(1)
6.    Execute Block(i)
7.  else if (Rest > d-1)
/* assignment phase of middle of loop */
8.    start_chunk = get_from_queue(d)
9.  else
10.   start_chunk = get_from_queue(Rest)
/* assignment phase of end of loop */
11. end if
12. update Rest
/* exec_test_routine */
13. for k=0, d-1
14.   i = start_chunk + k
15.   lock(CrossDep)
16.      Temp = CrossDep
17.   unlock(CrossDep)
Exec_test :
18.   if(Temp[i] == 1)
19.      Execute Block(i)
20.   else then
21.      waiting for system_defined interval
22.      reload Temp from CrossDep
23.      Exec_test
24.   end if
25. end for
Execute Block(i) :
26.   execute the code of iteration i
Exit Block :
27.   j = detect(i)    /* j = i+d */
28.   lock(CrossDep)
29.     CrossDep =Fetch_Set(j, 1)
30.   unlock(CrossDep)
```

Fig. 2. Pseudo Code for Proposed CDSS Scheme

The proposed scheduling method is divided into three blocks according to functionality. *Entry Block* (lines 1-25) is inserted into processors to schedule the loops. In *get_routine*, the processor obtains the iterations from the front of the *TaskQueue* for

execution. Here, the determination of the number of iterations is related to scheduling overhead. The scheduling police in *get_routine* is divided into roughly three phases. First, the processor obtains an iteration from the front of the queue only once (lines 4-6). Next, each idle processor obtains as much dependence distance as possible until the number of remaining iterations has less than d (lines 7-8). Finally, all remaining iterations are fetched by the processor (lines 9-10). In *exec_test_routine*, the processor tests whether the fetched iteration is ready for execution. The processor executes the iteration in *Execute Block*(line 26) when the iteration has satisfied dependence relations. If the dependent successor of the iteration is not finished, the processor will wait for execution of the iteration. When the iteration i is terminated, the value of the jth element of *CrossDep*, which has dependence information of the iteration depends on L_i, is set at 1 in *Exit Block* (lines 27-30). It synchronizes that iteration j is ready for execution. In order to schedule the loop with cross-iteration dependence using the existing self-scheduling method based on central queue, we have to modify them. First, the analyzing piece of the processor that tests the execution condition of the iterations must be inserted (lines 13-25). According to this operation, the iteration is running or waiting. Also, the dependence distance d in line 13 is changed according to different chunk sizes of each scheduling method. After the iteration executes, the routine to synchronize the dependence relations between iterations is also necessary (lines 27-30).

4 Implementation and Performance Evaluation

To adapt the proposed scheme and modified self-scheduling methods onto various platforms, including a uni-processor system, we use threads to perform processor activity for our implementation. Although only *SS*, *CSS*, *GSS* and *factoring* are multithreaded for our simulation study, *CDSS* is a general technique, which may be applied onto most self-scheduling schemes with reasonable chunk size. Other central queue based self-scheduling methods are all possible to be multithreaded to schedule the loop with iterations. We implanted five algorithms with the same form in thread level using a JDK1.2.2 programming environment and executed these methods with a Pentium–III 450MHz with 128Mb main memory (Redhat Linux 6.1).

In our simulated experiments, we compared the performance of the *CDSS* scheme with the modified four self-scheduling methods using central task queue in terms of overall execution time including scheduling overhead. For synchronization of critical sections, we use *synchronized* keywords and synchronization methods such as *wait* and *notifyAll* for implementation of locking [16]. The system parameter values in our experimentation are as follows: the number of threads (t) is 1 to 30 chosen randomly, the number of processors (p) is one in our simulation because our experimental tasks are fine grain and the scheduling operation time is relatively less than the execution time of a node. The application parameter including the number of iterations (n) is 60 to 180. The processing time of each iteration (e) was chosen to be 20, 70, 120, 170 and 220 miliseconds, respectively. And the dependence distance (d) of loops ranged from 2 to 4. Table 1 shows the overall execution times for varying parameter values in modified *SS* (*MSS*), modified *CSS* (*MCSS*), modified *GSS* (*MGSS*), modified *factoring* (*MFact*) and *CDSS*. As shown in the table, the proposed algorithm usually outperforms other modi-

fied self-scheduling algorithms in terms of total execution time including the scheduling costs and synchronization overheads. We analyzed the execution times effect on the system and application parameters.

Table 1. Execution Time for Scheduling Algorithm (msec)

t	*d*	*n*	*e*	Scheduling Algorithm				
				MGSS	MFact	MSS	MCSS	CDSS
2	2	60	20	1627	1537.2	888.8	1726.2	888.8
			120	7040.4	6656.8	3844.4	7540.6	3839.6
	3	120	170	19726.6	18284.4	10765.4	21158.4	10754.8
			220	25618	23390.6	13756.8	27052.4	13749
	4	180	20	4819.2	4508.2	2660.2	5202.6	2636.4
			120	20982.8	19624.6	11514.6	22606.6	11476.8
3	2	120	70	8792.2	8416.4	4775	9349.2	4768.4
			120	14199	13562.8	7742.4	15056.8	7714.8
	3	60	20	1386.6	1170.4	586.4	1646.4	588.8
			70	3727.8	3169	1588.6	4440.6	1584.8
	4	180	170	27668.8	25302.4	10759.8	31212.6	10764.8
			220	35322.8	32366.2	13766.2	39904.6	13754.8
4	2	60	70	4054.2	3734.4	2384	4522.8	2380.4
			170	9153.2	8434.6	5392	10225.6	5388.6
	3	120	20	2883.2	2665	1180.2	3371.4	1176.6
			120	12539	11543.4	5157.8	14628.2	5154
	4	180	20	4356.4	3976.2	1323.4	5052.8	1338.6
			220	33948.4	30974	10306.8	39220.8	10329.2
10	2	180	170	27252.2	25282.4	16174.6	30688.6	16172.2
			220	34845.8	32320.8	20680.2	39229.8	20669
	3	120	20	2281.8	1876.2	1179.8	2959.6	1170
			120	10044.6	8216.4	5159.4	13085.2	5144.4
	4	180	120	14763	12042.2	5834.8	19717	5800
			170	20463.8	16702.8	8080.4	27440	8050.6
20	2	120	20	2416.6	2110	1788.8	2972.8	1762.6
			70	6522.8	5653.6	4793.4	8025.2	4756.4
	3	60	20	698.6	765.2	592.8	652	592.6
			170	4301.2	4661.6	3588.2	3940.6	3576.8
	4	180	120	10900.2	8184.2	5845.4	15901.2	5835.8
			220	19287.2	14469.8	10341.2	28213.6	10340
30	2	180	20	3666.2	3173.8	2695.2	4495.8	2666.6
			70	9814.2	8471.6	7195.2	12034	7159.6
	3	120	170	9403.8	9000.4	7194.6	11117.6	7196.8
			220	11986	11487.8	9195.6	14223.8	9187.6
	4	60	120	2080.8	1951	1951.8	2076.4	1949.6
			170	2874	2699.8	2703	2880.8	2693.4

4.1 Effect of System Parameter

For parallel processing using threads, we created threads up to 30 on the same applications. Figure 3 shows the overall execution time of each scheduling scheme by varying the number of threads with various parameter values. Generally, it is clear

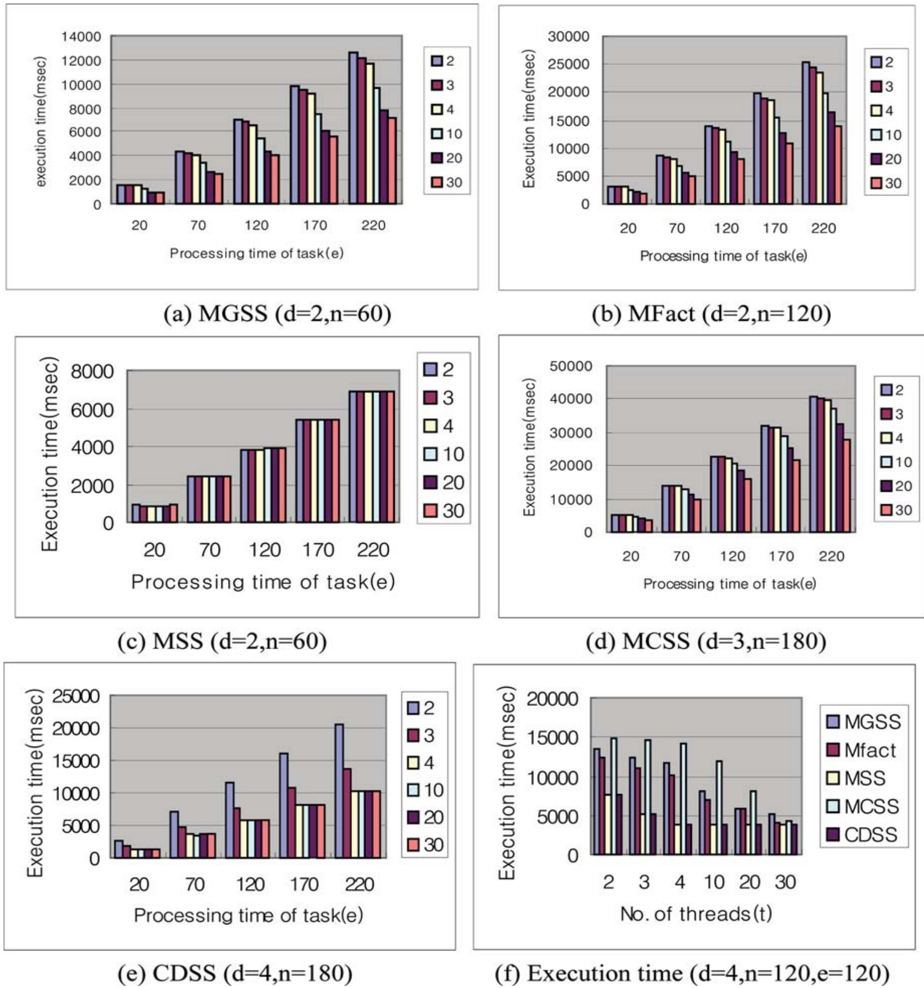


Fig. 3. Effect of the Number of Threads

that providing more threads can improve the performance substantially as shown in the Figure 3. Unfortunately, there are some cases in which some scheduling policies occasionally have poor performance when increasing the threads. In Figure 3(c), it shows performance degradation over a lot of threads when 30 threads were used. In *MGSS* (Fig.3(a)), *Mfact* (Fig.3(b)) and *MCSS* (Fig.3(d)), we have good performance

by providing substantially more threads. However, *MSS* and *CDSS* show that a reasonable number of threads used to improve the performance depends on the dependence distance of the loops. Thus, we achieved the best performance using a few threads which equal the dependence distance *d* in most applications. We can observe that to get a reasonable number of threads for improved performance is very difficult but it could be achieved by repeating experiments on a dedicated scheduling scheme and application. Next, we analyze the effect of the application parameters.

4.2 Effect of Application Parameters

The application parameters also affected the execution times as shown in Figures 4, 5 and 6. By increasing the number of iterations (*n*), the execution time is long on the same number of threads. For example, as shown in Figure 4, the proposed algorithm has 1201, 2373 and 3577ms execution times on the variations of *n* with 60, 120 and 180, respectively, when *d*=4, *e*=70 and *t*=30. As a result, by doubling the number of iterations, we get about double the execution time. Next, we changed the processing time of task (*e*). Figure 5 shows the execution time on the variations of task sizes when *d*=3, *n*=120 and *t*=10. For example, when the execution costs per iteration varied from 20, 70, 120, 170 and 220ms, it had the execution times of 1170, 3173, 5144, 7173 and 9160ms, respectively, using the proposed algorithm. By increasing the processing time of the task, it has long execution times. Finally, Figure 6 shows the execution

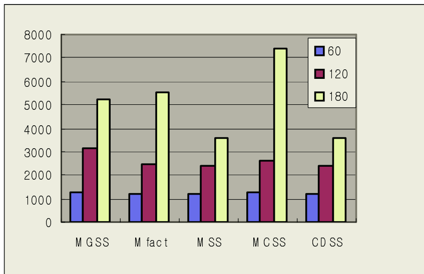


Fig. 4. Effect of n (d=4, e=70, t=30)

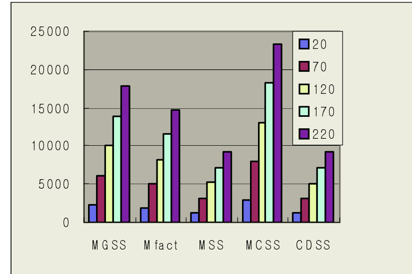


Fig. 5. Effect of e (d=3, n=120, t=10)

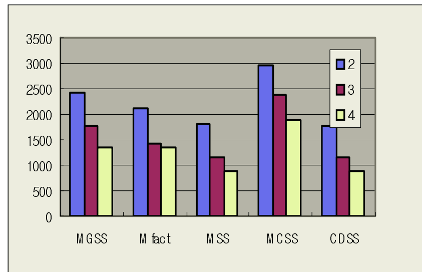


Fig. 6. Effect of d (n=120, e=20, t=20)

time on the effect of varying dependence distance (*d*) of loops with *n*=120, *e*=20 and *t*=20. In this experiment, the application with a large dependence distance can reduce the execution times by decreasing the overhead. For example, according to the variation of *d* with 2, 3 and 4, we can reduce the execution times to 1763, 1166 and 891ms, respectively, using *CDSS*. This is because the applications with large distances have few synchronization points for parallel execution and thus increase the thread parallelism. In various experimental environments, *CDSS* shows improved performance over *MSS*, *MFact*, *MGSS* and *MCSS* by about 0.02, 40.5, 46.1 and 53.6%, respectively, in our experimental parameter values.

5 Conclusions

We proposed a new scheduling method for efficiently execution of a loop with cross-iteration dependence on a shared memory multiprocessor. The proposed method is a self-scheduling algorithm and assigns the loops in three-level considering the synchronization point according to the dependence distance of the loops. Also, we studied the modification that converts the existing self-scheduling method based on the central task queue for parallel loops onto the same form applied to loops with cross-iteration dependence. To adapt the proposed and modified methods onto on various platforms, including a uni-processor system, we use thread for implementation. Compared to other assignment algorithms with various changes of application and system parameters, *CDSS* is found to be more efficient than other methods in overall execution time including scheduling overhead. With our new loop scheduling technique, the execution time of our experimental applications can be improved by 0.02%~53.6 compared to the modified methods.

References

1. Wolfe, M.: High Performance Compilers for Parallel Computing. Addison-Wesley (1996)
2. Quinn, M.J.: Parallel Computing -Theory and Practice. McGraw-Hill (1994)
3. Zima, H., Chapman, B.: Super Compiler for Parallel and Vector Computers. Addison-Wesley (1991)
4. Tang, P., Yew, P.C.: Processor Self-Scheduling for multiple nested parallel loops. Proc. 1986 Int. Conf. Parallel Processing (1986) 528-535
5. Fang, Z., Tang, P., Yew, P.C., Zhu, C.Q.: Dynamic Processor Self-Scheduling for General Parallel Nested Loops. IEEE Trans. on Computers, vol. 39, no. 7 (1990) 919-929
6. Kruskaland, C.P., Weiss, A.: Allocating independent subtasks on parallel processors. IEEE Trans. Software Eng., vol. 11, no.10 (1985) 1001 -1016
7. Polychronopoulos, C.D., Kuck, D.: Guided Self-Scheduling: A Practical Scheme for Parallel Supercomputers. IEEE Trans. on Computers, vol.36, no. 12 (1987) 1425-1439
8. Hummel, S.E., Schonberg, E., Flynn, L.E.: Factoring : A Method for Scheduling Parallel Loops. Comm. ACM, vol. 35, no. 8 (1992) 90-101
9. Eager, D.L., Zahorjan, J.: Adaptive guided self-scheduling. Tech. Rep. 92-01-01. Dept. of Comput. Sci. and Eng,, univ. of Wash (1992)
10. Tzen, T.H., Ni, L.M.: Trapezoid self-scheduling : A practical scheduling scheme for parallel computer. IEEE Trans. on Parallel and Distributed Syst., vol.4 (1993) 87-98

11. Lucco, S.: A Dynamic Scheduling Method for irregular parallel Programs. Proc. ACM SIGPLAN '92 Conf. Programming Language Design and Implementation (1992) 200-211
12. Liu, J., Saletore, V.A., Lewis, T.G.: Safe Self-Scheduling: A Parallel Loop Scheduling Scheme for Shared-Memory Multiprocessors. Int. Parallel Programming, vol.22, no. 6 (1994) 589-616
13. Markatos, E.P., LeBlanc, T.J.: Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors. IEEE Trans. on Parallel and Distributed Syst, vol. 5, no. 4. (1994) 379-400
14. Subramaniam, S., Eager, D.L.: Affinity Scheduling of Unbalanced Workloads. Proc. Supercomputing '94 (1994) 214-226
15. Yan, Y., Jin, C., Zhang, X.: Adaptively Scheduling Parallel Loops in Distributed Shared-Memory Systems. IEEE Trans. on Parallel and Distributed Syst., vol. 8, no.1 (1997) 70-81
16. Campione, M.: The Java Tutorial, Addison-Wesley (1999)