International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 4, Number 2, pages 279–290, July 2014

Novel List Scheduling Strategies for Data Parallelism Task Graphs

Yang Liu
Graduate School of Science and Engineering, Ritsumeikan University,
Kusatsu, Shiga, 525-8577 Japan

Lin Meng, Ittetsu Taniguchi, Hiroyuki Tomiyama
College of Science and Engineering, Ritsumeikan University,
Kusatsu, Shiga, 525-8577 Japan

Received: February 15, 2014
Revised: May 1, 2014
Accepted: May 30, 2014
Communicated by Susumu Matsumae

Abstract

This paper studies task scheduling algorithms which schedule a set of tasks on multiple cores so that the total scheduling length is minimized. Most of the algorithms developed in the past assume that a task is executed on a single core. Unlike the previous algorithms, the algorithms studied in this paper allow a task to be executed on multiple cores. This paper proposes six algorithms. All of the six algorithms are based on list scheduling, but the strategy for priority assignment is different. In our experiments, the six algorithms as well as an integer linear programming method are evaluated.

Keywords: task scheduling, multicore, data parallelism

1 Introduction

Due to the spread deployment of multicore processors not only in high-performance computers but also in embedded systems, task scheduling has now become a more important problem than ever. In general, an application is modeled as a task graph, where nodes represent tasks (i.e., pieces of the application) and direct edges represent data- or control-flow dependency between two tasks. A task scheduling problem decides when and on which core each task is executed so as to minimize the overall schedule length while meeting constraints on flow dependency and the number of cores available. Schedule length is execution time of the application. The task scheduling problem is known to be NP-hard [1], and has been extensively studied over decades to develop efficient heuristic algorithms.

Most of the previous researches assume that a task does not have data parallelism and runs on a single core, where data parallelism denotes the parallel execution of a single task on data distributed over multiple cores. However, this assumption does not hold true in many systems. Tasks may have data parallelism and run on multiple cores. This paper studies scheduling of data-parallel tasks on multicore processors.

There exist several research efforts on task scheduling with data parallelism in the past. Recent studies include [2, 3, 4]. In [2], Yang and Ha proposed a scheduling technique for data-parallel tasks based on integer linear programming (ILP) formulation, and extended the technique towards

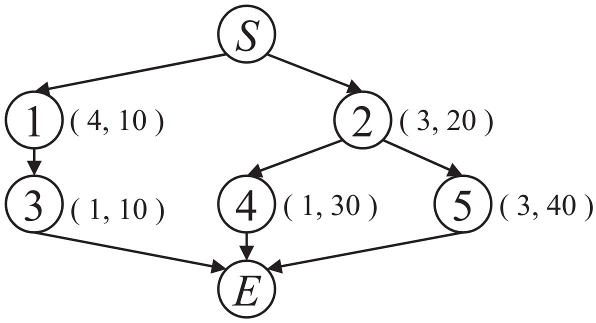


Figure 1: An example of a task graph.

pipelined scheduling in [3]. Their techniques perform task scheduling and allocation simultaneously, where allocation means a design process which decides the number of cores assigned to each task. Vydyanathan also proposed a simultaneous scheduling and allocation algorithm for data-parallel tasks [4]. The common assumption in [2, 3] and [4] is that the degree of data parallelism in tasks, i.e., the number of cores assigned to the task, is flexible, and the execution time of the task for each parallelism is known prior to task scheduling decision. However, this assumption may not be practical in some cases.

In contrast, this paper assumes that a task has a fixed degree of data parallelism. Tasks may have different degrees of data parallelism, but the degrees are not changed during task scheduling. To the best of our knowledge, this is the first paper to propose efficient algorithms for the scheduling problem.

The contributions of this paper are as follows:

- This paper first defines and formulates the scheduling problem for a set of data-parallel tasks.
- This paper proposes six algorithms for the scheduling problem.
- This paper presents quantitative evaluations of the algorithms using standard task sets.

The rest of this paper is organized as follows. Section 2 defines the scheduling problem, and Section 3 proposes six algorithms for the problem. Experiments are shown in Section 4, and Section 5 concludes this paper.

2 Problem Definition

This section defines the task scheduling problem addressed in this paper.

2.1 Problem Description

This work assumes a homogeneous multicore processor. An application is modeled as an acyclic directed graph (DAG), so called a task graph, where a node represents a task and a directed edge represents a flow dependency between the two tasks.

Figure 1 shows an example of a task graph. In this graph, tasks labeled S and E are dummy tasks which do not perform any meaningful computation. Tasks S and E denote an entry point and an exit point of the application, respectively. Two integer values are associated with each task. The first number denotes the degree of data parallelism of the task, and the latter number denotes the execution time of the task. For example, task 1 runs on four cores, and it takes 40 time units to perform the task.

In this paper, we assume that individual tasks are written in a parallel programming language by human programmers, and that the programmers decide the degree of data parallelism. How to decide the degree of parallelism and how to know the execution time are up to the programmers, and are out of the scope of this paper.

2.2 ILP Formulation

The task scheduling problem described above can be formulated by an integer linear programming (ILP) problem.

Let $time_i$, $start_i$, and $finish_i$ denote the execution time, start time and finish time of task i, respectively. par_i denotes the data parallelism, meaning that task i must be mapped onto par_i cores. $flow_{i1,i2}$ denotes a flow dependency between tasks $i1$ and $i2$. $flow_{i1,i2}$ is 1 if task $i1$ must proceed task $i2$. $map_{i,j}$ denotes mapping of tasks on cores. $map_{i,j}$ is 1 if task i is mapped to core j.

Then, the task scheduling problem is formally defined as follows: Given $time_i$, par_i and $flow_{i1,i2}$, decide $start_i$, $finish_i$ and $map_{i,j}$ which minimize the objective function (1), while meeting the constraints (2), (3), (4) and (5).

Minimize:

$$\max_i(finish_i) \tag{1}$$

Subject to:

$$\forall i \qquad \sum_j map_{i,j} = par_i \tag{2}$$

$$\forall i \qquad finish_i = start_i + time_i \tag{3}$$

$$\forall i1, i2, j \quad map_{i1,j} + map_{i2,j} \leq 1 \quad OR \quad finish_{i1} \leq start_{i2} \quad OR \quad finish_{i2} \leq start_{i1} \tag{4}$$

$$\forall i1, i2 \qquad flow_{i1,i2} = 1 \rightarrow finish_{i1} \leq start_{i2} \tag{5}$$

It should be noted that $finish_i$ is a dependent variable on $start_i$ (see Equation 3). Therefore, the decision variables of the scheduling problem are $start_i$ and $map_{i,j}$. We call values of $start_i$ and $map_{i,j}$ for all i and j a *schedule* (or a scheduling result) of the task graph. A schedule is called *feasible* if the schedule satisfies all of the constraints (2), (3), (4) and (5). The maximum value of $finish_i$, which is the objective function (1), is called the *schedule length*. Then, the scheduling problem can be restated as follows: For a given task graph, find a feasible schedule with the minimum schedule length.

Although optimal scheduling results can be obtained by solving the ILP formulas, it is not practical for large task sets in terms of CPU runtime. In the next section, we propose six heuristic algorithms based on list scheduling.

3 The Proposed Algorithms

In this section, we propose six algorithms for the scheduling problem. All of the six algorithms are based on list scheduling, but their priority assignment strategies are different.

3.1 The Overall Algorithm

The basis of the six algorithms is a simple list scheduling algorithm. An important concept of list scheduling is *ReadyList*, which contains a set of executable tasks. Here, a task is said to be executable if all of its preceding tasks are completed. Below is a fundamental algorithm of list scheduling.

1. Initialize $ReadyList$ and $IdleCores$;
   $ReadyList = \emptyset$;
   $IdleCores =$ the number of total cores;

2. Select a task which has the highest priority from *ReadyList*, and schedule the task as early as it is schedulable;

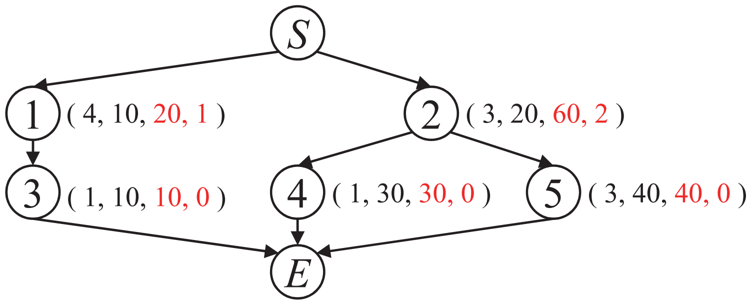


Figure 2: Critical path length and the number of immediate successors.

3. Finish if all tasks have been scheduled. Otherwise, update *ReadyList* and *IdleCores* and go back to step 2;

There exist a large number of variations of list scheduling depending on how to define the priority in step 2.

3.2 A Motivating Example

In [5], Kasahara and Narita propose a list-based scheduling algorithm, named CP/MISF (critical path/most immediate successor first). The CP/MISF algorithm is designed for task scheduling without data parallelism. Although it was proposed three decades ago, it is still recognized as one of the best heuristic algorithms because of the high quality of results as well as the low computational complexity. As the name of the algorithm indicates, the CP/MISF algorithm takes into account two factors to define the priority of tasks; the critical path length and the number of immediate successors. Figure 2 shows the same task graph as in Figure 1, but we have added two numbers to each task, denoting the critical path length and the number of immediate successors. The critical path length of a task is the length of the longest path from the node to the exit node. For example, the critical path length of task 2 is 60, by adding the execution time of task 2 and that of task 5. In the CP/MISF algorithm, the priority of tasks is defined according to the following two rules:

1. If the critical path of task i is longer than that of task j, task i has a higher priority than task j.

2. In case tasks i and j has the same critical path length, if task i has more immediate successors than task j, task i has a higher priority than task j.

Figure 3 shows the schedule when the CP/MISF algorithm is applied to the task graph in Figure 2. At time $t = 0$, tasks 1 and 2 are executable, but task 2 is scheduled first because it has a longer critical path. Then, tasks 5 and 4 are scheduled, followed by tasks 1 and 3. The total schedule length is 80 time units.

The CP/MISF algorithm works nice for tasks without data parallelism. However, the CP/MISF algorithm is not always efficient for tasks with data parallelism. Actually, the schedule in Figure 3 is not optimal. Figure 4 shows a better schedule for the same task set. The policy of this scheduling is that a task with the largest data parallelism has a priority. Due to this policy, task 1 is scheduled first, and then, task 3 is enabled to run in parallel with another task. Of course, this policy is not always optimal, but this example demonstrates that the degree of data parallelism should be taken into account in the priority.

3.3 The Proposed Priorities

We propose six algorithms, all of which are based on list scheduling, but their definitions of priority are different. In order to define the priority, we take into account three factors as follows:

	$t = 0$		20		40		60	80
Core 0	T2	T2	T5	T5	T5	T5	T1	T3
Core 1	T2	T2	T5	T5	T5	T5	T1	
Core 2	T2	T2	T5	T5	T5	T5	T1	
Core 3			T4	T4	T4		T1	

Figure 3: Schedule obtained by the CP/MISF algorithm.

	$t = 0$		20		40		60	80
Core 0	T1	T2	T2	T5	T5	T5	T5	
Core 1	T1	T2	T2	T5	T5	T5	T5	
Core 2	T1	T2	T2	T5	T5	T5	T5	
Core 3	T1	T3		T4	T4	T4		

Figure 4: Schedule which takes into account the degree of data parallelism.

- P: The degree of data parallelism
- C: The length of critical path
- S: The number of immediate successors

Based on the three factors, the first algorithm proposed in this paper defines the priority of tasks as follows:

1. If task i has a larger data parallelism than task j, task i has a higher priority than task j.
2. In case tasks i and j has the same degree of data parallelism, if the critical path of task i is longer than that of task j, task i has a higher priority than task j.
3. In case tasks i and j has the same degree of parallelism and the same length of critical paths, if task i has more immediate successors than task j, task i has a higher priority than task j.

The algorithm based on the above priority is named PCS since the three factors (P, C and S) are prioritized in the order of P-C-S. Let $PriorityPCS_i$ denote the priority of task i in the PCS algorithm, where a higher value means a higher priority. A simple formula to define $PriorityPCS_i$ is as follows.

$$PriorityPCS_i = U^2 \cdot P_i + U \cdot C_i + S_i \tag{6}$$

Here, P_i, C_i, and S_i denote the values of P, C and S factors for task i, and U is a constant integer number which is larger than any of P_i, C_i, and S_i for any i.

In the similar manner, we can define five algorithms CPS, CSP, SCP, PSC and SPC with different ordering of the three factors. The task priorities in the five algorithms are defined as follows:

$$\begin{aligned}
PriorityCPS_i &= U^2 \cdot C_i + U \cdot P_i + S_i && (7)\\
PriorityCSP_i &= U^2 \cdot C_i + U \cdot S_i + P_i && (8)\\
PrioritySCP_i &= U^2 \cdot S_i + U \cdot C_i + P_i && (9)\\
PriorityPSC_i &= U^2 \cdot P_i + U \cdot S_i + C_i && (10)\\
PrioritySPC_i &= U^2 \cdot S_i + U \cdot P_i + C_i && (11)
\end{aligned}$$

A common important feature in the six algorithms is that priorities are static. The priorities can be computed prior to scheduling, and they do not change during scheduling.

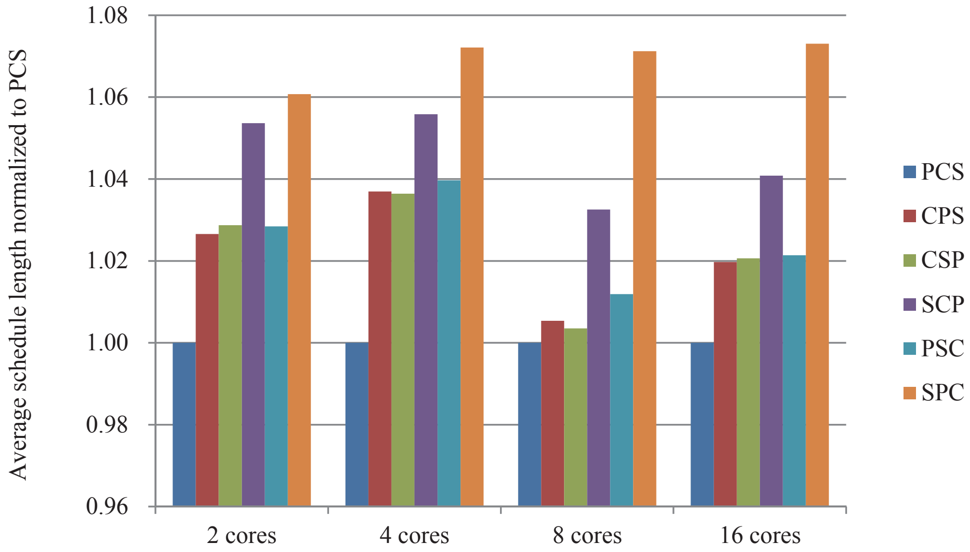


Figure 5: Averages of normalized schedule lengths for task graphs with 50 tasks.

The time complexity of the six algorithms is $O(N^2)$, where N is the number of tasks, assuming that the number of cores is constant. First, it takes $O(N^2)$ to compute the critical path lengths of the nodes. Then, we sort the nodes three times since we use three factors (P, C and S), and each sorting takes $O(N \log N)$. Therefore, it takes $O(N^2)$ in order to compute the priorities of the nodes before running the list scheduling shown in Section 3.1. The list scheduling process is repeated N times, and in each iteration, it takes $O(N)$ to update the ready list. Thus, we get the overall complexity of $O(N^2)$.

4 Experiments

We implemented the six algorithms in the C language, and tested their effectiveness. We used 43 task graphs from *Standard Task Graph (STG) Set* developed at Waseda University [6]. Forty out of the 43 task graphs are randomly generated ones, and the other three tasks are based on actual applications. Since tasks in STG do not assume data parallelism, we randomly assigned the degree of data parallelism to the tasks. The number of cores was changed from two to sixteen. In addition to the six algorithms presented in this paper, an integer linear programming (ILP) technique (see Section 2.2) was evaluated. In order to solve the ILP problems, IBM ILOG CPLEX 12.5 was used. Since exact solutions could not be found in a practical time, we limited the CPU time of CPLEX up to 60 minutes on dual Xeon processors (E5-2650, 2.00Hz, 128GB memory), and the best solution found at that time was compared with the six algorithms.

4.1 Results for Random Task Graphs

First, we conducted experimemts using 20 random task graphs, each of which consists of 50 tasks. Figure 5 shows the average schedule lengths of the 20 task graphs obtained by the six algorithms proposed in this paper. The schedule lengths are normalized to the PCS algorithm. This graph clearly shows the effectiveness of the PCS algorithm.

Table 1 shows detailed results for individual task graphs. The first column labeled as "Tid" shows the task ID, and the following columns show the lengths of the schedules obtained by the seven methods (the six algorithms proposed in this paper and the ILP method). For each benchmark, the best solution is shaded in yellow. X in the ILP column means that no feasible solution was found within 60 minutes in CPU time. In many cases, the ILP method failed to find a feasible schedule

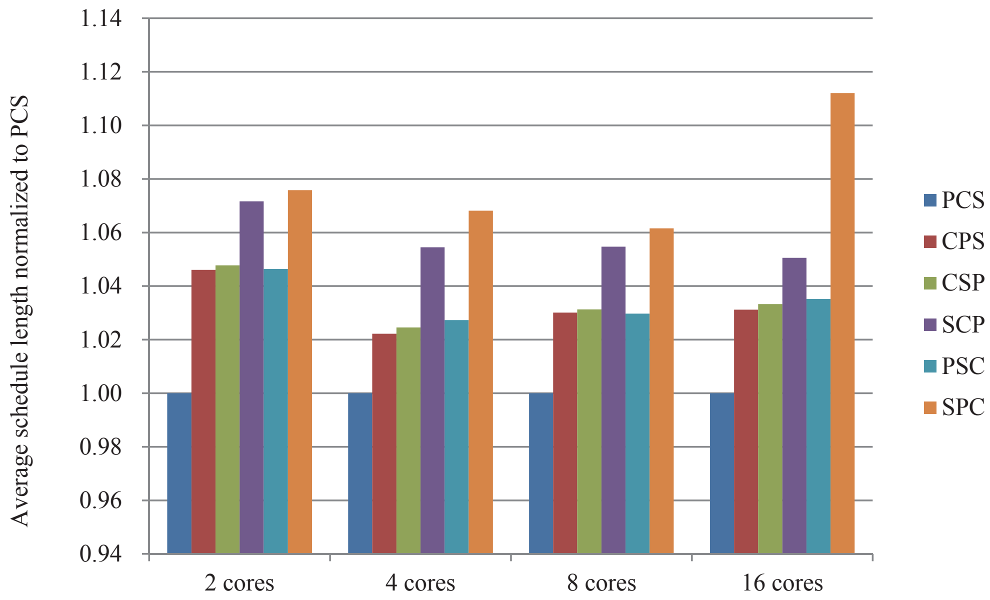


Figure 6: Averages of normalized schedule lengths for task graphs with 100 tasks.

within the limited time. Even when the ILP method found feasible schedules, they are lengthy. Although the PCS algorithm yields the best schedule results on average, Table 1 shows that the effectiveness of the six algorithms highly depends on the task graph.

Next, we conducted experimemts using 20 random task graphs, each of which consists of 100 tasks. Figure 6 shows the average schedule lengths of the 20 task graphs obtained by the six algorithms proposed in this paper. Again, this graph clearly shows the effectiveness of the PCS algorithm.

Table 2 shows detailed results for individual task graphs with 100 tasks. Compared with Table 1, the PCS algorithm yeilds best solutions in more cases, and the ILP method failed to find a feasible solution in more cases.

Table 1: Schedule lengths for task graphs with 50 tasks.

	2 cores							4 cores						
Tid	PCS	CPS	CSP	SCP	PSC	SPC	ILP	PCS	CPS	CSP	SCP	PSC	SPC	ILP
00	203	200	200	210	200	212	204	168	178	178	175	180	178	X
01	232	233	233	249	233	251	232	220	214	214	229	214	232	X
02	188	192	192	199	192	199	197	173	173	173	183	174	186	197
03	224	224	224	230	225	228	241	194	202	202	211	202	201	X
04	177	181	181	189	181	191	180	167	168	168	171	170	186	X
05	495	496	496	520	496	531	504	439	443	438	448	449	448	464
06	351	363	363	372	363	375	356	275	293	293	294	293	305	X
07	384	387	387	394	391	400	430	357	348	348	358	349	367	X
08	434	456	456	447	456	464	460	409	415	415	424	415	412	456
09	386	397	397	412	397	410	398	327	373	373	368	373	363	X
10	153	162	162	156	163	159	165	131	139	139	134	140	134	X
11	205	213	213	208	213	210	198	181	192	192	177	192	177	191
12	208	211	211	213	211	213	200	197	195	195	201	195	212	X
13	238	252	252	282	252	287	248	186	214	214	239	214	254	X
14	195	197	197	196	197	201	208	171	181	181	175	181	175	X
15	425	448	448	452	448	444	427	376	377	377	383	373	386	382
16	374	390	390	398	395	408	389	318	330	330	342	331	356	360
17	439	448	467	492	456	491	471	377	396	396	414	396	414	X
18	428	443	443	438	443	430	429	403	390	390	408	392	414	401
19	393	409	409	416	403	407	404	342	368	368	368	369	373	X

	8 cores							16 cores						
Tid	PCS	CPS	CSP	SCP	PSC	SPC	ILP	PCS	CPS	CSP	SCP	PSC	SPC	ILP
00	149	152	152	151	160	160	X	156	149	152	148	152	160	211
01	203	210	210	197	210	212	X	195	204	205	213	204	213	227
02	161	153	153	156	153	164	X	150	143	143	149	143	146	199
03	175	180	180	183	180	189	X	169	174	174	171	174	184	219
04	150	155	155	160	154	172	X	158	159	159	157	159	167	188
05	432	402	402	438	402	439	X	406	399	399	413	399	451	463
06	259	260	252	269	262	281	X	268	261	261	263	261	282	360
07	336	325	325	324	324	338	X	301	283	283	298	283	288	431
08	366	362	362	367	362	377	X	360	347	347	370	347	369	438
09	323	324	324	338	324	349	X	289	303	303	309	303	286	382
10	127	134	134	128	134	132	193	126	133	133	129	133	133	168
11	180	173	173	178	173	195	X	135	155	155	172	155	186	175
12	183	180	180	183	180	183	X	174	182	183	183	182	197	213
13	171	170	169	215	170	233	X	154	174	174	199	174	201	243
14	166	169	169	164	169	164	X	160	160	158	162	160	166	191
15	304	314	314	307	314	307	X	325	336	336	331	336	343	445
16	269	289	289	319	302	323	X	286	301	301	291	304	286	387
17	306	305	305	326	310	342	X	333	337	337	319	338	336	481
18	358	357	357	354	362	363	403	342	350	350	372	350	382	415
19	361	373	373	371	373	371	X	334	332	332	319	332	334	401

Table 2: Schedule lengths for task graphs with 100 tasks.

	2 cores							4 cores						
Tid	PCS	CPS	CSP	SCP	PSC	SPC	ILP	PCS	CPS	CSP	SCP	PSC	SPC	ILP
00	431	447	447	463	445	466	X	388	396	396	399	392	406	X
01	401	411	411	416	411	418	X	348	361	366	381	362	380	X
02	459	480	486	508	480	512	X	413	429	429	448	429	466	X
03	406	419	419	427	416	431	501	341	363	363	375	365	375	X
04	393	417	417	408	422	416	459	454	369	376	387	382	396	X
05	814	833	833	868	842	873	X	704	707	698	739	698	753	X
06	868	886	882	916	886	899	965	785	778	778	790	782	813	X
07	861	872	872	888	869	929	997	760	773	773	797	773	806	X
08	796	818	818	824	818	806	X	701	726	726	750	726	739	X
09	947	963	963	958	963	974	X	783	806	810	852	810	843	X
10	464	485	485	488	485	490	532	385	402	402	405	402	417	X
11	445	464	466	456	466	455	X	394	406	410	400	416	400	X
12	469	484	484	522	484	528	551	432	450	450	477	450	490	X
13	480	502	502	513	502	513	X	404	435	440	426	437	431	X
14	391	417	417	422	415	418	X	354	353	357	370	359	369	X
15	781	792	792	873	792	866	X	706	695	694	721	697	734	X
16	764	862	860	868	857	863	X	667	700	700	722	700	730	X
17	860	920	922	936	922	927	X	746	796	798	828	798	818	X
18	724	777	792	794	779	828	X	628	669	662	651	669	686	X
19	749	825	825	860	825	844	856	700	725	726	802	743	814	X

	8 cores							16 cores						
Tid	PCS	CPS	CSP	SCP	PSC	SPC	ILP	PCS	CPS	CSP	SCP	PSC	SPC	ILP
00	356	355	355	357	361	368	X	335	351	354	358	363	346	494
01	326	345	347	350	346	366	X	307	327	317	326	337	327	483
02	380	380	382	387	382	387	X	365	352	381	353	381	353	501
03	338	354	354	371	353	365	X	314	329	336	331	354	327	449
04	340	355	344	342	350	360	X	317	314	324	320	348	320	489
05	713	701	701	764	701	759	X	668	690	699	690	716	690	920
06	712	732	730	730	730	731	X	687	705	719	705	751	701	789
07	675	728	728	712	728	709	X	665	694	690	696	680	694	945
08	637	669	669	671	669	674	X	607	618	620	618	639	618	900
09	785	754	754	748	754	774	X	728	742	786	742	788	742	944
10	338	354	375	358	356	358	X	362	370	372	362	361	358	501
11	353	382	384	389	381	398	X	336	342	342	344	351	398	480
12	431	435	435	441	435	443	X	410	394	397	437	414	443	541
13	382	402	405	395	402	406	X	375	395	399	431	394	406	556
14	327	344	343	342	343	347	X	313	337	338	325	338	347	473
15	697	671	658	714	658	692	X	606	625	625	613	597	692	978
16	625	649	649	705	657	721	X	648	670	670	671	670	721	876
17	730	770	770	816	770	783	X	677	727	727	750	727	783	1024
18	657	668	668	673	668	679	X	591	644	652	615	652	679	832
19	679	705	701	801	701	775	X	676	682	686	731	690	775	796

Table 3: Schedule lengths for realistic task graphs.

(a) fpppp

	2 cores	4 cores	8 cores	16 cores
PCS	5361	4881	4533	4487
CPS	5738	5152	4987	4905
CSP	5738	5152	4987	4905
SCP	5809	5108	4946	4899
PSC	5363	4884	4538	4531
SPC	5509	5032	4689	4623

(b) robot

	2 cores	4 cores	8 cores	16 cores
PCS	1951	1739	1731	1615
CPS	1961	1769	1672	1641
CSP	1961	1769	1672	1641
SCP	1975	1791	1715	1637
PSC	1952	1767	1731	1615
SPC	2002	1783	1687	1627

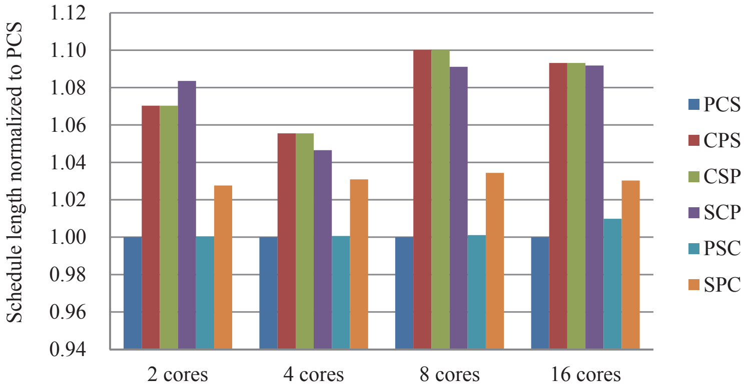
(c) sparse

	2 cores	4 cores	8 cores	16 cores
PCS	1458	1242	1132	1038
CPS	1442	1312	1222	1140
CSP	1442	1312	1222	1140
SCP	1454	1276	1172	1104
PSC	1458	1242	1136	1038
SPC	1454	1248	1166	1086

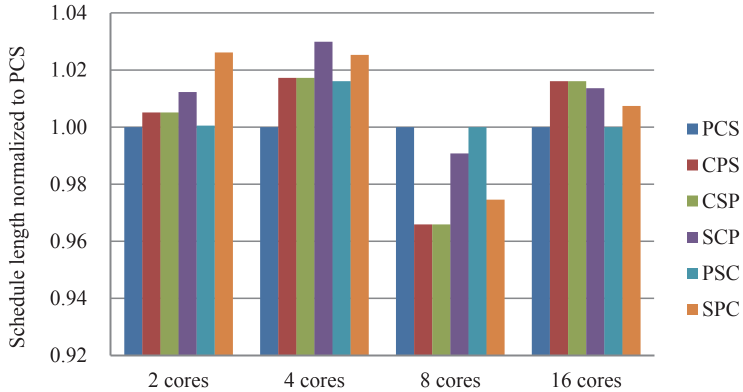
4.2 Results for Realistic Task Graphs

In addition to the random task graphs, we used three task graphs which are derived from realistic applications. The STG contains three task graphs based on realistic application programs, i.e., (a) a part of fpppp from in the SPEC benchmarks, (b) robot control and (c) sparse matrix solver [6]. The task graphs are generated by the OSCAR Parallelizing Compiler [7, 8, 9]. The task graphs of fpppp, robot and sparse contain 334 tasks, 88 tasks, and 96 tasks, respectively.

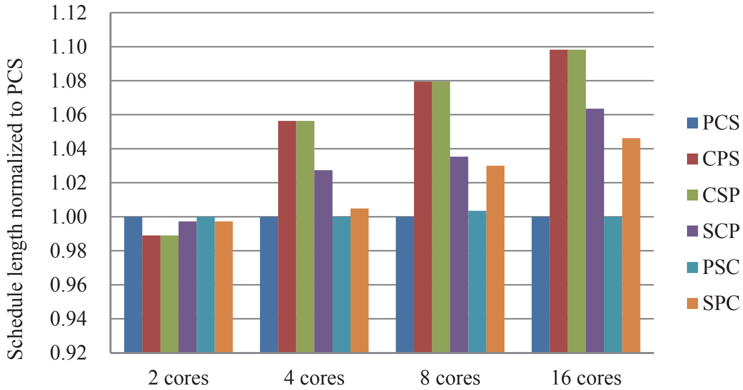
Table 3 shows the average schedule lengths for the three realistic task graphs. In order to understand more easily, we normalized the all results by the result of PCS, and converted the data to bar charts as Figures 7 (a), (b) and (c). We found the PCS algorithm yields good schedules in general. However, for robot on 8 cores and sparse on 2 cores, some others algorithms perform better than PCS.



(a) fpppp



(b) robot



(c) sparse

Figure 7: Normalized schedule lengths for realistic task graphs.

5 Conclusions

This paper proposed six algorithms for scheduling tasks on multi/many-core processors. Unlike most of previous research efforts, the proposed algorithms schedule tasks which have data parallelism and run on multiple cores. The experimental results show that, among the six algorithms, the PCS algorithm yields the best schedule results on average.

In some task sets, the PCS algorithm does not yield good schedules. The effectiveness of the six algorithms heavily depends on the structure of task graphs. In the future, we will investigate the algorithms theoretically and compare them with optimal schedules in order to further improve the algorithms. Also, the current algorithms do not take into account communication costs, which should be addressed in the future.

References

[1] E. G. Coffman: Computer and Job-shop Scheduling Theory, Wiley, 1976.

[2] H. Yang and S. Ha: ILP based data parallel multi-task mapping/scheduling technique for MPSoC, International SoC Design Conference (ISOCC), 2008.

[3] H. Yang and S. Ha: Pipelined data parallel task mapping/scheduling technique for MPSoC, Design Automation and Test in Europe (DATE), 2009.

[4] N. Vydyanathan, S. Krishnamoorthy, G. M. Sabin, U. V. Catalyurek, T. Kurc, P. Sadayappan, and J. H. Saltz: An integrated approach to locality-conscious processor allocation and scheduling of mixed-parallel applications, IEEE Trans. on Parallel and Distributed Systems, vol. 20, no. 8, pp. 1158-1172, Aug. 2009.

[5] H. Kasahara and S. Narita: Practical multiprocessor scheduling algorithms for efficient parallel processing, IEEE Trans. on Computers, vol. C-33, no. 11, Nov. 1984.

[6] http://www.kasahara.elec.waseda.ac.jp/schedule/ (Last accessed: July 24, 2013)

[7] H. Kasahara, H. Honda and S. Narita: Parallel Processing of Near Fine Grain Tasks Using Static Scheduling on OSCAR, Proc. IEEE ACM Supercomputing '90, 1990.

[8] H. Kasahra, H. Honda, A. Mogi, A. Ogura, K. Fujiwara and S. Narita: A Multi-grain Parallelizing Compilation Scheme for OSCAR, Proc. 4th Workshop on Languages and Compilers for Parallel Computing, 1991.

[9] A. Yoshida, K. Koshizuka and H. Kasahara: Data-Localization for Fortran Macrodataflow Computation Using Partial Static Task Assignment, Proc. 10th ACM Int'l Conf. on Supercomputing, pp. 61-68, 1996.