An Efficient Task Clustering Heuristic for Scheduling DAGs on Multiprocessors

Article · November 1996

Source: CiteSeer

2 authors, including:

Michael A. Palis
Rutgers, The State University of New Jersey
53 PUBLICATIONS **770** CITATIONS

SEE PROFILE

An Efficient Task Clustering Heuristic for Scheduling DAGs on Multiprocessors *

Jing-Chiou Liou

AT&T Laboratories
Middletown, NJ 07748, USA
jing@jolt.mt.att.com

Michael A. Palis
Department of Computer Science
Rutgers University
Camden, NJ 08102, USA
palis@crab.rutgers.edu

Abstract

For task clustering with no duplication, the DSC algorithm of Gerasoulis and Yang is empirically the best known algorithm to date in terms of both speed and solution quality. The DSC algorithm is based on the critical path method. In this paper, we present an algorithm called CASS-II for task clustering with no duplication which is competitive to DSC in terms of both speed and solution quality. With respect to speed, CASS-II is better than DSC: it has a time complexity of $O(|E| + |V| \lg |V|)$, as opposed to DSC's $O((|E| + |V|) \lg |V|)$. Indeed, our experimental results show that CASS-II is between 3 to 5 times faster than DSC. (It is worth pointing out that we used the C code for DSC developed by the authors of the DSC algorithm. The C code for CASS-II was developed by the authors of this paper.) With respect to solution quality, experimental results show that CASS-II is virtually as good as DSC and, in fact, even outperforms DSC for very fine grain DAGs (granularity less than 0.6).

1. Introduction

Previous work on compilers for parallel machines have focused largely on ''parallelizing" or ''vectorizing" compilers that automatically extract parallelism from existing sequential programs (e.g., ''dusty-deck" FORTRAN programs). While such compilers have their niche of applications, there is a greater and more pressing need today to develop compilers for parallel programming languages that incorporate language constructs for explicitly expressing concurrency in programs.

Many existing compilers for parallel programming languages do not perform granularity optimization, and if at all, make use of very simple algorithms. In some compilers, more sophisticated techniques for granularity optimization are used, but they can only be applied to certain segments of the parallel code. For example, loop spreading and loop coalescing are commonly used for granularity optimization. However, these techniques are only applicable at the loop level and can not be used to optimize the granularity of program segments that exist within loops or are outside of loops.

Our research addresses the granularity optimization problem in a more general context by using a parallel program representation (the task graph) that is essentially language independent. Moreover, unlike previous work which focuses on optimization for $specific$ architectures (as is the case for most commercial compilers), our investigation uses a parameterized parallel machine model, thus allowing us to develop granularity optimization techniques that are applicable to a wide range of parallel architectures. Consequently, we will not only be able to assess the effectiveness of today's parallel machines in solving MPP applications, but we will also be able to determine the key architectural features required by these applications, whether these features exist in current machines, and how future MPP machines should be built in order to solve MPP applications much more efficiently and cost-effectively.

Finding a clustering of a task graph that results in minimum overall execution time is an NP-hard problem [1, 12, 14]. Consequently, practical algorithms must sacrifice optimality for the sake of efficiency. We have investigated two versions of the problem: clustering without task duplication and clustering with task duplication. In clustering without task duplication, the tasks are partitioned into disjoint clusters and exactly one copy of each task is scheduled. In clustering with task duplication, a task may have several copies in different clusters, each of

*This research was supported in part by NSF Grant IRI-9296249.

which is independently scheduled. In general, clustering with task duplication produces shorter schedules than the ones produced by clustering without task duplication. We have developed efficient heuristic algorithms for these two problems, established theoretical bounds on the quality of solutions they generate, and validated the theoretical performance empirically.

When task duplication is allowed, we have an algorithm[11] (CASS-I ,which stands for Clustering And Scheduling System I) which for a task graph with arbitrary granularity, produces a schedule whose makespan is at most twice optimal. Unfortunately, we are unable to find a $provably$ good clustering algorithm with no task duplication for general task graphs. This problem appears to be very difficult because it is known that for general task graphs, clustering with no task duplication remains NP-hard even when the solution quality is relaxed to be within twice the optimal solution [12]. Consequently, we directed our efforts to develop an algorithm (CASS-II) which has fast time complexity of $O(|E| + |V|lg|V|)$ and good $empirical$ performance.

We compared CASS-II with the DSC algorithm of [3], which is empirically the best known algorithm for clustering without task duplication. Our experimental results [8] indicate that CASS-II outperforms DSC in terms of speed (3 to 5 times faster). Moreover, in terms of solution quality, CASS-II is very competitive: it is better than DSC for grain sizes less or equal to 0.6, and its superiority increases as the DAG becomes increasingly fine grain. On the other hand, for task graph with grain size 0.6 or greater, DSC becomes competitive and in some cases even outperforms CASS-II, but by no more than 3%.

2 Problem Statement

An important factor that determines program performance on a distributed memory parallel machine is the speed at which computation and communication can be performed. Over the last decade, processor speeds have increased at the dramatic rate of 50% a year. On the other hand, communication speeds have not kept pace. However, the cost of routing a message depends not only on its *transport time* (the time that it stays in the network) but also on the *overhead* spent in executing the operating system routines for sending and receiving the message. On contemporary machines, this software overhead is so large that it often dominates the transport time, even for messages traveling very long distances in the network. Typically, the message overhead is of the order of hundreds to a few thousands of processor clock cycles.

A parallel program can be viewed abstractly as a collection of tasks, where each task consists of a sequence of instructions and input and output parameters. A task starts execution only after all of its input parameters are available; output parameters are sent to other tasks only after the task completes execution. This notion of a task is called the ''macro-dataflow model'' by Sarkar [14] and is used by other researchers [13, 2, 12, 15, 16].

In the macro-dataflow model, a parallel program is represented as a weighted directed acyclic graph (DAG) $G = (V, E, \mu, \lambda)$, where each $v \in V$ represents a task whose execution time is $\mu(u)$ and each directed edge (or arc) $(u, v) \in E$ represents time constraint that task u should complete its execution before task v can be started. In addition, u communicates data to v upon its completion; the delay incurred by this data transfer is $\lambda(u, v)$ if u and v reside in different processors and zero otherwise. In other words, task v cannot begin execution until all of its predecessor tasks have completed and it has received all data from these tasks.

The granularity of a task graph is an important parameter which we take into account when analyzing the performance of our algorithms. Basically there are two distinct strategies for scheduling: parallelizing tasks or sequentializing tasks. The trade-off point between parallelization and sequentialization is closely related to the granularity value: the ratio between the task execution time and communication time. If communication cost is too high, parallelization is not encouraged.

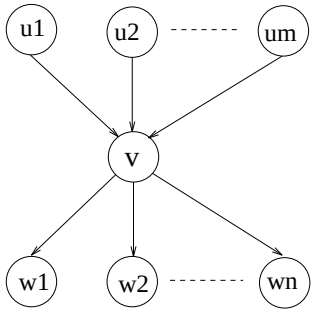


Figure 1. Definition of the task graph granularity.

We adopt the definition of granularity given in [3]. Let $G = (V, E, \mu, \lambda)$ be a weighted DAG. For a node $v \in V$ as shown in Figure 1, let

$g_1(v) = \min\{\mu(u)|(u, v) \in E\}/\max\{\lambda(u, v)|(u, v) \in E\}$
$g_2(v) = \min\{\mu(w)|(v, w) \in E\}/\max\{\lambda(v, w)|(v, w) \in E\}$.

The *grain-size* of v is defined as $\min\{g_1(v), g_2(v)\}$. The $granularity$ of DAG G is given by $g(G) = \min\{g(v) | v \in V\}$. One can verify that for the DAG G of Figure 2, $g(G) = \frac{1}{7}$.

The high communication overhead in existing distributed memory parallel machines imposes a minimum threshold on program granularity below which performance degrades significantly. To avoid performance degradation, one solution would be to coalesce several fine grain tasks into single coarser grain tasks. This reduces the communication overhead but increases the execution time of the (now coarser grain) tasks. Because of this inherent tradeoff, the goal is to determine the program granularity that results in the fastest total parallel execution time. This problem is called *grain size optimization*.

Grain size optimization can be viewed as the problem of scheduling the tasks of the program on the processors of the parallel machine such that the finish time of the last task (or ''makespan of the schedule'') is minimized. Much of the early work in scheduling algorithms considered only the task execution times and assumed zero communication times between interacting tasks.

More recent work in scheduling algorithms explicitly consider inter-task communication times. The basic idea behind most of these algorithms is ''task clustering'', i.e., scheduling several communicating tasks in the same processor so that the communications between these tasks are realized as local memory accesses within the processor, instead of message transmissions across the interconnection network. In other words, the communication time between two tasks becomes zero when these tasks are mapped to the same processor. The result is a reduction in the message overhead, and hence total parallel execution time.

Researchers have investigated two types of task clustering algorithms, depending on whether or not task duplication (or recomputation) is allowed. In general, for the same DAG, task clustering with duplication produces a schedule with a smaller makespan (i.e., total execution time) than when task duplication is not allowed.

Figure 2 gives an example of DAG; the node weights denote the task execution times and the arc weights denote the communication delays. Thus, assuming that each task resides in a separate processor, the earliest time that task T4 can be started is 19, which is the time it needs to wait until the data from task T2 arrives (the data from task T1 arrives earlier, at time 10). The makespan of the schedule is the length of the critical path, i.e., the path with the maximum sum of node and arc weights. In Figure 2, the critical

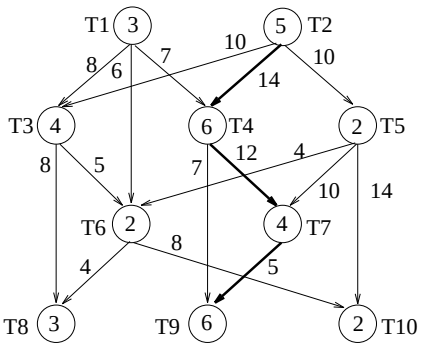


Figure 2. An example of DAG.

path is indicated by the bold arcs; its length, and hence the makespan of the schedule, is 52.

Task clustering -- with or without task duplication -- is an NP-hard problem [1, 12, 14]. Consequently, practical solutions will have to sacrifice optimality for the sake of efficiency. Nonetheless, task clustering heuristic algorithms have a number of properties in common when they try to achieve the goal of finding an optimal clustering for a DAG G. They all perform a sequence of *clustering refinements* starting with an initial clustering (initially each task is assumed to be in a cluster). Each step performs a refinement of the previous clustering so that the final clustering satisfies or ''near'' to the original goals. The algorithms are non-backtracking, i.e., once the clusters are merged in a refinement step, they cannot be unmerged afterwards.

A typical refinement step is to merge two clusters and $zero$ the edge that connect them. Zeroing the communication cost on the edge between two clusters is necessary for reducing the makespan (or parallel time) of the schedule. The $makespan$ is determined by the longest path in the scheduled graph. In other words, the makespan of a given schedule is equal to the time of the last task has been completely executed.

There are two important parameters in performing the refinement steps: the critical path of a task graph G and the earliest start time of each node $v \in V$. The *critical path* (CP) is the longest path in the task graph. In [3], Gerasoulis and Yang use *dominant sequence* (DS) instead of CP to represent the longest path of the scheduled task graph or the path whose length equals the actual makespan of the schedule. Nonetheless, the CP is so important that the heuristic algorithms rely on it for a global information of the task graph to guarantee the reduction of makespan in each refinement steps. We will show later the necessity of critical path as a global information. On the other hand, the earliest start time of a node $v \in V$ is the earliest time

that node v can start execution for any clustering. If the execution of every node in G is started at its earliest start time, then the schedule must be optimal.

For task clustering with no duplication, the DSC algorithm of Gerasoulis and Yang is empirically the best known algorithm to date in terms of both speed and solution quality. The DSC algorithm is based on the critical path method. At each refinement step, it computes the critical path of the clustered DAG constructed so far, i.e., the longest path from a source node to a sink node. (The length of a path is the sum of the node weights and edge weights along the path.) It then zeroes out an edge along the critical path if doing so will decrease the critical path length. The main source of complexity in the DSC algorithm is the computation of the critical path, which is done at each refinement step. On the other hand, the use of critical path information is also the reason why DSC performs very well compared to other algorithms.

3 CASS-II Algorithm

CASS-II employs a two-step approach. Let $G = (V, E, \mu, \lambda)$ be a weighted DAG. In the first step, CASS-II computes for each node v a value $s(v)$, which is the length of a longest path from a source node to v (excluding the execution time of v). Thus, $s(v)$ is the start time of v prior to any clustering of G. The s values are computed in node topological order of G. The s value of every source node is zero. Let v be a node all of whose immediate predecessors have been assigned s values. Then,

$$s(v) = \max\{s(u) + \mu(u) + \lambda(u, v) | (u, v) \in E\} \quad (1)$$

The second step is the clustering step. Just like DSC, it consists of a sequence of refinement steps, where each refinement step creates a new cluster or ''grows'' an existing cluster. Unlike DSC, CASS-II constructs the clusters bottom-up, i.e., starting from the sink nodes. To construct the clusters, the algorithm computes for each node v, a value $f(v)$, which is the longest path from v to a sink node in the current partially clustered DAG. Let $l(v) = s(v) + f(v)$. The algorithm uses $l(v)$ to determine whether the node v can be considered for clustering at current refinement step.

More precisely, the algorithm begins by placing every sink node v in its own cluster and by setting $f(v) = \mu(v)$ (hence, $l(v) = s(v) + \mu(v)$). The algorithm then goes through a sequence of iterations, where at each iteration it considers for clustering every node u all of whose immediate successors have been clustered (and hence been assigned f values). Call such a node *current*. For every current node u, its f value is computed as

$$f(u) = \max\{\mu(u) + \lambda(u, v) + f(v) | (u, v) \in E\} \quad (2)$$

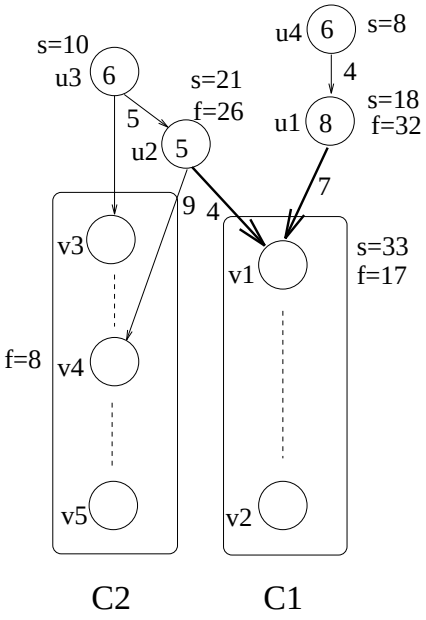


Figure 3. An example of computing the f values of current nodes.

The immediate successor v which determines $f(u)$ is defined as the *dominant successor* of current node u. In general, two or more current nodes may share the same dominant successor. Figure 3 illustrates the computation of the f values of current nodes. In Figure 3, nodes u_1 and u_2 are current nodes, but u_3 is not since one of its immediate successors, u_2, has not been clustered and assigned an f value. Since v_1 is the only immediate successor of u_1, $f(u_1) = \mu(u_1) + \lambda(u_1, v_1) + f(v_1) = 32$, and v_1 is the dominant successor of u_1. On the other hand, current node u_2 has two immediate successors v_1 and v_4. Thus, $f(u_2) = \max\{\mu(u_2) + \lambda(u_2, v_1) + f(v_1), \mu(u_2) + \lambda(u_2, v_4) + f(v_4)\} = 26$; the dominant successor of u_2 is v_1. Thus, current nodes u_1 and u_2 have the same dominant successor, v_1. Note that $l(u_1) = s(u_1) + f(u_1) = 18 + 32 = 50$ and $l(u_2) = s(u_2) + f(u_2) = 21 + 26 = 47$. Finally, we define the f value of a cluster as the f value of the first node in the cluster. For example, in Figure 3, assuming that node v_1 is the first node in the cluster C_1, then $f(C_1) = f(v_1) = 17$.

Once the l values of all current nodes have been computed, one of them will be placed in a cluster during the current iteration. The current nodes are considered for

clustering in nonincreasing order of their l values. For a given current node u, let v be its dominant successor and let C_v be the cluster containing v. Then, u is included in the cluster C_v if doing so does not increase *both* $f(u)$ and $f(C_v)$. Otherwise, u is placed in a new cluster all to itself.

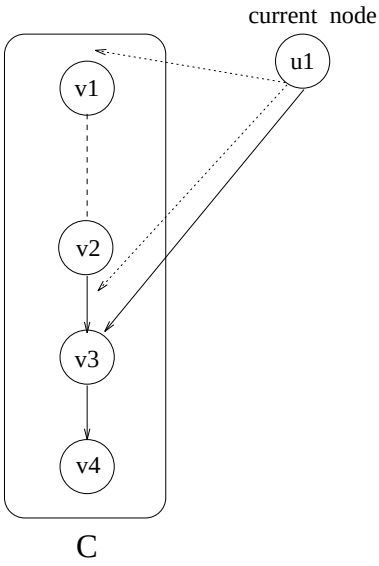


Figure 4. The strategy of edge zeroing.

Figure 4 illustrates the method that could be used for including a current node u_1 in a cluster C. In the figure, it is assumed that v_1 is the first node in C and v_3 is the first immediate successor of u_1 in C (with respect to the sequential ordering). Then u_1 is included in C by placing it either: (1) immediately before v_1, or (2) immediately before v_3. Note if u_1 is inserted before v_1, $f(u_1)$ may increase but the f values of the other nodes in C would not change. On the other hand, if u_1 is inserted before v_3, $f(u_1)$ may decrease but $f(v_1)$ and $f(v_2)$ may increase. In the second case, updating the $f(v_1)$ and $f(v_2)$ will increase the time complexity. Therefore, only the first case is used by CASS-II. Note that we also do not update the l and s values for the nodes in the cluster. If the placement is not *acceptable* (that is, reduce the f values of neither the current node nor the cluster), the current node is placed in a new cluster all to itself.

Figure 5 shows the result of applying the clustering strategy to Figure 3. Current node u_1 will be considered first since it has a higher l value than current node u_2. If u_1 were included in the cluster C_1 containing its dominant successor v_1, $f(u_1)$ would be reduced from 32 to 25 and $f(C_1)$ would not be changed. (For C_1, its f value would be determined by node v_1, whose f value would remain 17 but whose s value would now be $s(u_2)+\mu(u_2)+\lambda(u_2,v_1) = 21+5+4 = 30$.

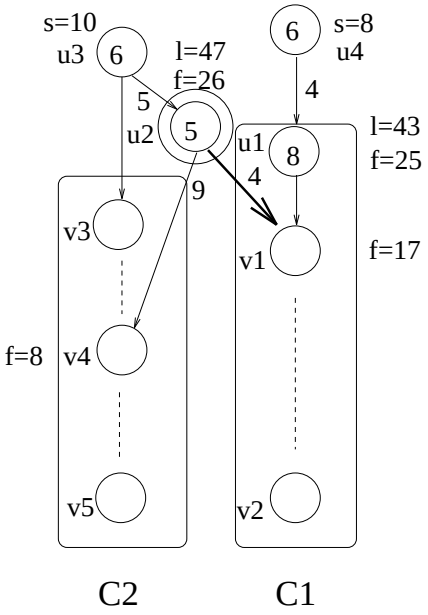


Figure 5. The clustering of the DAG in Figure 3.

Hence, $l(v_1) = 47$.) Thus, the clustering is acceptable and current node u_1 is included in cluster C_1. Next, current node u_2 is considered. One can check that clustering u_2 with C_1 increases $f(u_2)$; hence u_2 is placed in a new cluster all to itself.

The complete algorithm is given as Algorithm CASS-II below. The algorithm maintains a priority queue S consisting of items of the form $[u, l(u), v]$, where u is a current node, $l(u)$ is the l value of u, and v is the dominant successor of u. INSERT$(S, item)$ inserts an item in S and DELETE-MAX-L-VALUE(S) deletes from S the item with the maximum l value. The algorithm returns, for each node v, the cluster $C(v)$ containing it.

1. **Algorithm** CASS-II(G)
2. **begin**
3. **for** each node v **do**
4. compute $s(v)$;
5. **endfor**;
6. **for** each sink node v **do**
7. $f(v) \leftarrow \mu(v)$; $l(v) = s(v) + f(v)$; $C(v) \leftarrow \{v\}$;
8. **endfor**;

9. $S \leftarrow \emptyset$;

10. **while** there are current nodes **do**

11. **for** each new current node u **do**

12. find u's dominant successor v;

13. INSERT$(S, [u, l(u), v])$;

14. **endfor**;

15. $[x, l(x), y] \leftarrow$ DELETE-MAX-L-VALUE(S);

16. **if** edge zeroing is acceptable **then**

17. Zero the edge $e(x, y)$;

18. **else if** y is not a sink node **then**

19. $C(x) \leftarrow \{x\}$;

20. **endif**;

21. **endwhile;**

22. **return** $(\{C(v)|v \in G\})$;

23. **end** CASS-II.

Note that at lines 19-20, if the children of the current node are all sink nodes, each child will be examined without increasing the complexity to see if a cluster merging is acceptable. If it is, then edge zeroing is performed to merge the two clusters. Otherwise, the child (sink node) will be left alone.

Apply the CASS-II to the DAG shown in Figure 2. The resulted clustering is as follows:

$$C_1 = \{1\}, C_2 = \{2, 5, 4, 7, 9\}, C_3 = \{3, 6, 8, 10\}.$$

The makespan of the clustering is 26. On the other hand, applying DSC on the example DAG results in the following clusters:

$$C_1 = \{1\}, C_2 = \{2, 5\}, C_3 = \{3, 6, 10\},$$
$$C_4 = \{4, 7, 9\}, C_5 = \{8\}.$$

The makespan of the clustering produced by the DSC algorithm is 30 by searching both directions (i.e., top-down and bottom-up) of the DAG. Note that in practice, like the DSC algorithm, CASS-II also searches both directions of the DAG and uses the makespan of whichever is better.

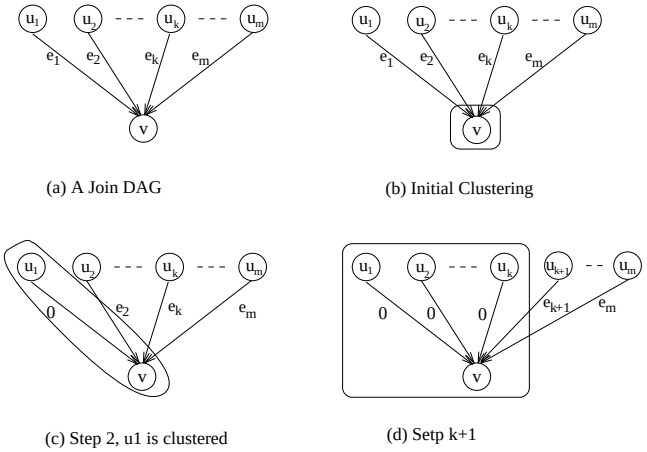
4 Special Cases

In this section, we discuss the performance of CASS-II for fork and join DAGs, and show the optimality of the algorithm for these DAGs.

Figure 6 demonstrates the clustering steps of CASS-II for a join DAG. Let $f(u_i) = \mu_i + \lambda_i + \mu(v)$, for $i = 1, 2, \cdots, m$. Without loss of generality, we assume that the root nodes in the DAG shown in (a) are sorted in a nonincreasing order of the f values (i.e., $f(u_1) \geq f(u_2) \geq \cdots \geq f(u_m)$). Initially, each node is in a unit cluster as shown in Figure 6(b). The l value of v equals to $\max_i\{f(u_i)\}$ which is $f(u_1)$. At step 2 shown in (c), the cluster of v is grown to include node u_1. The current node becomes u_2 at this moment. The cluster will keep growing until the following condition can not be satisfied.

$$\sum_{i=1}^{k} \mu_i \leq \lambda_{k+1}$$

As shown in (d), CASS-II stops at node $vuk + 1$ and the original leftmost scheduled cluster forms a linear chain. The steps applied in join DAGs can be applied to fork DAGs by just simply reversing the fork DAG into a join DAG.



(a) A Join DAG (b) Initial Clustering

(c) Step 2, u1 is clustered (d) Setp k+1

Figure 6. CASS-II clustering steps for a join DAG.

Theorem 1 *CASS-II achieves optimal scheduling for fork and join DAGs.*

5 Experimental Results

For clustering without task duplication, a number of other algorithms have been reported in the literature besides DSC and CASS-II. The most well-known are the algorithms proposed by Sarkar [14], the MCP heuristic of Wu and Gajski [15], and the algorithm of Kim and Browne [5].

All algorithms assume an unbounded number of processors.

Table 1 compares these algorithms with CASS-II with respect to performance guarantee and theoretical runtime. It shows that none of the algorithms have a performance guarantee on general task graphs. Among these algorithms, CASS-II is superior to the others in terms of speed.

Experimental results in [2, 3, 16] have shown that DSC outperforms all the algorithms listed in Table 1, except for CASS-II, in terms of both speed and solution quality. To see how CASS-II compares with DSC, we tested both algorithms on 350 DAGs. The 350 DAGs were divided into 14 groups of 25 DAGs each according to their grain size, as indicated by column 1 of Table 2. Column 2 of the table gives the range of number of nodes for the DAGs in the group. Column 3 and 4 give the average runtimes (in milliseconds) of DSC and CASS-II, respectively, when executed on a Sun Sparc workstation. Column 5 gives the average makespan ratio of DSC over CASS-II. Finally, column 6 gives the average runtime ratio of DSC over CASS-II.

Table 2 indicates that CASS-II is between 3.85 to 5.35 times faster than DSC. Moreover, in terms of solution quality, CASS-II is very competitive: it is better than DSC for grain sizes less or equal to 0.6, and its superiority increases as the DAG becomes increasingly fine grain. For example, for grain sizes equal to 0.1 or less, CASS-II generates makespans which are up to 37% shorter than DSC's. For some DAGs with this grain size, sequentializing a set of tasks (one cluster produced) can achieve a better makespan than executing them in parallel (two or more clusters produced). It is interesting that the case can be detected by CASS-II, but DSC seems not to find the necessity for serial execution and produces several clusters instead of one. On the other hand, for task graph with grain size 0.6 or greater, DSC becomes competitive and in fact even outperforms CASS-II, although by no more than 3% (grain size = 1.0).

6 Conclusions

In this paper, we have presented a simple task clustering algorithm without task duplication. Unlike the DSC algorithm, CASS-II uses only limited ''global'' information and does not recompute the critical path in each refinement step. Therefore, the algorithm runs in $O(|E| + |V|lg|V|)$ which is faster than $O((|V| + |E|)lg|V|)$ of the DSC algorithm. Unfortunately, we are unable to find a provable bound on the solution quality produced by the CASS-II. This is very difficult because it is known that for general task graphs, clustering without task duplication remains NP-hard even when the solution quality is relaxed to be within twice the optimal solution. However, we exhibit optimal schedules for the special cases such as join and fork DAGs.

We have also compared CASS-II with the DSC algorithm of clustering without task duplication and showed that CASS-II is 3 to 5 times faster than DSC. For fine grain DAGs (granularity = 0.6 or less), CASS-II consistently gives better schedules. For DAGs with grain size 0.6 or greater, DSC becomes comparable to CASS-II, and in some cases even strictly better, but by no more than 3%.

Details of the implementation of the algorithm and its complexity analysis, as well as the proof of the theorem, can be found in the full paper [9].

References

[1] P. Chretienne. Complexity of tree scheduling with interprocessor communication delays. Tech. Report M.A.S.I. 90.5, Universite Pierre et Marie Curie, 1990.

[2] A. Gerasoulis and T. Yang. Clustering task graphs for message passing architectures. *Proc. Int. Conf. on Supercomputing*, pp. 447-456, 1990.

[3] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and distributed Systems*, 4:6:686--701, June 1993.

[4] J. J. Hwang, Y. C. Chow, F. D. Anger, and C. Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18:2:244--257, April 1989.

[5] S. Kim and J. C. Browne. A general approach to mapping of parallel computation upon multiprocessor architectures. *Proc. Int. Conf. on Parallel Processing*, 3:1--8, 1988.

[6] C. Y. Lee, J. J. Hwang, Y. C. Chow, and F. D. Anger. Multiprocessor scheduling with interprocessor communication delays. *Oper. Res. Lett.*, 7:3:141--147, 1988.

[7] J.-C. Liou. *Grain-Size Optimization and Scheduling for Distributed Memory Architectures.* PhD thesis, Department of Electrical and Computer Engineering, New Jersey Institute of Technology, 1995.

[8] J.-C. Liou, M. A. Palis, D. S. Wei. Performance analysis of task clustering heuristics for scheduling static DAGs on multiprocessor. To appear in the Journal of Parallel Algorithms and Applications.

	Sarkar	MCP	Kim & Browne	DSC	CASS-II
Join/Fork	no	no	no	optimal	optimal
General DAGs	no	no	no	no	no
Runtime	$O(e(n+e))$	$O(n^2 lgn)$	$O(n(n+e))$	$O((n+e)lgn)$	$O(e+nlgn)$

Table 1. A comparison of static clustering algorithms without task duplication. n = no. of tasks. e = no. of edges.

Grain Size	# of Tasks Min - Max	DSC Avg Runtime	CASS-II's Avg Runtime	M(DSC,CASS-II)	T(DSC,CASS-II)
0.1	85-988	544	131	1.37	4.15
0.2	129-987	698	156	1.10	4.47
0.3	86-988	684	149	1.05	4.59
0.4	87-989	566	129	1.00	4.39
0.5	88-990	532	123	1.01	4.33
0.6	89-949	533	116	1.00	4.59
0.7	90-997	615	141	1.00	4.36
0.8	89-992	524	136	0.99	3.85
0.9	93-993	652	141	0.98	4.62
1.0	90-992	687	133	0.97	5.17
2.0	91-993	930	178	1.00	5.22
3.0	92-994	884	171	1.00	5.17
4.0	93-953	851	159	1.00	5.35
5.0	94-995	737	155	1.00	4.75

Table 2. Experimental results of CASS-II and DSC algorithm run on a Sun Sparc workstation.

[9] J.-C. Liou, M. A. Palis. *Grain-Size Optimization of parallel programs*. Submitted to *IEEE Transactions on Parallel and Distributed Systems.*

[10] C. McCreary and H. Gill. Automatic determination of grain size for efficient parallel processing. *Comm. ACM*, pages 1073--1078, Sep. 1989.

[11] M. A. Palis, J.-C Liou, and D. S. Wei. Task clustering and scheduling for distributed memory parallel architectures. *IEEE Transaction on Parallel and Distributed Systems*, 7:1:46-55, 1996.

[12] C. H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM Journal on Computing*, 19:2:322--328, April 1990.

[13] H. E. Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, 9:138--153, 1990.

[14] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors.* MIT Press, 1989.

[15] M. Y. Wu and D. D. Gajski. A programming aid for hypercube architectures. *Journal of Supercomputing*, 2:349--372, 1988.

[16] T. Yang. *Scheduling and Code Generation for Parallel Architectures*. PhD thesis, Department of Computer Science, Rutgers University, 1993. Ph.D. Thesis, Tech. Report DCS-TR-299.

[17] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5:9:951--967, 1994.