ISSN:0128-680
Pertanika J. Sci. & Technol. 3(2): 241-259 (1995)

Top-down Heuristic for Finding Optimal Grain Size of Parallel Tasks

Md Yazid Mohd Saman and D.J. Evans[1]

Department of Computer Science
Faculty of Science and Environmental Studies
Universiti Pertanian Malaysia
UPM 43400 Serdang, Selangor, Malaysia

[1]Parallel Algorithms Research Centre
Loughborough University of Technology
Loughborough,Leics, United Kingdom

Received 10 March 1994

ABSTRACT

In order to have an optimal execution time of a program running on a multi-processor system, the program has to be partitioned into concurrent tasks. Partitioning of programs to grain size suitable for parallel execution is an NP-complete problem but near-optimal time can be derived. This paper discusses a heuristic to determine the near-optimal grain size of parallel tasks that will give the best execution time. The effects of communication overheads between the different processors are examined. The heuristic developed is capable of balancing between maximizing parallelism and minimizing overheads.

ABSTRAK

Untuk mendapatkan masa perlaksanaan yang optimum bagi satu aturcara yang dilaksanakan di sistem multi-pemproses, aturcara tersebut mestilah dipetakan menjadi tugas serentak. Pemetakan aturcara untuk mendapatkan saiz butir yang sesuai untuk perlaksanaan serentak adalah satu masalah NP-complete tetapi masa yang hampir-optimum boleh diperolehi. Kertas ini membincangkan satu heuristik untuk menentukan saiz butir yang hampir-optimum untuk tugas serentak yang akan memberikan masa perlaksanaan yang terbaik. Kesan overhed komunikasi di antara pelbagai pemproses dikaji. Heuristik yang dibina berkemampuan untuk mengimbang diantara keserentakan yang maksimum dan overhed yang minimum.

Keywords: grain size, scheduling, shared-memory computers, Bernstein sets

INTRODUCTION

In a multi-processor computer, each processor can execute different parts of a program in parallel; the task of programming in parallel has increased. In writing programs for a parallel computer, the programmer must be able to carry out analysis to identify any parallelizable parts and to ensure that they are free from any data dependences. Task mapping or scheduling is a process of assigning the concurrent tasks in a program to

a target parallel machine (Kruatrachue and Lewis 1988; Polychronopoulos 1988). Its main objective is to attain an optimal overall execution time for the program. This depends on the size of the tasks, their communication times, the number of processors and the strategy of task assignment to processors.

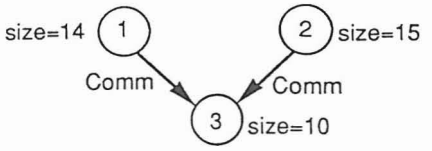
This paper presents a description of a heuristic to identify automatically the size of task granularity that gives the optimal execution performance of a sequential program. The main process is to partition the program into parallel tasks that can be executed on an n-processor shared-memory parallel machine such as the Sequent Symmetry (Osterhaug 1987). This is not an easy task because the boundaries for partitioning the program are not well defined except in cases where they are predetermined to be of certain language constructs (such as the loop body or basic blocks).

DETERMINATION OF TASK GRANULARITY

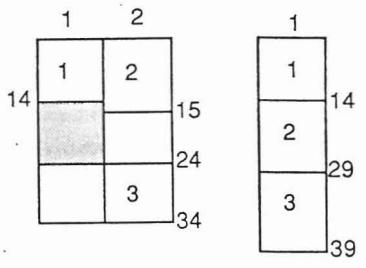
The determination of task granularity is a partitioning problem. It is a process of breaking down a program into a set of tasks suitable for parallel execution (Girkar and Polychronopoulos 1988; Kruatrachue and Lewis 1988; Polychronopoulos 1988; McCreary and Gill 1989; Sarkar 1989; Kwan *et al.* 1990). A grain is defined as a module containing one or more tasks that has to be executed in a sequential manner by a single processor. Polychronopoulos (1988) defines the size of a task derived entirely from the syntax of the underlying language. For languages such as Pascal, C and Fortran, the tasks are the loop body, procedure calls and basic assignment blocks (BAS). Williams (1978) has limited the maximum size of a task to be a group of statements with 15 variables; it is not necessarily a BAS. This, however, does not ensure an optimal task size. In this paper, a task can be a statement, a block of statements delimited by begin-end block as in Pascal, a loop or a procedure call.

The problem is to determine the best task size that will give the shortest execution time. A large grain size will limit potential parallelism. Small grain size, however, will result in greater communication overheads and may cause execution time degradation. This needs a good automatic merging (or packing) strategy to decide which tasks are best executed on the same processor. Together with the scheduling process, they will have to balance between the possible parallelism and the communication overheads to achieve the best grain size. It has been shown that the general solution to this granularity problem is NP-complete but a near-optimal solution to a subproblem can be determined (Garey and Johnson 1979; Kruatrachue and Lewis 1988; Sarkar 1989). In this paper, a heuristic is developed to determine this near-optimal task size.

The way tasks are merged is crucial. Sometimes, merging produces groups of tasks which give degrading execution time. This is illustrated in *Fig. 1.* With a low communication time (such as 10 unit time in *Fig. 1*), merging will degrade the performance (as shown in *Fig. 1(b)*). On the other hand, merging improves the performance if high communication time is involved *(Fig. 1(c))*. It shows that the main factors governing this merging operation are the task sizes, the communication times and the dependence relations. It is essential to determine before merging, the effects of these factors. If it proves to degrade the execution time, then the tasks should be left unmerged. However, since the study described in this paper involves heuristics, then improved solutions cannot always be guaranteed. This is

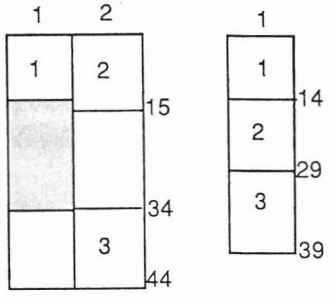


(a) A stanza with two predecessors



(i) unmerged (ii) merged

(b) Merging degrades the performance if Comm=10



(i) unmerged (ii) merged

(c) Merging improves the performance if Comm=20

Fig. 1 Effects of task and communication size on merging operation

because the merging operation restructures the dependence relations of the new sets of tasks and this may create less parallelism.

Most of the work done by researchers assumes a parallel program in the form of a task graph. Kwan *et al.* (1990) uses critical path analysis to improve the performance of parallel programs. Aggregating tasks by forming clans as grains has been proposed by McCreary and Gill (1989). These clans can then be assigned to parallel processors to achieve an optimal execution time. Sarkar (1989) proposes two models for partitioning and scheduling task graphs, called the macro-dataflow model (compile-time partitioning and run-time scheduling) and the compile-time scheduling model (partitioning and scheduling at compile-time).

Polychronopoulos (1988) also uses the task graphs to model the program to be scheduled. A critical process size (CPS) is estimated for each task and the size of processes are determined, based on this CPS. The CPS is the minimum size of a process whose execution time is equal to the overhead that it incurs during scheduling. Kruatrachue and Lewis (1988) have developed a method to optimize parallel programs called grain packing which reduces total parallel execution time by balancing the sequential execution time and communication time. Their duplicating scheduling heuristic duplicates tasks where necessary to reduce overall communication delays and maximizes parallelism at the same time. Bieler (1990) has studied the partitioning of parallel programs written in UNITY by developing the d-graphs of the programs. These d-graphs are then mapped on a parallel processor. D-graphs are graphs with two edges, weak edges and solid edges. Statements connected by weak edges are suitable for allocation in different processors.

THE TOP-DOWN MERGING HEURISTIC

In this paper, concurrent tasks are derived by partitioning a sequential program targeted for execution for a shared-memory computer such as the sequent symmetry (Osterhaug 1987). Therefore, any implicit parallelism that exists will be first determined. Williams (1978) has developed an approach to detect implicit parallelism in a sequential program. The technique is based on Bernstein sets (BSs) (Bernstein 1966). She develops a set of conditions (termed Bernstein tests (BTs) in this paper) to determine whether the tasks can be executed in parallel or not. In this paper, the BTs are used to derive the predecessor tasks that another task depends on for data during execution (i.e., the data dependence relations of tasks). The definitions of tasks (called stanzas in Williams (1978)), Bernstein sets and Bernstein tests are given in the Appendix. Detailed discussions on this topics are given in Md Yazid (1993b).

Another process involved in this heuristic is concurrent tasks scheduling which is a widely studied research topic (Sahni 1984; Bokhari 1988;

Duda 1988; Kruatrachue and Lewis 1988; Polychronopoulos 1988; Sarkar 1989; Md Yazid 1994). Its main goal is to have an optimal execution time, i.e., the shortest possible execution time for the program on a certain parallel processor system. However, communication overheads have become the main problem in performing this function.

In order to reduce communication overheads, some tasks have to be merged so that they will be executed on the same processor. This is performed by the Merger module which forms bigger tasks from the basic ones. It is assumed that this process will add the execution times of the tasks but still maintain the same communication time (Kruatrachue and Lewis 1988).

The heuristic is called Top-down Merging Heuristic. It is based on the following principles. Given a task S and its predecessors tasks PSs (as shown in *Fig. 2*), starting from the top level tasks:

a. they will be merged if none of the PSs has been merged with other tasks, because PSs could also be predecessors of other tasks.

b. let PSi be the largest of all PSs and comm(PSs) be the communication times for PSs to transfer data to any other tasks. Then for all PSj (where $i \neq j$), they are merged with PSi if:

$$comm(PSj)) > (size(PSi)$$

Otherwise they are left unmerged. This is to ensure that the merge operation does not give a new task whose execution time is higher than that of the unmerged tasks.

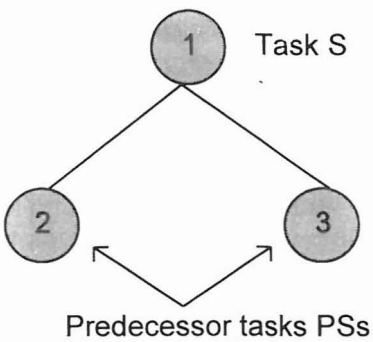


Fig. 2. A task S and its predecessor tasks PSs

When the BSs of two tasks are merged, it produces new BSs whose contents depend on which sets the variables are members of before merging. For example, if a variable v is a member of the X set in BS1 and it is also a member in the W set in BS2, then it will be included in the Z set of the resulting BSs. *Fig. 3* shows the algorithm to merge any two BSs of tasks i and j resulting in task k. It should be noted that MERGE(S1,S2) will not necessarily give the same result as MERGE(S2,S1).

TASK MERGING ALGORITHM

```
INPUT  : Tasks Si (with Wi,Xi,Yi,Zi sets) and Sj (with Wj,Xj,Yj,Zj sets)
OUTPUT : Task Sk (with Wk,Xk,Yk,Zk sets)
BEGIN
        FOR each variable v in the Wi,Xi,Yi and Zi sets of Si, search for it in task Sj.
        BEGIN
            IF v is an element of Wi
            IF v an element of Wj or NOT found,
                THEN v is an element of Wk
                ELSE v is an element of Yk
            IF v is an element of Xi
                IF v an element of Xj or NOT found,
                THEN v is an element of Xk
                ELSE  v is an element of Zk
            IF  v is an element of Yi
            THEN v is an element of Yk
            IF v is an element of Zi
            THEN v is an element of Zk
        END
        FOR all variable v in task Sj NOT found in task Si
        then v is an element of its original set of task Sk
        END
```

Fig. 3. The merge algorithm

Once the merging process has been completed, an average granularity size of the new tasks and new dependence relations is determined. Then, another process of scheduling is carried out to determine a new schedule with another estimated parallel execution time and speed-up factor. This process of merging and scheduling is repeated until no more merging operations are carried out. A schedule with the best execution time is taken as the near-optimal execution time and its average task size is the near-optimal average grain size. The description of the scheduling heuristic is given in Md Yazid (1994). *Fig. 4* shows the whole process of repeated scheduling/merging to find the near-optimal task granularity.

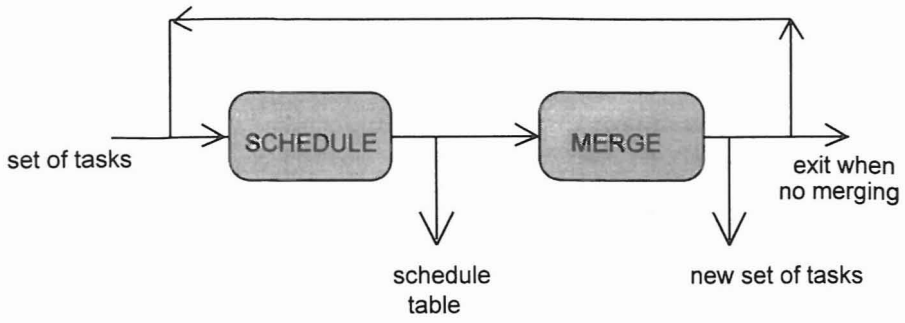


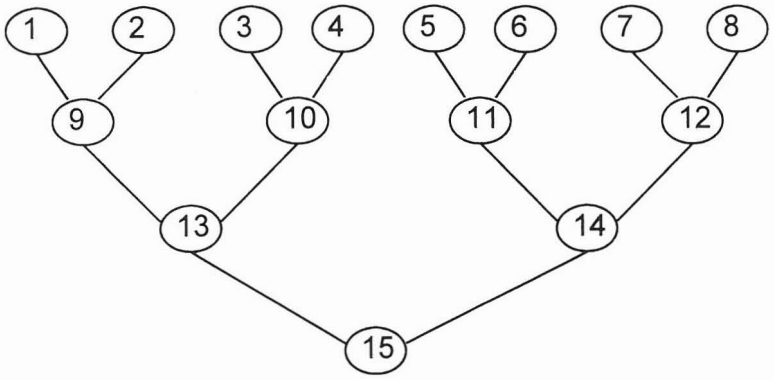
Fig. 4. Repeated processes of scheduling and merging to determine granularity size

EXAMPLE OUTPUT

In this section, examples of the output of the scheduling and merging heuristic are presented. *Fig. 5* shows a simple program that performs Σ(ai*bj) and its graphical representation in the form of a task graph depicting the inter-dependence of the statements.

```
program SUM;
begin
    n1 := a1*b1;
    n2 := a2*b2;
    n3 := a3*b3;
    n4 := a2*b4;
    n5 := a5*b5;
    n6 := a6*b6;
    n7 := a7*b7;
    n8  := a8*b8;
    n9  := n1+n2;
    n10 := n3+n4;
    n11 := n5+n6;
    n12 := n7+n8;
    n13 := n9+n10;
    n14 := n11+n12;
    n15 := n13+n14;
end.
```

(a) A simple program SUM



(b) The task graph

Fig. 5. SUM and its task graph

Fig. 6 shows information for the Bernstein sets (BSs) of each task, derived from each statement in the program. Note that the communication time for each task has been fixed as 10 units during execution. This can be changed to other values. The Bernstein tests (BTs) are applied on the BSs to produce the dependence table and the contemporary table as shown in *Fig. 7*. The derivation of BSs and the implementation of the BTs have been described in Md Yazid (1993a).

Task	W sets	X sets	Y sets	Z sets	Exec Time	Comm Time
1	a1 b1	n1	-	-	11	10
2	a2 b2	n2	-	-	11	10
3	a3 b3	n3	-	-	11	10
4	a2 b4	n4	-	-	11	10
5	a5 b5	n5	-	-	11	10
6	a6	n6	-	-	11	10
	a7 b7	n7	-	-	11	10
8	a8 b8	n8	-	-	11	10
9	n1 n2	n9	-	-	2	10
10	n3 n4	n10	-	-	2	10
11	n5 n6	n11	-	-	2	10
12	n7 n8	n12	-	-	2	10
13	n9 n10	n13	-	-	2	10
14	n11 n12	n14	-	-	2	10
15	n13 n14	n15	-	-	2	10
Total sequential time= 102						

Fig. 6. Bernstein sets of the SUM program

Task no.	{ Predecessor tasks }
1	- ø
2	- ø
3	- ø
4	- ø
5	- ø
6	- ø
7	- ø
8	- ø
9	- { 1 2 }
10	- { 3 4 }
11	- { 5 6 }
12	- { 7 8 }
13	- { 9 10 }
14	- {11 12 }
15	- {13 14 }

(a) Dependence Table

task	{ concurrent tasks }
1	- { 2 3 4 5 6 7 8 13 14}
2	- { 3 4 5 6 7 8 13 14}
3	- { 4 5 6 7 8 9 14}
4	- { 5 6 7 8 9 14}
5	- { 6 7 8 9 10 14}
6	- { 7 8 9 10 14}
7	- { 8 9 10 11 15}
8	- { 9 10 11 15}
9	- {10 11 12 15}
10	- { 11 12 15}
11	- { 12 13}
12	- { 13}
13	- { 14}
14	- ø
15	- ø

(b) Contemporary Table

Fig. 7. Dependence table and contemporary table generated by the Bernstein tests (BTs)

The diagrams in *Fig. 8* show the first merging operation of tasks 9, 10 and 13 to form a new task 9 and tasks 11, 12 and 14 to form a new task 10. They are merged after considering the effects of their sizes and the communication overheads. Tasks 1, 2 and 9 or tasks 3, 4 and 10 or tasks 5, 6 and 11 or tasks 7, 8 and 12 have not been merged because merging will degrade their execution times.

Fig. 9 shows a new set of tasks produced after the first merging operation. Note that tasks 1 to 8 are the same as previous tasks and task 11 is the same as task 15. Task 9 and 10 are the new merged tasks. Task 9 in this table comes from tasks 9, 10 and 13 and task 10 from 11, 12 and 14. The new dependence table and the contemporary table after the first merging process are shown in *Fig. 10.*

In the second merging operation, tasks 9, 10 and 11 are merged. *Fig. 11* gives an illustration of their merging. The new set of tasks after the second merging is shown in *Fig. 12.* In *Fig. 13* are the revised dependence table and contemporary table after the second merging.

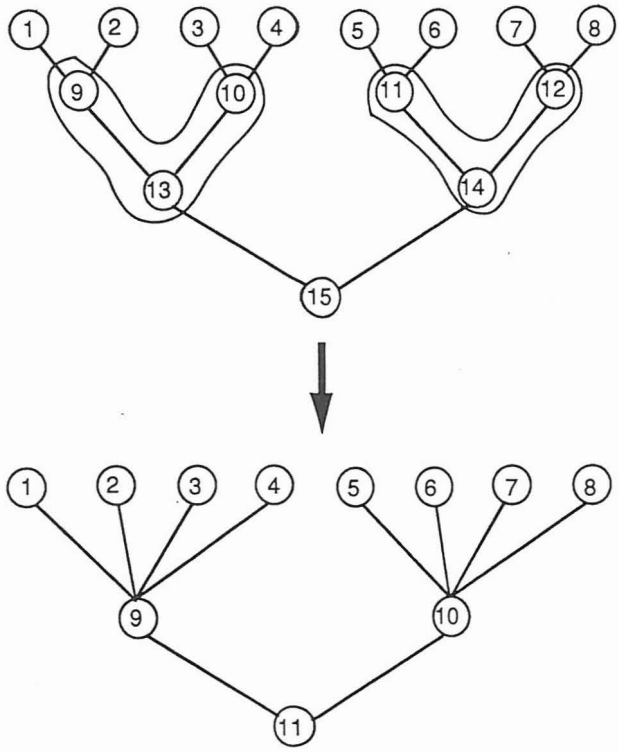


Fig. 8. First merging process of SUM (communication time = 10 units)

DISCUSSION

As a conclusion of the merging process, the heuristic arrives at the final result after performing two operations. The third merging does not produce any new tasks and hence it stops. Table 1 shows the performances produced by the schedules generated before and after the two merging operations. It indicates that the best execution time of 2.91 speed-up value comes with a schedule for a 8-processor parallel machine after the second merging operation (column 4 row 7). The average granularity size is 11.33 which has nearly doubled from the original size of 6.80. *Fig. 14* shows the grains of tasks that give the best schedule. The performances shown in Table 1 are for the program wiht a communication time of 10 units. The diagrams in *Fig. 15* show the effects of a higher communication time on the merging operation. In this case, 20 units is fixed as the communication time, resulting in different groups of tasks being merged.

In the first merging operation, four groups of tasks are merged. If this is compared with the diagrams in *Fig. 7* and *10*, different groups are being merged. In the second merging operation, only one group is merged.

Task	W sets	X sets	Y sets	Z sets	Exec Time	Comm Time
1	a1 b1	n1	-	-	11	10
2	a2 b2	n2	-	-	11	10
3	a3 b3	n3	-	-	11	10
4	a2 b4	n4	-	-	11	10
5	a5 b5	n5	-	-	11	10
6	a6 b6	n6	-	-	11	10
7	a7 b7	n7	-	-	11	10
8	a8 b8	n8	-	-	11	10
9	n1 n2 n3 n4	n13	- n10	n9	6	10
10	n5 n6 n7 n8	n14	-	n11 n12	6	10
11	n13 n14	n15	-	-	2	10
	Total sequential time = 102					

Fig. 9. A new set of tasks after the first merging

Table 2 shows its performances where the best schedule has a speed-up value of 2.04 on a 4-processor machine. *Fig. 16* illustrates the grains of tasks for this particular schedule. The speed-up is lower than with 10 units of communication time (see *Fig. 14*(b)) and it is achieved after the second merging. However, the average task size (that is 20.40) is about three times bigger than the original size (that is 6.80) as shown in Table 2.

Task no.	{ Predecessor tasks }	task	{ concurrent tasks }
1	-ø	1	- { 2 3 4 5 6 7 8 11 }
2	-ø	2	- { 3 4 5 6 7 8 11 }
3	-ø	3	- { 4 5 6 7 8 11 }
4	-ø	4	- { 5 6 7 8 11 }
5	-ø	5	- { 6 7 8 9 }
6	-ø	6	- { 7 8 9 }
7	-ø	7	- { 8 9}
8	-ø	8	- { 9 }
9	- { 1 2 3 4 }	9	- { 10}
10	- { 5 6 7 8 }	10	- ø
11	- { 9 10 }	11	- ø
(a) Dependence Table		(b) Contemporary Table	

Fig. 10. New dependence table and contemporary table after the first merging

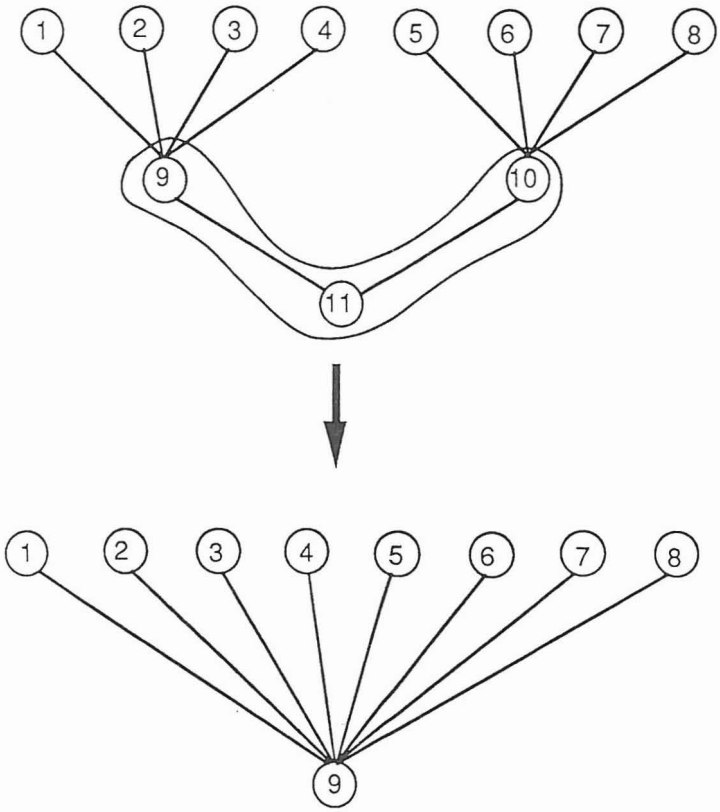


Fig. 11. Second merging operation of SUM (communication time = 10 units)

Task	W sets	X sets	Y sets	Z sets	Exec Time	Comm Time
1	a1	n1	-	-	11	10
	b1					
2	a2	n2	-	-	11	10
	b2					
3	a3	n3	-	-	11	10
	b3					
4	a2	n4	-	-	11	10
	b4					
5	a5	n5	-	-	11	10
	b5					
6	a6	n6	-	-	11	10
	b6					
7	a7	n7	-	-	11	10
	b7					
8	a8	n8	-	-	11	10
	b8					
9	n1	n15	-	n13	14	10
	n2			n14		
	n3			n9		
	n4			n10		
	n5			n11		
	n6			n12		
	n7					
	n8					
	Total sequential time =				102	

Fig. 12. The new set of tasks after second merging

CONCLUSION

This paper has discussed the problem of finding the optimal grain size for parallel tasks. It is known to be an NP-complete problem but near-optimal solutions can be derived as this paper has shown. The process of merging of tasks to form tasks with bigger granularity has been used in order to find a faster execution time. This needs to be performed carefully due to the presence of inter-processor communication overheads. An improved execution time is only possible if the communication time that exists after merging is shorter with respect to the task size and the program still has adequate parallelism. This, however, is not always the

Task no.	{ Predecessor tasks }
1	- ø
2	- ø
3	- ø
4	- ø
5	- ø
6	- ø
7	- ø
8	- ø
9	- { 1 2 3 4 5 6 7 8 }

(a) Dependence Table

task	{ concurrent tasks }
1	- { 2 3 4 5 6 7 8 }
2	- { 3 4 5 6 7 8 }
3	- { 4 5 6 7 8 }
4	- { 5 6 7 8 }
5	- { 6 7 8 }
6	- { 7 8 }
7	- { 8 }
8	- ø
9	- ø

(b) Contemporary Table

Fig. 13. New dependence table and contemporary table after the second merging

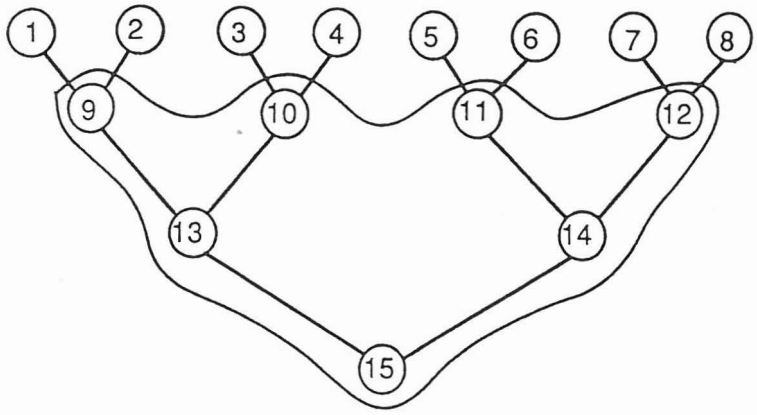
TABLE 1
Performances of SUM with 10 units of communication time

Number of processors	Before merging	First merging	Second merging
2	1.65	1.65	1.50
3	2.00	2.00	1.79
4	2.12	2.55	2.22
5	2.17	2.55	2.22
6	2.55	2.55	2.22
7	2.17	2.62	2.83
8	2.17	2.62	2.91
average granularity	6.80	9.27	11.33

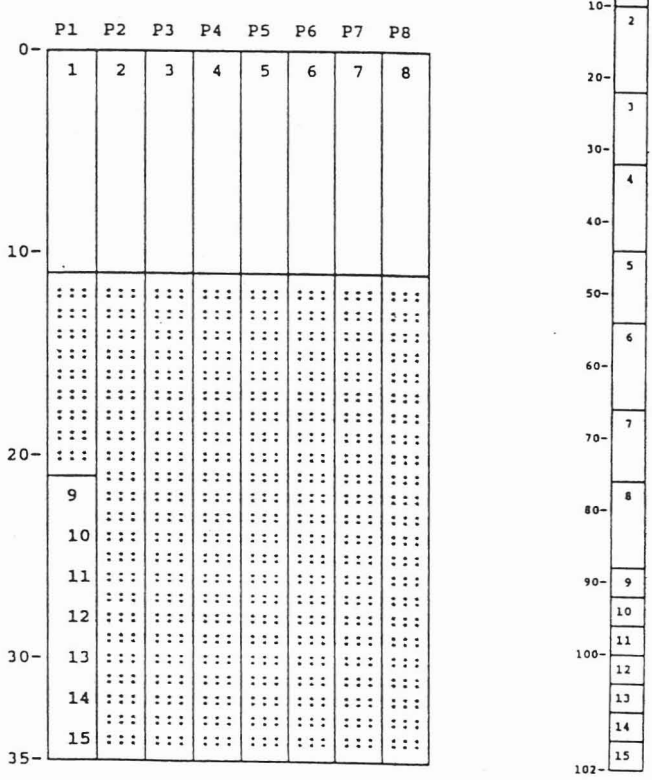
case. Merging can also cause execution time degradation. Thus, the heuristic proposed in this paper only merges tasks if it proves to give better results.

ACKNOWLEDGEMENT

Financial assistance to carry out the above project was provided by Universiti Pertanian Malaysia.



(a) 9 grains of stanzas



(b) Schedule for the grains (speed-up=2.91)

Fig. 14. Schedules for SUM

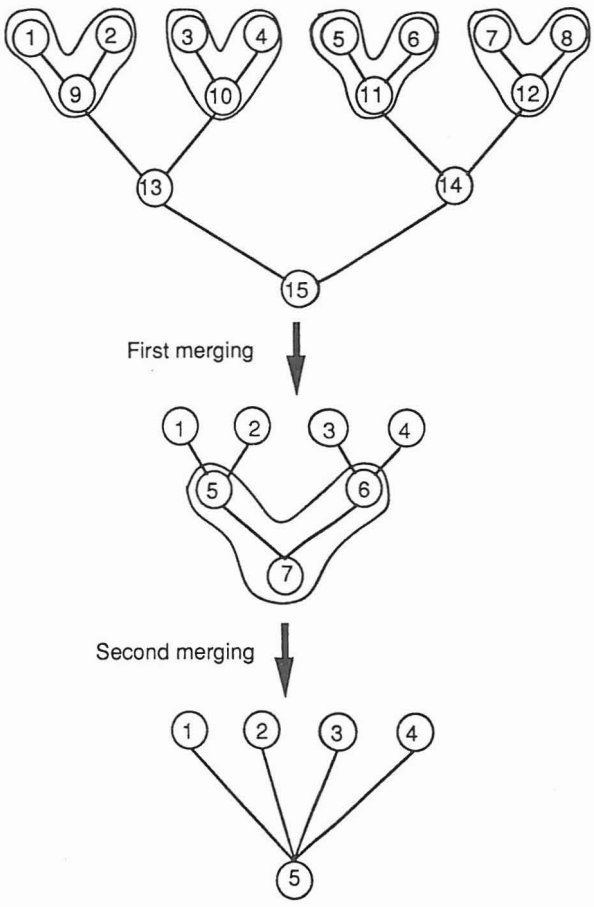
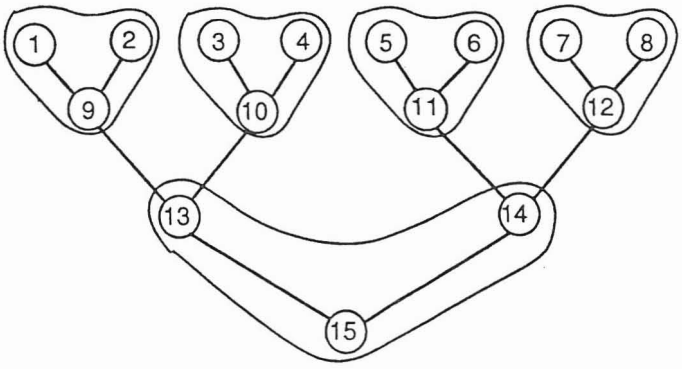


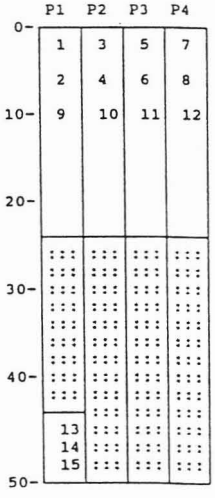
Fig. 15. Merging of tasks from SUM with 20 units of communication time

TABLE 2
Performances of SUM with 20 units of communication time

Number of processors	Before merging	First merging	Second merging
2	1.42	1.42	1.38
3	1.50	1.50	1.89
4	1.50	1.50	2.04
5	1.32		
6	1.73		
7	1.32		
8	1.32		
average granularity	6.80	14.57	20.40



(a) grains of stanzas



(b) best schedule (speed-up = 2.04)

Fig. 16. Grains of tasks for sum with the communication time = 20 units

REFERENCES

BERNSTEIN, A.J. 1966. Analysis of programs for parallel processing. *IEE Trans. on Elec. Comp.* 1 EC-15: 757-763.

BIELER, F. 1990. Partitioning programs into processes. In: *Proceedings of Joint International Conference on Vector and Parallel Processing,* CONPAR 90-VAPP IV. p. 513-524. Springer-Verlag.

BOKHARI, S.H. 1988. Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Trans. on Computers* **37(1)**: 48-57.

DUDA, A. 1988. On the tradeoff between parallelism and communication. In: *Proceedings of the 4th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation.* p. 323-334.

GAREY, M.R. and D.S. JOHNSON. 1979. *Computers and Interactability: A Guide to the Theory of NP-completeness.* San Fransisco: WH Freeman.

GIRKAR, M.B. and C.D. POLYCHRONOPOULOS. 1988. Partitioning programs for parallel execution In: *Supercomputing ACM Proceedings of International Conference,* St. Malo, France, p. 216-229.

KRUATRACHUE, B. and T. LEWIS. 1988. *Grain Size Determination for Parallel Processing. IEEE Software.* p: 23-32.

KWAN, A.W., L. BIC and D.D. GAJSKI. 1990. Improving parallel program performance using critical path analysis. In: *Languages and Compilers for Parallel Computing* ed. D. Gelernter, A. Nicolau and D Padua. London: Pitman. p. 358-373.

MCCREARY, C. and H. GILL. 1989. Automatic determination of grain size for efficient parallel processing. *CACM* **32-9:** 1073-1078.

MD YAZID MOHD SAMAN. 1993a. Automatic paralleization of programs. PhD thesis, Loughborough University of Technology, UK.

MD YAZIZ MOHD SAMAN. 1993b. The Bernstein method for data dependence analysis. Technical report no. 3, Dept. of Computer Science, UPM.

MD YAZID MOHD SAMAN. 1994. A scheduling heuristic for a shared-memory computer. Technical report, Department of Computer Science, UPM [in preparation]

OSTERHAUG, A. 1987. *Guide to Parallel Programming.* 2nd edn. Sequent Computer Systems.

POLYCHRONOPOULOS, C.D. 1988. *Parallel Programming and Compilers.* Bosten Mass: Kluwer.

SAHNI, S. 1984. Scheduling multi-pipeline and multi-processor computers. *IEEE Trans. on Computers.* **C-33. 7:** 637-645.

SARKAR, V. 1989. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* London: Pitman.

WILLIAMS, S.A. 1978. Approaches to the determination for parallelism for computer programs. PhD thesis, Loughborough University of Technology, UK..

APPENDIX

BERNSTEIN SETS AND BERNSTEIN TESTS

The Bernstein method is based on the set formation containing variables in programs. The sets, called the Bbernstein sets (BSs), show how the variables are being used, i.e., fetched and stored. The BSs consist of W, X, Y and Z sets.

DEFINITION 1.

(i) A task is either a single program statement or a group of statements appearing adjacently in a computer program and intended to be executed one after the other.

(ii) Bernstein sets (BSs) consist of four sets defined as follows:

a. W set - set of variables fetched during execution of task
b. X set - set of variables stored during execution of task
c. Y set - set of variables which involves a fetch and one of the succeeding operations is a store
d. Z set - set of variables which involves a store and one of the succeeding operations is a fetch

DEFINITION 2.

Bernstein tests (BTs) between two tasks i and j, are tests to determine whether they can be run concurrently or not, i.e., if they satisfy all of the following three conditions:

$(Xi \cup Yi \cup Zi) \cap (Wj \cup Yj \cup Zj) = \emptyset$
$(Wi \cup Yi \cup Zi) \cap (Xj \cup Yj \cup Zj) = \emptyset$
$(Xi \cup Yi \cup Zi) \cap (Xj \cup Yj \cup Zj) = \emptyset$

The BTs are for shared-memory computers only. The operators"$\cup$" and"$\cap$" are set operators for 'union' and 'intersection' respectively. The symbol "$\emptyset$" denotes an empty or null set.