数理モデル化と問題解決　40－１
（２００２．６．２６）

An Improvement of Program Partitioning Based Genetic Algorithm

Masami Takata, Hayaru Shouno, and Kazuki Joe
Graduate School of Human Culture
Nara Women's University
Nara city, JAPAN

Abstract *We propose a sorting rule that improves a genetic algorithm based program partitioning algorithm, and evaluate the effectiveness by experiments. The sorting rule is sensitized the order of nodes of a given task graph. Hence, it is necessary to change the node number to make effective use of the sorting rule. Several variations of the method are investigated and experimentally evaluated. Approximate solutions that provide a sufficient practical partitioning are obtained using the accelerated sorting method, and execution times and error decreased considerably by changing node numbers of the task graph.*

Keywords: program partitioning, parallel program, task graph, genetic algorithm

1 Introduction

To execute numerical simulations in reality, we are working for the development of an automatic parallelizing compiler *PROMIS-NWU* [8]. The *PROMIS-NWU* is an extension of the *PROMIS* [4] to support distributed memory environments.

To develop parallelizing compilers for distributed memory parallel computers, data partitioning should be optimized as well as parallelization, since both partitioning are known as NP-complete problems [1]. However, it is difficult to develop an algorithm to solve the problems, so we only treat a parallelization algorithm in this paper.

Girkar *et al.* [2] had proposed a well-known program partitioning algorithm based a branch and bound method. This requires huge memory capacity besides long calculation time, since the combinatorial explosion occurs. To avoid this explosion, we have proposed several heuristic and edge sorting methods [6] [7]. Nevertheless some large program partitioning could not be performed because of lack of memory. That is why a genetic algorithm (GA) based program partitioning algorithm was proposed [5]. In this paper, to improve the solution by [5], we propose several coding methods.

In section 2, we explain Girkar's algorithm. In section 3, we describe some definitions for GA, and propose several gene coding methods. In section 4, we evaluate the result of proposed gene coding methods.

2 Program Partitioning Algorithm

In general, a program can be transformed into an acyclic weighted directional task graph $G = (N, E)$, where N and E indicate the set of all nodes and edges in the graph respectively. Each $n \in N$ corresponds to a task of the program and is assigned with a sequential number (starting from 1). An edge $e = (n_i, n_j) \in E$ $(n_i < n_j)$ indicates a dependency from node n_i to node n_j. Costs $t(n)$ and $c(e)$ are the execution and the communication time respectively.

The Girkar's algorithm [2] has three kind of conditions for edges. The first is *Inter Partitioning Edge (Inter-PE)* that is an edge between nodes assigned different processors. The second is *Intra Partitioning Edge (Intra-PE)* that is an edge between nodes assigned the same processor. The other edges are called *Unexamined Edge (U-E)*. Each condition set is sorted in the descending order of $t(n_i)+t(n_j)+c(e)$. When the algorithm is terminated, a result graph is shown in Fig.1.

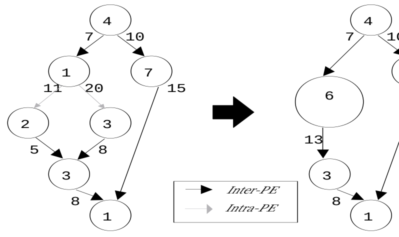

Figure 1: Example for program partitioning

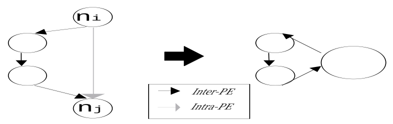

Figure 2: Example of deadlock

Let $P = \langle n_1, ..., n_m \rangle$ be a path, which is made with *Inter-PE*s and some nodes. The cost T_τ is given as $T_\tau = \Sigma_{n_i \in P} t(n_i) + \Sigma_{e_i \in P} c(e_i)$. The critical path is defined as the largest T_τ among the whole paths in the task graph.

Fig.2 shows a breaking out of deadlock, and a partial configuration is neglected.

3 GA based Program Partitioning Algorithm

Since Girkar's algorithm [2] is one of the enumerative methods, the algorithm can not obtain the optimal partitioning of a large task graph because of the combinatorial explosion.

Saito *et al.* proposed a GA based program partitioning algorithm [5], which provides quasi optimal partitioning of large task graphs. However, they reported that the GA based algorithm could not provide better partitioning, in the case of complicated task graphs.

In the GA, genes correspond to the edges in a task graph. Frequently, the result provided by GA is sensitive to the coding of genes. Hence, to provide more superior partitioning, we propose an edge sorting rule in this paper.

In subsection 3.1, we describe some setting for a GA. In subsection 3.2, we propose a sorting rule and several ordering methods.

3.1 Setting for a GA

Chromosome: Each gene corresponds to an edge, and the length of the individual is $\varepsilon(E)$, that means the number of edges. Each *gene* assigned $\{0, 1\}$ corresponds to $\{$*Intra-PE*, *Inter-PE* $\}$ respectively.

Crossover: We adopt one point crossover. The crossover rate is 0.75.

Mutation: The mutation rate is 0.01. For the mutation, the individuals are selected 5% and its genes are flipped 20% randomly.

Fitness value: The fitness value is the length of the critical path. Hence, the individual with the small fitness value is better. In the case a task graph with deadlocks, the fitness value as $\sum_{n \in G} t(n) + \sum_{e \in G} c(e) + 1$.

Process: As the initial setting, $S = \varepsilon(E)^2$ individuals are generated, and genes are substituted 0 or 1 randomly.

In the calculation part, the elitism strategy is adopted [3]. After $2 * S$ individuals are generated by the crossover and the mutation, the superior S individuals are selected as the offspring.

When fitness values of all individuals in a generation become identical, the algorithm is terminated except that the whole individuals have deadlock.

3.2 Accelerating GA

To avoid a deadlock, all edges in a connectional sub-graph G" generated by *Intra-PE*s should be assigned in the neighborhood. Hence, we propose a sorting rule *Sorting 0O* as following.

Sorting 0O

Edges $e = (n_i, n_j)$ are sorted in the ascending order by n_i, and by n_j when the node n_i has multiple incoming edges.

Sorting 0O depends heavily on the order of the nodes. Hence, we propose following sixteen methods, to preserve all nodes in G" to near n.

Ordering 0A

1. Let max be the largest node number among all nodes.
2. (a) When $max \leq 2$, the algorithm is terminated.
   (b) When $max > 2$, $large = max - 1$.
3. Make a list *In-E* including $e_i = (n_i, n_{max})$: $1 \leq i \leq max - 1$
4. (a) When *In-E* is empty, reorder all nodes without n_i ($large < i$) starting from 1. Subtract 1 from max. Go to step 2.
   (b) When *In-E* is not empty, select and remove the e_i, where n_i is the smallest node number, from *In-E*.
5. Change n_i to n_{large}. Subtract 1 from $large$. Go to step 4.

Ordering 0B

Step 4(b) of *Ordering 0A* is changed as follows.

- When *In-E* is not empty, select and remove the e_i, where n_i is the largest node number, from *In-E*.

Ordering 0C and **0D**

The following is added to step 3 of *Ordering 0A* and *0B* respectivery.

(a) When only one incoming edge exists, subtract 1 from max. Go to step 2.

Ordering 0E

Step 3 and step 4 of *Ordering 0A* are replaced as follows.

- Step 3
  Make a list *In-Out-E* of edges related to the node n_{max}.
- Step 4
  (a) When *In-Out-E* is empty, reorder all nodes without n_i ($large < i$) starting from 1. Subtract 1 from max. Go to step 2.
  (b) When *In-Out-E* is not empty, select and remove e_i, where n_i is the smallest node number, from *In-Out-E*.

Table 1: Average generation and error ratio

Method	average generation	standard deviation (generation)	average error	standard deviation (error)
NO	40.02	14.93	6.1	14.39
0O	31.11	9.57	5.76	13.25
0A	30.68	8.36	5.71	14.85
0B	31.51	8.31	4.78	12.63
0C	31.76	8.44	5.78	13.40
0D	32.35	9.73	5.32	12.74
0E	31.08	10.91	4.08	10.44
0F	34.21	12.20	4.15	10.33
0G	31.23	8.69	3.57	10.00
0H	32.17	10.99	6.30	16.15
1A	31.63	10.16	5.72	13.39
1B	30.93	9.36	5.05	12.05
1C	31.40	9.86	5.39	13.99
1D	31.93	10.31	6.36	15.42
1E	29.79	8.50	6.33	15.55
1F	33.13	11.37	5.49	13.86
1G	30.42	8.90	5.15	12.46
1H	32.98	11.31	4.79	11.22

Ordering 0F
Step 4(b) of *Ordering 0E* is modified as follows.

- When *In-Out-E* is not empty, select and remove e_i, where n_i is the largest node number, from *In-Out-E*.

Ordering 0G and **0H**
The following is added to step 3 of *Ordering 0E* and *0F* respectively.

(a) When only one edge exists, subtract 1 from max. Go to step 2.

Ordering 1A

1. Let max be the largest node number among all nodes. Let min be 1.
2. (a) When $max = min$, the algorithm is terminated.
   (b) When $max \neq min$, $small = min + 1$.
3. Make a list *Out-E* including $e_i = (n_{min}, n_i)$: $min \leq i \leq max$.
4. (a) When *Out-E* is empty, reorder all nodes without n_i ($i < small$) starting from $small$. Add 1 to min. Go to step 2.
   (b) When *Out-E* is not empty, select and remove e_i, where n_i is the smallest node number, from *Out-E*.
5. Change n_i to n_{small}. Add 1 to $small$. Go to step 4.

Ordering 1B
Step 4(b) of *Ordering 1A* is modified as follows.

- When *Out-E* is not empty, select and remove e_i, where n_i is the largest node number, from *Out-E*.

Ordering 1C and **1D**
The following is added to step 3 of *Ordering 1A* and *1B* respectively.

(a) When only one outgoing edge exists, add 1 to min. Go to step 2.

Ordering 1E
Step 3 and step 4 of *Ordering 1A* are modified as follows.

- Step 3
  Make a list *In-Out-E* of edges related to the node n_{min}.
- Step 4
  (a) When In-Out-E is empty, reorder all nodes unrelated to the edges related to the node n_{min} starting from $small$. Add 1 to min. Go to step 2.
  (b) When *In-Out-E* is not empty, select and remove e_i, where n_i is the smallest node number, from *In-Out-E*.

Ordering 1F
Step 4(b) of *Ordering 1E* is modified as follows.

- When *In-E* is not empty, select and remove e_i, where n_i is the largest node number, from *In-Out-E*.

Ordering 1G and **1H**
The following is added to step 3 of *Ordering 1E* and *1F* respectively.

(a) When only one edge exists, add 1 to min. Go to step 2.

4 Experiments

We performed experiments Girkar's algorithm [2], the GA without sorting rule (*Sorting NO*), and the GA with our proposing methods.

We used a workstation for the experiments: DEC Alpha 21264 500MHz (Digital Unix 4.0F) with memory capacity of 1GB.

Acyclic weighted directional task graphs are created randomly. $t(n)$ and $c(e)$ are set to random numbers in $[0, 1000]$.

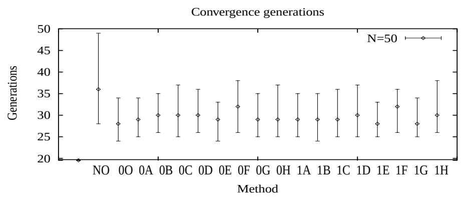

Figure 3: The quartiles deviation of convergence generation with 50 nodes

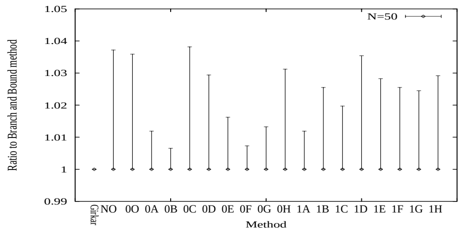

Figure 4: The quartiles deviation of errors in 50 nodes

For the first experiment, 100 random task graphs with 50 nodes and 49 edges are generated. As the result, the average calculation times in Girkar's algorithm and the GA are $1,200[sec]$ and $20[sec]$ respectively. In Tab.1, the average errors to the optimal partitioning are about 5% in the GA by any. Hence, it turned out that the GA is effective.

By Tab.1 and Fig.3, the convergence generations in *Sorting NO* is larger than our proposing ordering methods. In Fig.4, the errors of some ordering methods decrease. Hence, the proposal methods are effective.

To examine the efficiency of each method, 100 random task graphs with 100 nodes and 99 edges are generated. Since Girkar's algorithm can not partition the task graphs, we regard the minimum critical path as the one from all GA solutions for each trial, and calculate error as the increasing rate from the minimum critical path. Also, we evaluate the efficiency of each method by the decreasing rate from the critical path of the original graph.

Fig.5 shows the convergence generations. *Sorting 0O* and each ordering method are not so different clearly except *Sorting NO*.

Fig.6 shows the efficiency of GA solutions. Whole methods indicates considerable improvement except *Sorting NO*.

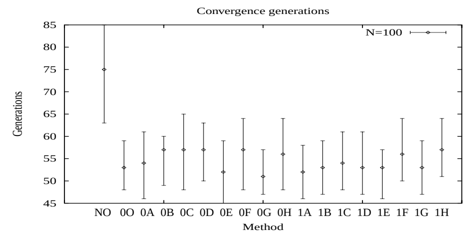

Figure 5: The quartiles deviation of convergence generation with 100 nodes

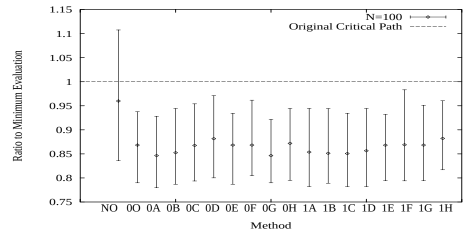

Figure 6: The quartiles deviation of improvement with 100 nodes

Fig.7 shows the error rate to the minimum, and we discuss comparisons of each method in detail. In the case of *Ordering xA* and *xB*, all related nodes are always changed. Hence, the distances of related node should be smaller than *Ordering xC* and *xD*. On the other hand, the error of *Ordering xG* and *xH* are nearly equal to *Ordering xE* and *xF*. In this case, we guess that a single edge is rarely found. Compared with *Ordering xA* and *xB*,*Ordering xE* and *xF* may cause that chromosomes are destroyed by some crossover because of the large distance in related node. Since all edge $e = (n_i, n_j)$ are generated like $n_i < n_j$, the number of the paths including n_j is larger than n_i. Hence, *Ordering xB* may cause that the related node numbers are assigned widely than *Ordering xA*. *Ordering 0A* is not so different from *Ordering 1A*. Consequently, we validated *Ordering xA* as the best.

5 Conclusions

In this paper, we adopted the genetic algorithm based program partitioning algorithm, and proposed the sorting method of edges. In addition, we also inspected the order of edges to preserve all nodes in a sub-graph to be near

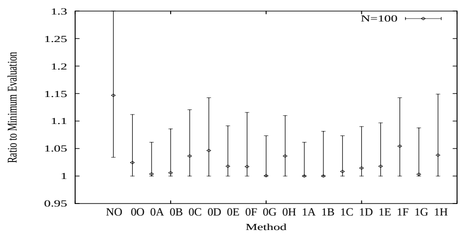

Figure 7: The quartiles deviation of errors in 100 nodes

nodes. We investigated the possible ways to sort edges and to change node numbers, and indicated their effectiveness by experiments.

For the future works, we should investigate crossover, mutation, and fitness value, to use the genetic algorithm more effectively.

References

[1] M. R. Garey, D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, San Francisco, California, 1979.

[2] M. Girkar, C. D. Polychronopoulos. Partitioning Programs for Parallel Execution. *Proc. of the 1988 Int. Conf. on Supercomputing*, pp.216–229, 1988.

[3] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[4] H. Saito, N. Stavrakos, C. D. Polychronopoulos, A. Nicolau. The Design of the PROMIS Compiler. *Int'l J. of Parallel Programming*, Vol.28, No.2, pp.195–212, 2000.

[5] T. Saito, T. Nakanishi, Y. Kunieda, A. Fukuda. Genetic Algorithm Based Program Partitioning. *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, Vol.II, pp.707–712, June, 2000.

[6] M. Takata, Y. Kunieda, K. Joe. Accelerated Program Partitioning Algorithm - An Improvement of Girkar's Algorithm-. *Proc. of the Int. Conf. on Parallel and Distributed Processing Techniques and Applications*, Vol.II, pp.699–705, June, 2000.

[7] M. Takata, Y. Kunieda, K. Joe. A Heuristic Approach to Improve a Branch and Bound based Program Partitioning Algorithm. *The proceedings of the 1999 International Workshop on Innovative Architecture*, pp.105–114, November, 2000.

[8] T. Yamaguchi, H. Ishiuchi, A. Iwasaka, M. Haneda, H. Shouno, K. Joe. The Overview of the Paralleilzing Compiler PROMIS-NWU *Proc. of IPSJ SIGARC 2001-144-14*, pp.79–84, July, 2001. (in Japanese)