Genetic Algorithm Approach towards Optimal Parallelization and Scheduling of Loop Nests[*]

Tong WeiQin, Yao WenSheng, and Ye Hua
Email: wqtong@citiz.net
School of Computer Engineering and Science, Shanghai University
YanChang Road 149, Shanghai 200072
P. R. China

Abstract

This paper describes a genetic algorithm approach to the parallelization and scheduling of loop nests. For parallelization, a chromosome-like coding scheme for weighted directed acyclic graph (DAG) obtained through data dependence analysis is proposed. An evolutionary strategy integrating merging rules and evolution operations is then derived to optimize parallelization. The result obtained is a deterministic DAG ignoring possible branches and dynamic spawning, based on which scheme for the static allocation and reallocation in case of possible system exception are also investigated. A fair allocation scheme with minimal execution time is then obtained.

[*] This work is supported in part by the NSF of China under grant No. 69503005, the National HPC Foundation of China under grant No. 970030, Shanghai Applied Materials R&D Foundation under grant No. 98003 and Shanghai Municipal Committee of Science and Technology under grant No. 99QD14002.

1. Introduction

Numerous efforts have been dedicated to loop transformations, which is the key to automatic parallelization of program. Several approaches have been proposed, the most remarkable among them are unimodular [Banerjee 91] and non-unimodular [Fernandez 95] transformations, extended hyperplane method [Darte 94], optimal graph partition methods [Calinescu 97a][Calinescu 97b] and graph transformation method [Liu 93].

The process of loop transformation can be divided into three stages. The first stage is to gather useful knowledge about the underlying dependences of loop nests. The second stage is to choose the optimal loop transformations in such a way that the dependences of loop nests are obeyed and certain predefined goals are arrived. The third stage is to rewrite loop nests.

However, the predefined goals of loop transformation methods [Banerjee 91] [Fernandez 95][Calinescu 97a][Calinescu 97b] are limited to maximal parallelization blocks and minimal total communication time. And minimal total communication time does not mean minimal total execution time. Moreover, since optimal partition of graph is a NP-complete problem, and existing algorithms, for instance, homomorphic transformation [Liu 93], can not reach optimal results. Hyperplane method aims at minimal execution time but requires that the distance vectors must be constant. In this paper, we propose a genetic algorithm (GA) approach to parallelization of loop nests.

Since GA optimization is time-consuming and the cost of dynamic scheduling itself is high, we attempt to optimize allocation of tasks during parallelization phase and schedule tasks according to allocation scheme.

This paper is organized as follows: a brief description of constructing data dependence graph, namely DAG, is given in section 2. Section 3 presents the simple transformation rules. A chromosome-like coding scheme for DAG and an evolutionary strategy integrating merging rules and evolution operations are discussed in detail in section 4. Section 5 describes static allocating at parallelization phase and reallocating at runtime. Experimental results, which are not limited to loop nests, are given in section 4 and 5. Section 6 presents our concluding remarks.

2. Data Dependence Analysis

The interdependence of statements can be identified by read and/or write reference to variables. For statements S_1 and S_2 referring to the same variable x, if at least one of them modifies x, a *data dependence* exists between the two statements, denoted $S_1\delta^* S_3$.
For loop nests, data dependence is analyzed in terms of statement instances. A *statement instance* is an execution of statements for a fixed value of the surrounding indexes.

In order to identify potential parallelism available in the loop body, *distance vector* is used to represent the data dependence between statement instances. For two statement instances (not necessarily distinct) S_1 and S_2 that refer to the same k-dimension array variable A at the iteration points $(i_1, i_2, ..., i_n) = (x_1, x_2, ..., x_n)$ and $(i_1, i_2, ..., i_n) = (y_1, y_2, ..., y_n)$, respectively, where $i_1, i_2, ..., i_n$ are loop indexes, the distance vector is $(y_1 - x_1, y_2 - x_2, ..., y_n - x_n)$.

The dependence information can be organized as a DAG comprising nodes and edges representing statement instances and data dependence between them respectively.

Consider the loop nest in Figure 1, where the data dependence $S(1)\delta\ S(3)$ exists, a dependence equation can be derived:

$$I_1 = I_2 - 2, \text{ where } 1 <= I_1, I_2 <= 8$$

The solutions are (I_1, I_2)=(1,3), (2,4), (3,5), (4,6), (5,7) and (6,8). The pair (1, 3) stands that S(I_1=1) modifies the variable A[1] and S(I_2=3) refers to the same variable A[1], so S(I_1=1) and S(I_2=3) should be executed in the order of $1 \rightarrow 3$. So does the pair (3, 5). Moreover, the pairs (1, 3) and (3, 5) stand that S(I_2=3) and S(I_1=3) are the same statement instances, so S(1)S(3)S(5) should be executed on the same processor and the execution order of $1 \rightarrow 3 \rightarrow 5$ must be preserved. We can see that there exist two dependence orders:

$$1 \rightarrow 3 \rightarrow 5 \rightarrow 7 \text{ and } 2 \rightarrow 4 \rightarrow 6 \rightarrow 8$$

Therefore a DAG representing the data dependences is obtained (as shown in Figure 1(b)). Algorithms constructing such dependence graph can be found in [Liu 97]. The original loop nests can be rewritten as Figure 1(c).

We will use weighted DAG through the rest of this paper, where the weight associated with node stands for computation time of statement instances, and the weight associated with edge stands for communication time between them.

3. Transformation of DAG

DAG constructed as above can form a parallelization if it is not connected. However, it is usually connected. Through transformation of DAG, that is, merging nodes and edges, a more effective DAG can be obtained.

For two DAGs G=(V, E), G'=(V',E'), where V, V' are sets of nodes, and E, E' are sets of directed edges, and a transformation T transforming G into G', the following definitions and lemma are taken from [Liu 93]:

Definition 1

T is a *homomorphic transformation* if T satisfies:

1) Homomorphism if T maps nodes A_{j1}，...，A_{jn} in G to A'_l in G', and if $< A'_k, A'_l > \in E'$, there exists

$A_{ip} \in A'_k$, $A_{jq} \in A'_1$, such that $<A_{ip}, A_{jq}> \in E$. If $<A_{ip}, A_{jq}> \in E$, then there exists A'_k、$A'_1 \in V'$, such that $< A'_k, A'_l > \in E'$ or $A'_k = A'_1$ for the case $k=1$.

2) Connectivity (A'_l is a connected subgraph consists of A_{j1}，...，A_{jn}, preserving the topology of G.)

Definition 2

T is a *regular transformation* if G' is a DAG and data dependence is obeyed.

Lemma

1) A regular transformation can be implemented by a sequence of merging operations. Every merging operation merges the start node and the end node of a directed edge or a single-level tree, called *Simple Transformation* (ST);

2) The effect of ST can be determined by the in-degree of the end node and the out-degree of the start node, computation time (weight of node) and communication time (weight of edge).

3) ST can be classified into the following four types: namely, SISO (Single In-degree and Single Out-degree), SIMO (Single In-degree and Multiple Out-degree), MISO and MIMO.

According to the Lemma, we have the following ST rules (illustrated in Figure 2):

1) Type-A

Node A can be merged into node B if out-degree of node A and in-degree of node B are one;

2) Type-B

Node A_i (i=1,...,k) can be merged into node B if out-degree of node A_i (i=1,...,k) is one, in-degree of node B is larger than one and node A_i (i=1,...,k) is parent node of node B;

3) Type-CD

Node A can be copied and merged into children of node A, denoted B_1, ..., B_k, if out-degree of node A is k and in-degree of node B_i (i=1, ..., k) is equal to or greater than one.

Now, we can apply these rules to any DAG. However, transformation results of a DAG depend on the sequence of merging. As shown in Figure 3, different sequences of transformations may lead to different DAGs whose execution times are different,too. It is a NP-complete problem to search the minimal-execution-time transformation of DAG, which can only be reached after exhausting all possible merging sequences. In the following section, GA approach is adopted to solve such problem.

4. GA Approach to Optimal Transformation

Given a DAG G=(V, E), for the sake of convenient treatment, unique nodes such as V_{in} and V_{out} with zero-weight can be added, which are called source node and destination node, to G respectively. And edges with zero weight can be added from V_{in} to all zero in-degree nodes and from all zero out-degree nodes to V_{out}.

Definition 3

For $v \in V$, the length of the longest directed path from V_{in} to v is called the *level* of v, denoted Level (v), here by longest we mean the maximal number of edges on the path. We have the following:

Level (V_{in}) = 0

Level (V_j) = $\max_i\{Level(V_i)\}+1$, here V_i are parent nodes of V_j.

Definition 4

For all the directed paths from V_i to V_j, the path with the largest sum of weights is called the *critical path* from V_i to V_j. Here the sum includes both the weights of nodes and edges on a path.

Definition 5

The *critical path* from V_{in} to V_{out} is called the *critical path* of the DAG.

Definition 6

The sum of all weights of a critical path from V_{in} to V is called the *earliest completion time* of node V, denoted T_e(V). Also we have the following:

$T_e(V_{in})$ = cost(V_{in})

$T_e(V_j) = \max_i\{T_e(V_i) + R \cdot \cos t(E_{ij})\} + \cos t(V_j)$, where cost($V_j$) stands for weight of V_j and cost(E_{ij}) stands for weight of E_{ij}, V_i is parent nodes of V_j, and R is the ratio of computation time and communication time.

Definition 7

The sum of weights of the critical path of DAG G is called the *earliest completion time* of G, denoted T_e(G). T_e(G) is the lower bound of the execution cost of G.

Definition 8

For DAG G=(V,E),

$$T_l(G) \quad = \quad \sum_{v\in V, e\in E} (\cos t(v) + R * \cos t(e))$$

is called the *last completion time* of G, denoted T_1(G).

T_1(G) is the maximal execution time of a program executed in sequential order in the case only one physical processor is available. For a DAG G and its transformation results G_1,G_2,..., G_n obtained by applying ST rules, we can choose the one with the least T_e(G). We may identify the one with the least T_1(G) if there are more than one DAGs with the same T_e(G). Here T_e(G) and T_1(G) are object function.

We now introduce the chromosome-like coding scheme. First, we arrange all edges <V_i, V_j> in G by the level (V_i), level(V_j) in ascending order. One bit is assigned to each directed edge, a binary vector (chromosome) with length of |E| is obtained. The merging of two nodes, V_i and V_j, is accomplished by setting the bit corresponds to <V_i, V_j> to zero.

Definition 9

The set of edges that will be changed when applying one ST rule to G is called *basic merging edge set* (BME set). The elements of the BME set are arranged according to the significance (from the most significant to the least) of bits in a binary vector.

The BME sets of ST rules are listed in the following:

1) Type–A

The bit corresponds the edge <a,b> is zero, {<a,b>} is a BME set.

$c_b = c_a + c_b;\ \ c_a = 0;\ \ c_e = 0$

2) Type–B

The bits correspond to the edges *e*1=<a1,b>, *e*2=<a2,b>, e3=<c,a1> and e4=<d,a2> are zero. The bits corresponds to the edges *e*5=<c,b> and *e*6=<d,b> remains unchanged,.{<a1,b>,<a2,b>,<c,a1>,<d,a2>,<c,b>,<d,b>}is a BME set.

$c_b = c_b + c_{a1} + c_{a2}; c_{e5} = c_{e5} + c_{e3}; c_{e6} = c_{e6} + c_{e4}; c_{e1} = 0; c_{e2} = 0$

3) Type-CD

The bits correspond to the edges *ei*=<a, bi> (i=1,...,k) are zero. {<a,bi>(i=1,...,k)}is a BME set.

$c_{bi} = c_a + c_{bi},\ \ c_{ei} = 0,\ \ i = 1 \cdot\cdot k;\ \ c_a = 0$

An edge may belong to two distinct basic transformation edges sets. BME sets are first applied to initialization process of GA and then to crossover and mutation operations. When

ST rules are applied to DAG G, BME sets of G are treated as a whole. Especially when crossover operation is concerned, transformation requires that crossover operation should not treat edges in BME sets differently. Assume that C_1 and C_2 are two chromosomes participating in crossover operation, we have the following crossover rules:

Rule 1. B is one of the BME sets of C_1 and C_2, then crossover operation is not allowed to divide B into two non-null sets;

Rule 2. If an crossover operation divides C_1 into two parts, and the corresponding node sets are V_1 and V_2, for any $v \in V_1$ and $v \in V_2$, then only one of the following holds:

i) v has only been changed in C_1;
ii) v has only been changed in C_2;
iii) v has never been changed.

For DAG G, we summarize our parallelization algorithm, with $T_e(G)$ and $T_l(G)$ as object function, P_c as crossover probability, P_m as mutation probability, and Pop_Size as population size, as follows:

Step 1 Generate BME sets of original dependence graph. Randomly choose several BME sets (not interleaving) to exercise transformation on G. Then a binary vector (chromosome) is obtained, and added into initial *Population*; generate a *Population* with size Pop_Size;

Step 2 Repeat from step 3 through step 6 until termination conditions are satisfied;

Step 3 Calculate the object function (i.e. $T_e(C_i)$, $T_l(C_i)$) value of each chromosome in *Population*; and then calculate fitness function value of each chromosome with $F(C_i) = \overline{T_e} / T_e(C_i)$ where $\overline{T_e}$ is mean value of $T_e(C_i)$; record the fittest chromosome in history;

Step 4 Choose chromosomes with size Pop_Size from *Population* to form a new population according to their fitness;

Step 5 Apply crossover operation on the new *Population*, subject to the crossover rules;

Step 6 Apply mutation operation to exercise transformation with newly generated BME sets;

Step 7 Calculate object function value and fitness function value of each chromosome in the last *Population*; choose the fittest chromosome in history.

The DAG obtained is deterministic. Moreover, branch structure is discussed in this paper. Simulation result is shown in Figure 4, where a complex DAG is taken as example.

5. Static Allocation Based on DAG

We naturally extend our approach to the static allocation of DAG obtained previously aiming at minimal total time of parallel execution and fair allocation. The two goals can be judged according to weights of nodes and edges, and allocation scheme. At the same time, robustness is also a factor under consideration. When an exception occurs, *scheduler* responses and generates reallocation scheme from the exception point.

The most recent DAG-based scheduling algorithm can be found in [Blelloch 99], in which dynamic DAG is also considered. As with their work, we have the following definitions:

Definition 12

A *schedule* of a DAG is a sequence of S_i, i=1, 2, ..., k, steps, where each step comprises a set of nodes such that each node appears only once and a node can not be scheduled unless all its parents have been scheduled.

Definition 13

A *greedy schedule* is such a schedule that every step comprises all ready nodes (tasks).

One step corresponds to one level of DAG from the definition of greedy schedule. The algorithm of greedy schedule is the same to the one that calculates the level of nodes. Here we use *step*(V_i) to represent the step of node V_i in greedy schedule.

Here, we propose an allocation algorithm. First, the algorithm divides nodes into several sequential sets according to greedy schedule algorithm. And then, GA approach is adopted to optimize allocation scheme.

We begin with encoding nodes of DAG with a structure vector. Every element represents a node (task) and has two items, i.e. PROCESSOR_ID and PRIORITY. The PROCESSOR_ID item represents the processor this task is allocated to. And the PRIORITY item represents the priority this task owns. The priority is relative to other tasks allocated to the same processor in the same step. Obviously, tasks allocated to the same processors should have different PRIORITY, which leads to

Rule 1: For two distinct elements *x*, *y* of Chromosome, *x*<>*y* must be obeyed, i.e., *x*.PROCESSOR_ID= *y*.PROCESSOR_ID and *x*.PRIORITY=*y*.PRIORITY can not be satisfied at the same time;

Object function consists of three parts:

1 Computation time of tasks;

2 Communication time between tasks if any; it can be ignored if the two tasks are allocated to the same processors;

3 Fairness degree.

Here, T_i stands for load on processor *i*, $\overline{T}$ stands for average load, and *n* is the number of available processors.

We use fairness degree factor, denoted B, to represent the importance of fairness degree. The greater B is, the more important fairness degree is. When B is zero, the optimization algorithm considers only minimum execution time.

For computation time of tasks, we have the following:

$T_e(V_{in}) = \text{cost}(V_{in})$

$T_e(V_j) = \max_i\{T_e(V_i) + R\cdot \text{cost}(E_{ij}), T_e(V_k)\} + \text{cost}(V_j)$, where cost(V) stands for weight of V and cost(E_{ij}) stands for weight of E_{ij}, V_i is parent node of V_j, and R is the ratio of computation time and communication time, and cost(E_{ij}) is ignored if nodes $V_{i,}$ V_j are allocated to the same processor, and $T_e(V_k)$ is considered if nodes V_k, V_j are allocated to the same processor and V_k, V_j are in the same step and V_k is completed just before V_j.

$$DEF = \sqrt{\sum_{i=1}^{n}(T_i - \overline{T})^2} \;/\, n$$

Assume that there are certain processors, denoted *P*, with probability P_f fails in some step due to unexpected events. Consider robust static allocation. To do so, we randomly choose one step, denoted S_i, where fault appears, and define:

Rule 2: $\forall x \in S_j\,(j \ge i) : x.PROCESSOR \notin P$

Consider the static allocation scheme under such conditions. We calculate the value of object function as follows:

Step 1 Calculate execution time of every processor according to static allocation scheme and choose the largest value as the minimum execution time, denoted T;

Step 2 Calculate fairness degree, denoted DEF;

Step 3 The results of T+B*DEF are the value of object function.

We now present static allocation algorithm as follows:

Step 1 Randomly generate a structure vector (chromosome), and add it into initial *Population* if it is subject to the two rules; generate a *Population* with size Pop_Size;

Step 2 Repeat from step 3 through step 6 until termination conditions are satisfied;

Step 3 Calculate the object function value of each chromosome; and then calculate fitness function value of each chromosome with $F(C_i) = \overline{T_e} / T_e(C_i)$ where $\overline{T_e}$ is mean value of $T_e(C_i)$; keep record of the fittest chromosome in history;

Step 4 Choose Pop_Size chromosomes from *Population* according to their fitness; the new *Population* consists of these Pop_Size chromosomes;

Step 5 Apply crossover operation on Population;

Step 6 Apply mutation operation on Population;

Step 7 Calculate object function value and fitness function value of each chromosomes in the last Population; choose the fittest chromosome in history.

The static allocation scheme can be generated during parallelization phase. *Scheduler* can allocate tasks *step* by *step* according to this scheme at runtime. The earliest completed task in one step will trigger *Scheduler* to allocate new tasks onto processors and set reasonable priority on tasks. If any exception event occurs, *Scheduler* generates new allocation scheme. To do so, the information about completed tasks and running tasks on each processor need to be recorded. We use completed set and waiting set to keep record of such information. Here completed set refers to the tasks that have been completed on this processor, and waiting set refers to the tasks that are waiting for being scheduled on this processor.

When an exception event occurs, *Scheduler* cancels all waiting tasks and the tasks on the processor on which fault occurs, and recalculates the value of object function as follows:

Step 1 Calculate the execution time of every processor according to allocation scheme and choose the largest value as the minimum execution time, denoted T, not including tasks in completed set;

Step 2 Calculate fairness degree, denoted DEF;

Step 3 The results of T+B*DEF are the value of object function.

Then *Scheduler* recreate static allocation scheme according to the algorithm described above.

Simulation result is shown in Figure 5. Special techniques [Huang 99] are adopted to accelerate the convergence of allocation algorithm.

6. Conclusion

We have presented our approach towards parallelization and scheduling of loop nest. Simulation experiments show that the scheme described in this paper can be applied to optimize deterministic DAG-based problems

References

[Banerjee 91] **U. Banerjee,** "Unimodular Transformations of Double Loops", *Advances in Language and Compilers for Parallel Processing*. MIT Press, 1991, pp.192-219.

[Blelloch 99] **G. E. Blelloch, et al,**. "Provable Efficient Scheduling for Languages with Fine-Grained Parallelism", *Journal of the ACM*, vol.46, No.2, March 1999, pp.281-321.

[Calinescu 97a] **R. Calinescu**, "BSP Scheduling of Regular Patterns of Computation", PRG-TR-1-97, Oxford University Computing Laboratory.

[Calinescu 97b] **R. Calinescu**, "A Scheme for the BSP Scheduling of Generic Loop Nests", PRG-TR-26-97, Oxford University Computing Laboratory.

[Darte 94] **A. Darte, Y. Robert,** "Constructive Methods for Scheduling Uniform Loop Nests", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 8, Aug. 1994, pp.814-822.

[Fernandez 95] **A. Fernandez, J. M. Llaberia, M. Valero-Garcia,** "Loop Transformation Using Nonunimodular Matrices", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 8, Aug. 1995, pp.832-840.

[Holland 75] **J. Holland,** "Adaptation In Natural and Artificial Systems", University of Michigan Press, 1975.

[Huang 99] **Y. Huang, et al,** "A Solution to Deceptive Problems in Genetic Algorithm Based on an Adjustable Mutation Operator", (in Chinese), *Chinese Journal of Software*, Vol. 10, No. 2, Feb. 1999, pp. 216-219.

[Liu 93] **Liu Jian, et al,** "A Merging Method for Process Partition", (in Chinese), *Computer Engineering and Application*, 1993(4), pp.5-11.

[Liu 97] **Liu Jian,** "Parallel Distributed Programming", (in Chinese), published on July 1, 1997, HuaZhong University of *Science and Technology Press*, pp. 105-116.

```
For i = 1 to 8 do
S      A[i] = g(A[i-2]);
            (a)

For i=1 to 8 step 2 do
       A[i]=g(A[i-2]);
For i=2 to 8 step 2 do
       A[i]=g(A[i-2]);
            (c)
```

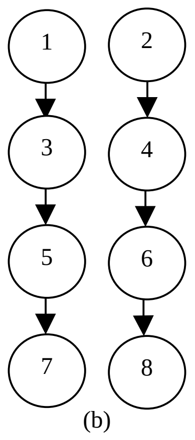


Figure. 1: Dependence graph and transformation of loop nests

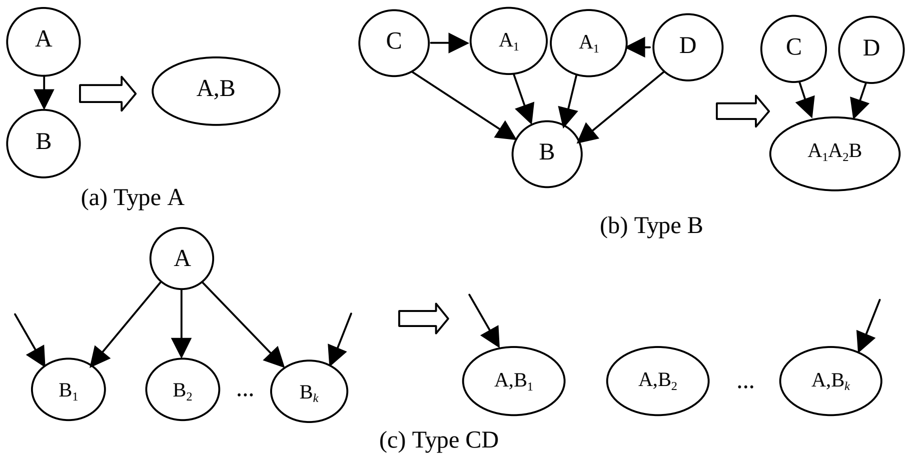


Figure 2: Fundamental Merging Rules

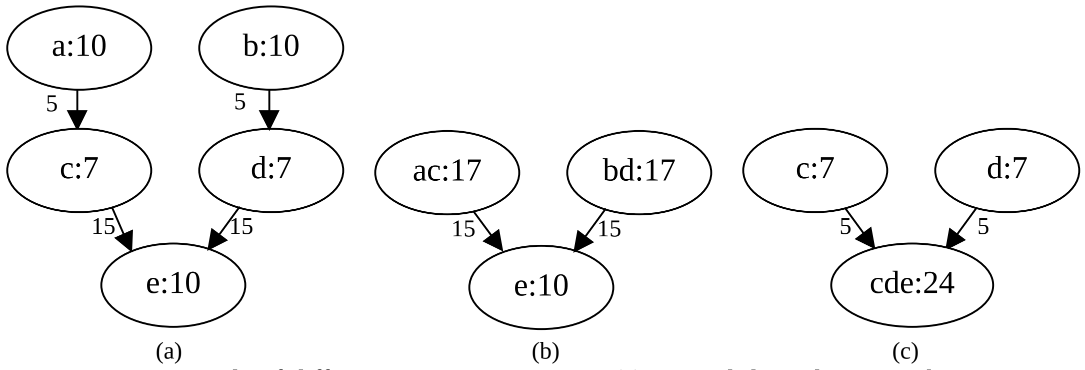
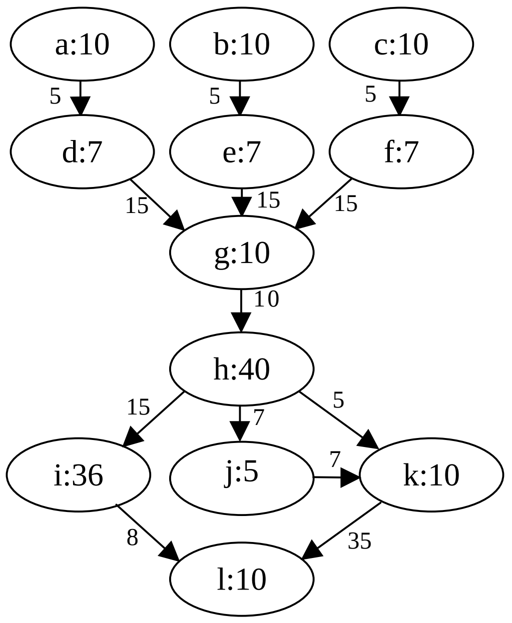
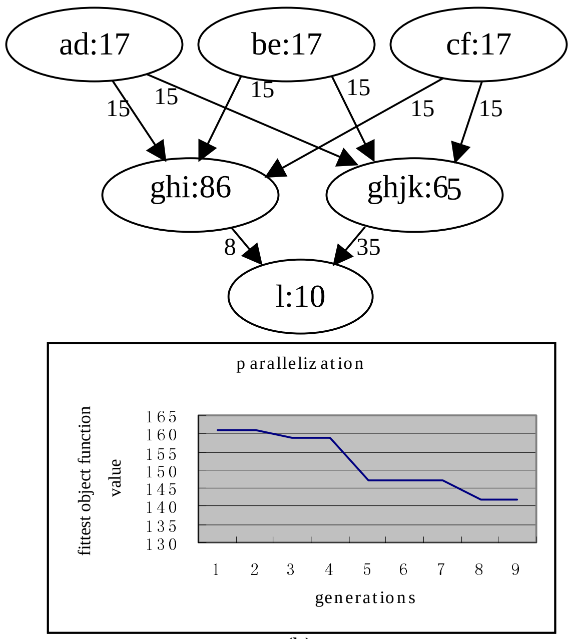


Figure. 3: Example of different merging sequences. (a) original dependence graph; its cost is 47; (b) merge nodes a, b into nodes c, d respectively (by Type-A); its cost is 42; (c) merge nodes c and d into node e (by Type-B); its cost is 36;

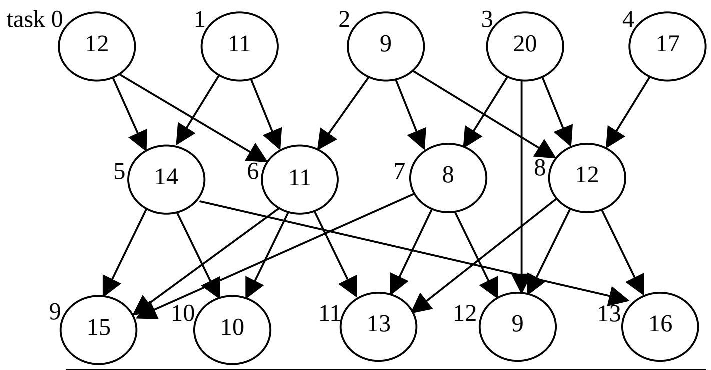


(a)



(b)

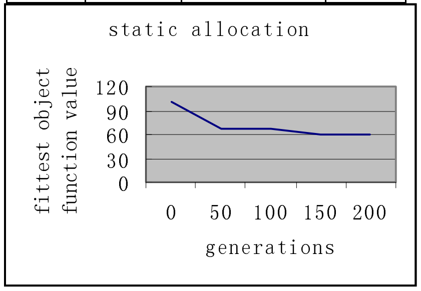
Figure 4: Convergence of algorithm; (a) original dependence graph; (b) result: chromosome=0001110000101, T_e=142; P_c=0.75, P_f=0.02, Pop_Size=100, R=1.



edges	cost	edges	cost	edges	cost	Edges	cost
<0,5>	2	<2,8>	2	<4,8>	3	<7,9>	2
<0,6>	3	<3,6>	2	<5,9>	2	<7,11>	2
<0,7>	2	<3,7>	3	<5,10>	3	<7,12>	4
<1,5>	2	<3,8>	2	<5,13>	2	<8,11>	2
<1,6>	4	<3,12>	2	<6,9>	4	<8,12>	4
<2,6>	3	<4,5>	3	<6,10>	2	<8,13>	3
<2,7>	3	<4,7>	2	<6,11>	3		

(a)

task	priority	processor_ID	T_e
0	7367	2	12
1	21062	1	11
2	28978	2	21
3	22058	0	20
4	25529	3	17
5	3034	3	31
6	1169	2	33.2
7	22967	0	32.3
8	21207	3	43
9	9054	2	49.5
10	1021	0	45.4
11	15302	0	58.4
12	14194	2	58.5
13	16861	3	59



(b)

Figure 5. Convergence of algorithm; (a) DAG-based task flow graph; (b) allocation scheme where R=1.17, B=13, P_f=0.1, P_c=0.7, P_m=0.1, Pop_Size=100, number of processors is 4 and assume processor 1 will be down at step 1 (P_m increases to 0.7 linearly). Result: 60.97.