Vol. 40 No. 12 Transactions of Information Processing Society of Japan Dec. 1999

Regular Paper

GA-based Task Allocation by Throughput Prediction

Yoichi Aoyagi,† Minoru Uehara,†† Hideki Mori†† and Akira Sato††

Recently, multimedia data processing dealing with audio and visual data and various combinations of such data is gaining importance. In such multimedia processing, data streaming is utilized by pipelined parallel processing on successive data flow computations such as data compressions, data expansions, effect processing and so on. With this background, high throughput task scheduling for software oriented pipeline processing is expected. Scheduling issues are classified into allocations of tasks to processors, and deciding the appropriate execution orders of tasks. As there are so many solutions, these issues are called an NP hard problem. In this paper, a static task allocation method for stream-based computations using a Genetic Algorithm (GA) is proposed. In our proposed method, in order to find global solutions efficiently, a list-scheduling algorithm and a GA are adopted, and both are combined together. Finally, we show that our proposed method gives better allocation results compared with a conventional CP/MISF method.

1. Introduction

In task allocation, there are many possibilities for allocating tasks to processors and the execution orders of tasks. Therefore, it is difficult to find the optimal solution. Previously, list scheduling algorithms such as a CP (Critical Path) method and a CP/MISF (Critical Path/Most Immediate Successors First) method[4] have been proposed. If tasks are cyclical, these scheduling methods are not suitable, because critical path analysis of tasks is utilized. With task allocation, it is difficult to find the optimal allocation, because there are enormous combinations of allocations, this is known as the NP-hard problem. The possible task allocation patterns for a computation increase enormously as the number of processors and the number of tasks increase, therefore it is difficult to decide the most suitable allocation pattern. In this paper, we propose a static task allocation method using a genetic algorithm (GA), and name it the GA-Knapsack method[1],[2]. This method treats the list of list-scheduling as a gene, and performs static allocation.

This paper consists of following sections. Section 2 describes related works on task allocations and GA. In Section 3, the stream-based computing model[2], which is suitable for stream processing and easy for task analysis, is proposed. A stream computing model[1],[2] featuring high speed communications demonstrates its ability best in a distributed memory environment. In Section 4, we propose a task allocation method using greedy knapsack approach. In Section 5, we propose a practical task allocation method which combines a Knapsack algorithm and a Genetic Algorithm. In Section 6, we evaluate our GA-Knapsack algorithm by comparing it with a conventional CP/MISF method.

2. Background

Task allocation algorithms are classified into a static method and a dynamic method[8],[9]. The static allocation method features scheduling performed in advance of task operation, such as a compile time. This is useful for a no conditional branch case (basic block). However, the static allocation method is not practical, in the case where a conditional branch is included, because it is not predictable at compilation time. On the other hand, the dynamic method features scheduling performed during the execution period of task operations. Generally speaking, it is not suitable in the case of fine-grained processing such as instruction based operations, because increased overheads exist. Therefore, the static method is suitable for our streaming operations. Previously, list scheduling-based or heuristic-based task scheduling methods have been presented. Conventional list-scheduling is as follows. The order for allocating tasks is registered in a list table. When any processor becomes free, then a new task is extracted from the top of the

† Annex Information, Inc.
†† Department of Information and Computer Sciences, Toyo University

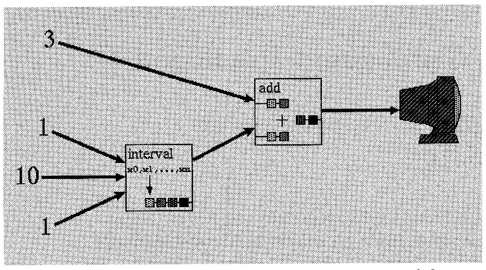


Fig. 1 The stream-based computing model.

list table. The CP/MISF[4] is a typical method based on list scheduling algorithm. In this paper, CP/MISF and our proposed GA-Knapsack method are compared and evaluated. In addition, a task allocation on cyclic tasks such as a streaming operation is discussed.

As a cyclic task analysis model, Mars (a Maintainable Real-time System)[7] has been presented. Mars consists of cyclic tasks where tasks are executed in certain cycles. At the stage of task analysis, the execution timing of real-time tasks is adjusted by the time editing operations implemented by the text editor. Global timing adjustments are made possible by setting the temporal execution period to a code part.

In our paper, in order to search for solutions globally, GA are utilized. The GA takes a hint from the theory of biological evolution, in that it can be used for random sorting, learning and optimization. This is utilized for problems such as placing parts effectively in limited space and limited time. A typical example is the VLSI layout problem. The evaluation function to represent fitness is a key factor in GA.

3. Stream-based Computing Model

We have proposed a stream-based computing model, which is suitable for data streaming[6], pipeline processing and so on. In this section, we describe this model.

3.1 Definition of Basic Model

First, we show conceptual definitions of the basics of the stream-based computing model in **Fig. 1**. This model allows simultaneous tokens on one stream as communication channels. And the sequence of each token's arrival will be preserved and guaranteed. Figure 1 is an example of stream-based computation, where computation "3 + interval(1,10,1)" is performed. The task "interval(1,10,1)" generate data "1, 2, ..., 10" periodically. Task "add"

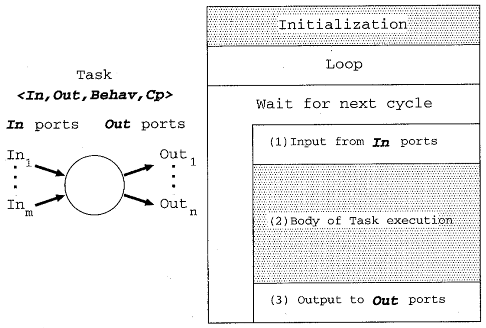


Fig. 2 Structure of a task.

performs "+" operation between constant data 3 and output data from the task "interval". Then the result "4, 5, ..., 13" is displayed on the monitor. A data path between two tasks is called a stream. Tasks and streams are represented in a graph, where a task is shown as a node, and a stream is shown as an arc.

3.2 Task

A task in the stream-based computing model is a concurrent object which communicates with other tasks via streams. The task has two kinds of ports, one is for input, the other is for output. Streams are linked to those ports. In this model, a task is composed of one or more primitive tasks. The metrics model of a task is represented as $\langle In, Out, Behav, C_p \rangle$, and the structure of a task is shown in **Fig. 2**, where $In_1, \ldots, In_m$ are the input ports, $Out_1, \ldots, Out_n$ are the output ports (m, n are number of input ports and output ports), *Behav* is the behavior of the task, and C_p is the processing cost of the task.

The behavior of a task consists of one initialization part and one loop part shown in Fig. 2. The loop part consists of the following sequence:

(1) Waiting period for next cycle.
(2) Input from *In* ports.
(3) Body of task execution.
(4) Output to *Out* ports.

3.3 Stream

A stream is defined as a communication path from an *Out* port to an *In* port. A stream is represented as one input and one output which are connected to tasks' ports. A stream has the following features:

- It is easy to analyze because communicating tasks are fixed.
- The order of tokens in a stream can be guaranteed.

- It is easy to optimize the granularity of communication by tuning buffer size.

Generally, it is difficult to predict the size of tokens in a stream. In our model, however, connections between tasks are statically given, and communications are realized by flowing a constant size of data continuously, so it is easy to estimate communication cost using static analysis.

3.4 Metrics Model

In this section, we describe the metrics model of stream-based computing. In this paper, metrics model means a formal representation used for performance evaluation of the given computation. In our model, a pipeline processing, which repeats a cyclic procedure, is composed of both tasks and streams. Computing cost consists of processing cost depending on processing time of an individual task and communication cost depending on communication delay by tasks through streams. The cyclic period of an allocated task is predictable if both processing cost and communication cost are given.

In the stream-based computing model, computing cost is classified into both processing cost, depending on the execution time of a task, and communication cost, depending on delay through stream. Next, we describe both costs.

3.5 Measurement of Processing Cost

Here, we describe how to decide the processing cost of a task. A task is an execution unit repeating a particular procedure and consists of one initialization part and one loop part as mentioned in Section 3.2. Both parts are shaded in gray in Fig. 2. The processing cost of a task is defined as the period of an iteration. There are two methods used to get processing cost, one is to analyze the program code generated by the compiler, the other is to measure processing time by executing the generated code directly. In the analysis based method, however, the influences of inputs/outputs synchronization are ignored. Therefore, we employ the measurement based method.

3.6 Measurement of Communication Cost

In this section, we present the measurement of communication cost. There is a large difference between local communication and remote communication in the distributed system. Local communication is often realized by means of memory-to-memory copy. On the other hand, remote communication is realized through a variety of methods such as message passing, shared memory and so on. In this paper, we employ message passing to realize stream communication.

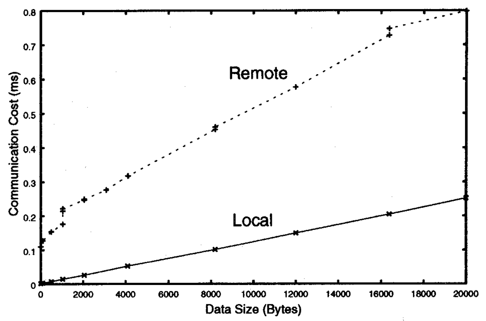


Fig. 3 Communication cost.

In general, the message is divided into fixed size pieces called packets. The sending time T_s is shown as follows:

$$T_s = \alpha\lceil\frac{N}{Buf}\rceil + \beta \cdot N + T_t(N) \qquad (1)$$

where N is the data size in a message, Buf is the size of a packet, $T_t(N)$ means a function giving transfer time and α, β are system dependent constants. The first term represents packet creation time, the second term represents data copy time to packet and the third term gives transfer time in the network. This transfer time is hidden in the case of asynchronous communication where this can be overlapped with other processing time. We used distributed system Fujitsu AP3000 (UltraSparc 140 MHz $\times$ 8 nodes), as a testbed for measuring communication cost. In order to realize message passing between tasks, we used MPI (Message Passing Interface) libraries[3]. The communication cost is shown in **Fig. 3**.

In this figure, X axis is data size per transfer, Y axis is the time of a transfer where the line labeled "Local" shows local communication cost and the line labeled "Remote" shows remote communication cost.

In Fig. 3, parameters in Eq. (1) are given from the segment and slope of the graph such as $Buf = 8192$, $\alpha = 0.14$, $\beta = 6.8 \times 10^{-6}$, $T_t(N) = 0.215$ in data size over 1024 (Bytes). There are influences against communication cost, such as message conflicts and system optimization. So approximate value is used for estimating communication costs. Generally speaking, it is difficult to estimate the data size of messages. In our model, however, connections between tasks are statically given,

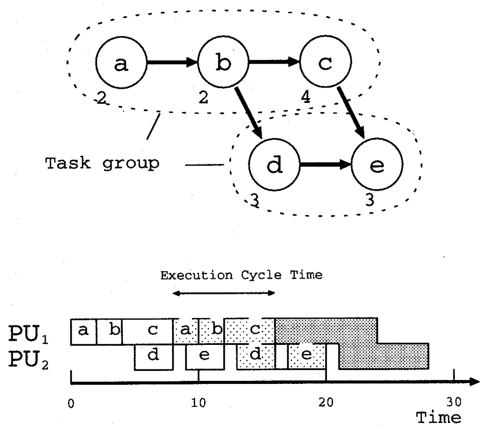


Fig. 4 Example of allocation and execution cycle time of the task groups.

and communications are realized by flowing a constant size of data periodically, so it is easy to estimate communication cost with static analysis.

3.7 Throughput

Here, we describe the throughput which is an important factor in the stream-based computing model. In this paper, we define the throughput *th* as follows:

$$th = 1/T \tag{2}$$

where T is the execution cycle time of the computation. The computation is a set of all tasks. The allocated tasks are executed in a block repeatedly, and this block length is the execution cycle time. **Figure 4** is an example of task allocation and its timing chart. In this figure, the computation is represented by graph of tasks from 'a' to 'e', where an arc is a data stream between tasks, and $PU1$, $PU2$ are processors. Task groups are shown by a dashed line in Fig. 4, and tasks belonging to a group are allocated to the same processor. The throughput of computation is determined when a task allocation pattern in the target system is given. An allocation pattern is defined as the combination of tasks' allocation to processors and the executing order of the tasks. In this case, task groups are already allocated to the processors in the way mentioned in Section 4.2.1 and its task allocation pattern *al* is as follows:

$$al : \left\{ \begin{array}{l} PU_1 : a, b, c \\ PU_2 : d, e \end{array} \right\} \tag{3}$$

where the task groups consists of three tasks "a,b,c" by PU_1, and the other consists of "d,e" by PU_2, and pipeline processing is composed as a whole.

4. Task Allocation Method Using the Knapsack Approach

Here, we propose the task allocation method using the Knapsack approach. Our task allocation algorithm consists of one analysis part and one allocation part. The analysis part is the investigation of system and application dependencies. In the allocation part, the practical allocation is decided by choosing from various allocation patterns by comparing the predicted throughput of the allocation patterns. The allocation part is described in Section 4.2.

4.1 Analysis for Deciding Task Allocation

We use throughput prediction for deciding the task allocation. We need the following data as prerequisite parameters to the prediction:

- communication cost of each stream
- processing cost of each task
- connectivity relation among tasks and streams
- number of processors

Since these data depend on system and application, they are estimated at the following two stages.

4.1.1 Preliminary Evaluation

At this stage, system dependent factors such as processing costs of tasks and remote and local communication latency are measured by an evaluation program. Then α, β in Eq. (1) are derived using the latencies.

4.1.2 Analysis at Compilation Time

In this stage, data size per communication and the relation among tasks are analyzed, where the relation means the order of execution sequence among tasks. The communication cost is calculated using the parameters provided at the preliminary stage and the data size of the stream. With these data, the prediction of throughput for the allocation becomes possible.

4.2 Global Search Algorithm for Task Allocation

In this section, we propose a global search algorithm for task allocation. This method uses allocating order lists which are lists of list-schedulings, and search the practical allocation among the allocation patterns generated from the lists. **Figure 5** is an example flow of task allocation to 2 processors, and **Fig. 6** is its algorithm. The flow of Fig. 5 is as follows:

(1) Represent computation as a graph of tasks.

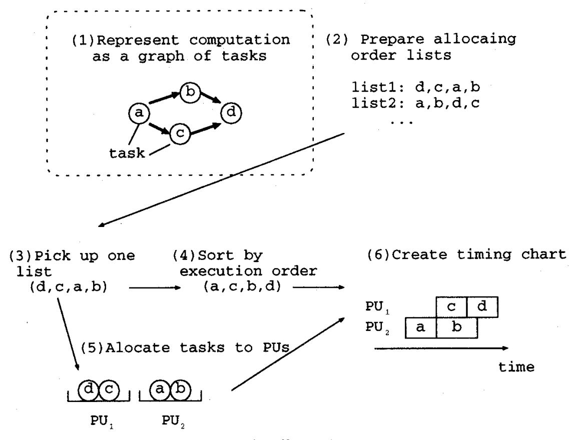


Fig. 5 Flow of task allocation to 2 processors.

```
Lists :array of allocating order lists
       where $Lists[i]=\{a_{i,1}\dots a_{i,n}|a_{i,j}$ is a task
       ($j \in 1\dots n$, $n$ is number of tasks)}
m     :number of processors
al    :allocation pattern
$al_{best}$ :the best allocation pattern observed
       before
th    :throughput of al
$th_{best}$:throughput of $al_{best}$

$th_{best}$=0;
for i=1 to #Lists {//#Lists is No. of Lists
  al = allocate(Lists[i], m);
  th = get_throughput(al);
  if th > $th_{best}$ { // update
    $th_{best}$ = th;
    $al_{best}$ = al;
  }
}
```

Fig. 6 Algorithm of Global search Knapsack.

(2) Prepare allocating order lists.
An allocating order list is a kind of list based on the list-scheduling algorithm. The allocating order lists in our method are not related to the task execution order, in order to create various task allocation patterns from the lists. On our global search algorithm, the allocating order lists represent all the permutation lists.

(3) Pick up one list ("d,c,a,b") from an array of allocating order lists.
A task allocation pattern is created from the list. A task scheduling pattern consists of an order of task execution, and an allocation of tasks to processors.

(4) Sort the list by the dependency order of tasks, then the first sorted list ("a,c,b,d") is used as the execution order list.

(5) Allocate tasks to processors (PUs) by the Knapsack method described in Section 4.2.1.
In Fig. 6, function "*al*=allocate(*list*, *m*)" allocates tasks to *m* processors in the order of the *list* and register the allocation pattern to *al*. An example of *al* is shown in Eq. (3) in Section 3.7.
PU1: c, d
PU2: a, b
The detail of "allocate(*list*, *m*)" is shown in Section 4.2.1.

(6) Create timing chart.
This chart instructs the execution sequence of tasks on each processor. The allocation pattern is evaluated using a timing chart. In Fig. 6, function "*th*=get_throughput(*al*)" returns the throughput of the allocation pattern *al* mentioned in Section 3.7.

4.2.1 Knapsack Task Allocation Algorithm

Figure 7 represents a process of task allocation by our Knapsack task allocation method which is a variant of a knapsack problem[5], and **Fig. 8** is the algorithm. In Fig. 7, PU_1-PU_m are processors. In Knapsack method, allocation patterns are decided from the allocating order list. An allocation pattern consists of an allocation of tasks to processors and executing order of tasks. There are many combinations of these allocation patterns, but most of them are not suitable as a solution. So, the Knapsack method takes a greedy approach, and created allocation patterns by the Knapsack method

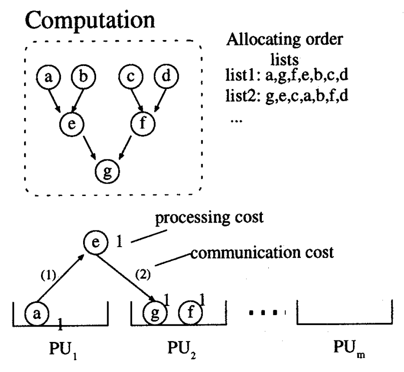


Fig. 7 Process of knapsack task allocation.

are restricted as follows:

- Tasks communicating with larger communication latency are gathered in the same processor to reduce communication costs.
- The maximum processor load is limited to C to avoid an imbalance of processor loads. C is calculated as

$$C = e \cdot T_{ideal} \qquad (4)$$

where e is the coefficient (≥ 1) which depends on applications, and T_{ideal} is the ideal execution period assuming that the computation is equally divided into processors and there are no communication delays. It is calculated as

$$T_{ideal} = \sum_{i=1}^{n}(p_i)/m \qquad (5)$$

p_i: processing cost of task a_i
n: number of tasks
m: number of processors

With this restriction, the allocated load to the processors gets closer to equal.

Figure 7 is an example of task allocation by the Knapsack method, and Fig. 8 is its algorithm, and the flow of the algorithm is as follows:

(1) Repeat following steps until all the tasks in *list* are allocated, where *list* is the allocating order list of tasks. In Fig. 7, following allocating order list is used.
list1: a,g,f,e,b,c,d
('a'-'f' are task names)
In Fig. 7, tasks 'a', 'g', 'f' are already placed and 'e' is in the process of being allocated. To make the explanation simpler, all the tasks' processing costs are assumed to be 1.

(2) Select the processor task is allocated to.

```
Function allocate(list, m)
list:a task list which represents allocating order
m  :number of processors
{
  C    :capacity of processor load
  tsk  :a task
  th   :throughput
  p_alloc:the processor id that a task is
          allocated to
  p    :processor
  al   :allocation pattern
  for i=1 to #list {//#list is No. of tasks
    tsk = list[i];
    // select the allocating processor p_alloc
    p_alloc = 1;
    for p = 2 to m {
      if gravity(tsk,p)
              > gravity(tsk, p_alloc){
        p_alloc=p if ( load_of( p ) < C );
      }
    }
    //put tsk into p_alloc
    place(al, tsk, p_alloc);
  }
  return al
}
```

Fig. 8 Knapsack task allocation algorithm.

Function "gravity(tsk,p)" returns the sum of remote communication costs between allocating task tsk and other tasks in the processor p.
Function "load_of(p)" returns the current load of the processor p. In Fig. 7, the sum of load in PU_1 is 1, and in PU_2 is 2, at this time. The allocating task 'e' is pulled by both 'a' and 'g', and the gravity (communication latency) to 'g' is stronger, so 'e' is placed into PU_2 which has 'g'. Then the sum of the load of PU_2 becomes 3. If the load of PU_2 is over the limit, 'e' is placed into PU_1 which has extra space to take 'e'.

(3) Allocate task to the selected processor, and change the allocation status.
Function "place(al, a, p_{alloc})" places a task a into the processor p_{alloc}, and the current allocation status al is changed. Tasks communicating with high communication latency are placed into the same processor with this grouping policy.

5. GA-based Task Allocation Method

Task allocation problems aim to find the optimal allocation solution seeking the optimal combinations of tasks. So, we used GA in order to find global solutions efficiently.

```
Chroms: set of task allocating order lists
        randomly generated
t      : generation of GA
gen_max : upper limit of number of generations
Fits   : set of fitness where Fits[i] is fitness
        of Chroms[i]
Fit_best : the best fitness (∈ Fits) on
        generation t

Chroms = generate_rand_orders();
fit_best=0;
for t = 1 to gen_max {
  Fit_best[t]=evaluate(m, Chroms, Fits);
  crossover(Chroms, Fits);
  mutation(Chroms);
  exit if is_converged(Fit_best) // End
}
```

Fig. 9 GA-based task allocation algorithm.

5.1 Flow of the GA-Knapsack Algorithm

The GA-based task allocation algorithm in **Fig. 9** is as follows:

(1) Prepare random allocation lists. Function "*Chroms* = `generate_rand_orders()`" generates lists of task allocating orders *Chroms* randomly, and they are used as chromosomes, where a task is a locus of a chromosome. The detail of the allocation process is shown in Section 4.2.1.

(2) Repeat the following operations from 1st to gen_{max}th generation.

(a) Create allocation pattern from chromosomes, then evaluate it. Function "$Fit_{best}[t]$ = `evaluate`(*m*, *Chroms*, *Fits*)" allocates the tasks into *m* processors in the order of each *Chroms*, and the corresponding fitness is set to *Fits*. It is described in **Fig. 10**. It is also used as an evaluation function which returns the best fitness (∈ *Fits*) in that generation *t*.

(b) Perform crossover operation. Crossover combines the features of two parent chromosomes to form two similar offspring by swapping corresponding segments of the parents. In this case, roulette strategy is used. Two-parent chromosomes are selected with a probability equal to the probability of fitness, and randomly select the crossover points, then swap the segments. The procedure "`crossover`(*Chroms*, *Fits*)" generates a set of new *Chroms* (allocation patterns) from the existing *Chroms*. *Fits* are a set of fitness correspond to a set of chromosomes *Chrome*, and new *Chroms* in the next generation is crossbred according to the ratio of the selection of *Fits* in crossover procedure.

(c) Perform mutation operation. Mutation alters one or more genes (positions in a chromosome) of a selected chromosome by a random change. It prevents a convergence to a local solution, which is far from the optimal one. The procedure "`mutation`(*Chroms*)" breaks into a mutation of *Chroms* at some probability.

(d) Function "`is_converged`(Fit_{best})" judges whether the search for a solution should stop or not, by checking the best fitness status Fit_{best} among generations is stable or not.

```
Function evaluate(m, Chroms, Fits)
m      : number of processors
Chroms : lists of task allocating order randomly
         generated (=chromosomes)
Fits   : set of fitness
{
  list    : task allocating order list (∈ Chroms)
  n_chrom : number of chromosomes
  al_best : the best allocation corresponds to th_best
  th_best : throughput of al_best
  th_ideal : ideal throughput
  fit_best : the best fitness among Fits

  th_best=0; //initialize th_best
  for i=1 to n_chrom {
    list = Chroms[i]; //allocating order
    al = allocate(list, m);
    th = get_throughput(al);
    Fits[i] = th/th_ideal;
    if th_best < th { //update
      th_best=th
      fit_best=Fits[i]
      al_best=al;
    }
  }
  return fit_best;
}
```

Fig. 10 Evaluation algorithm of chromosomes.

In the Global search algorithm, the patterns of allocation are extremely diverse and it is not practical to examine all the patterns, so we employ GA[5] for obtaining the practical allocation quickly. The GA-based algorithm consists of two stages from GA1 to GA2. In GA1, basic GA operations such as crossover and mutation are performed in the loop, and generation changes according to iteration of the loop. In GA2, the fitness of the chromosomes (=allocating order lists) in that generation are evaluated.

Table 1 The number of trials for deciding an optimal allocation.

	Global Search	GA-Knapsack
m	n_{trial}	n_{trial}
1	1	1
2	20!	1600
4	20!	1600

5.2 Throughput of Allocated Tasks

In the Stream-based Computing Model, the amount of data processed in unit time is important. So we use throughput as the evaluation factor of the performance.

Figure 10 is the evaluation algorithm of chromosomes in the GA-Knapsack method, and it consists of loop (1–n_{chrom}) of following steps:

(1) Generates allocation pattern from the allocating order list *list*. Function "`allocate(`*list*, *m*`)`" from *list*, and it is described in Section 4.2.1.

(2) Function "th=`get_throughput(`*al*`)`" returns the throughput of the allocation pattern *al*. The variable th_{ideal} used in the calculation of the fitness is the global value and it is calculated as

$$th_{ideal} = 1/T_{ideal} \qquad (6)$$

where T_{ideal} is the ideal execution period of the computation in Eq. (5).

(3) Fitness fit of the chromosome is calculated as follows:

$$fit = th/th_{ideal} \qquad (7)$$

where th is the throughput of the allocation pattern corresponds to the chromosome.

6. Evaluations

Here, we present the evaluation of this allocation method focusing on

- the degree of efficiency of the GA-based allocation, and
- comparison to another method (CP/MISF method).

6.1 Evaluation about Trials of Allocations

In this section, the comparison of the average number of trials for deciding the allocation pattern between Global Search Knapsack method and the GA-Knapsack method is discussed to get the same throughput. The comparison is shown in **Table 1**. With one trial, corresponding to the chromosome (in allocating order list), allocation pattern is generated using the GA-Knapsack method, and the throughput is evaluated. For one trial, when the number of operational processors is only one, theoretically, the processing time in any combination is the same, because dependency in the relationship only exists. Therefore, we define speedup rate, with this throughput as a unit, in case the number of trial is one. The GA-Knapsack method basically consists of the Knapsack allocation, mutation and crossover. Therefore, the amount of allocation time per one trial of the GA-Knapsack method is larger than the Global Search method in the amount of crossover and mutation. However, the time of crossover and mutation is small, so the one trial time of the Global Search method and the GA-Knapsack are almost the same.

In this comparison, 10 sorts of computations composed of 20 tasks with random connections are generated, and their average results are shown in Table 1. Tasks' processing cost is given as exponential random numbers with average 100 ms, and data size of the stream is set to 100 Bytes. On this evaluation, the number of populations in the GA-Knapsack method was 100. In Table 1, m is the number of processors for deciding the allocation, n_{trial} is the average number of trials for searching the solution. *speedup* is the average speedup ratio between m processor allocation to 1 processor allocation. It is calculated as

$$speedup = \frac{\text{throughput}(m)}{\text{throughput}(1)} \qquad (8)$$

throughput(m): throughput of the allocation on m processors, that means m divisions

throughput(1): throughput of the allocation on 1 processor

In the case, Table 1 shows the solution is given by too few number of tries (1600 = number of units 100 × number average generation 16). The *speedup* implies, the closer the value of *speedup* to m is, the higher the throughput is.

From this table, it is observed that in order to get the same speedup rate of the Global Search and the GA-Knapsack, the Global Search Knapsack method takes 20! trials, the GA-Knapsack takes only 1,600 trials. So the efficiency of applying GA on task allocation is shown.

6.2 Evaluation about Throughput of Computation

In this section, speedup rates are discussed comparing our scheduling method and the CP/MISF[4] method. **Figures 11** and **12** show comparisons between our proposed method and

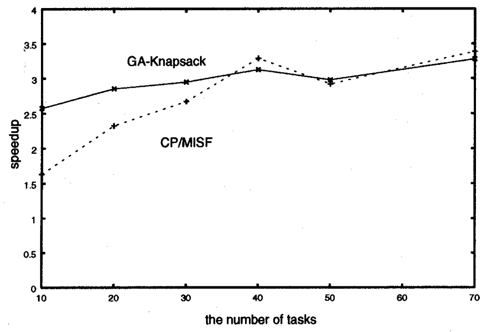


Fig. 11 Relations between number of tasks and *speedup* (number of processors = 4).

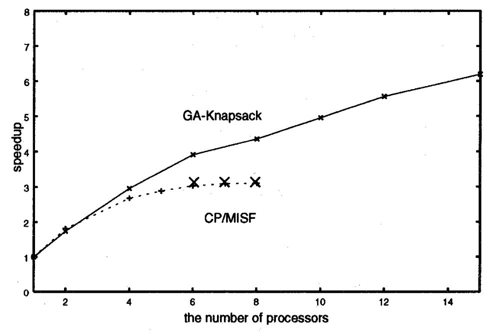


Fig. 12 Relations between number of processors and *speedup* (average of 10 computations).

CP/MISF. Figure 11 shows the relationship of the number of tasks and the speedup rate, where randomly generated tasks are allocated to four processors. Figure 12 shows the relationship of the number of processors. In Figs. 11 and 12 both, the estimated execution period in the computation of speedup is utilized, and the average of processing of 10 kinds is plotted. In the CP/MISF case, processing cost of the task is measured the same as our proposed method, execution period is estimated based on the cost, then the speedup rate is given. In our method, the GA condition to finish the search is as follows: The number of chromes is 500; the GA operation is terminated by the 10th generation, when the fitness is not improved. The average number of trials (n_{trial}) is 11,400, in case the number of tasks is 30 and the number of streams is 45. In Fig. 11, when the number of tasks is under 40 in case of cyclic processing, our method shows the superiority of the speedup. When the number of the tasks is just 40, the speedup rate of the CP/MISF exceeds our GA method slightly. The reason is that our proposed GA gets a local solution, instead of optimal solution. As the number of tasks increase, the difference between our proposed GA and CP/MISF decreases. This is why each processor load becomes balanced.

Figure 12 describes the relationship of the number of processors and the average speedup for 10 kinds of computations, consisting of randomly generated 30 tasks. This result shows that the proposed GA-Knapsack has 1.4 times speedup over CP/MISF, in case the number of processors is 8. In addition to that, in the CP/MISF of 6 to 8 processors, the scheduling period reaches to the length of its critical path. Critical path gives limitations of the minimum comutation time and the appropriate number of processors in the CP/MISF method. So no further improvement in speedup is expected, even if the number of processors is increased more than 6 in Fig. 12. On the other hand, the GA-Knapsack method shows better speedup rate, because of more parallelism and more divisions possible.

Time complexity for scheduling in the CP/MISF method is $O(m \cdot n + n^2)$ where m is the number of processors, n is the number of tasks. In the GA-Knapsack method, the complexity is $O(s \cdot n \cdot n_{trial})$, where s is number of streams, n is number of tasks, and n_{trial} is the number of trials). This is derived by experiment, in which the complexity is proportional to s when n and n_{trial} are fixed, and it is also proportional to n when s and n_{trial} are fixed.

7. Conclusions

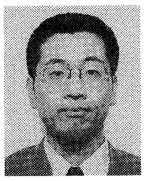
In this paper, we proposed the practical task allocation method adapting GA to choose the suitable allocation pattern quickly from a large number of possible choices of allocations. We used throughput prediction as the evaluation function of GA. In the evaluation, it is proved that the GA-Knapsack method can reduce the number of trials for task allocation in contrast to the Global Search Knapsack method. In addition, we proved that our GA-Knapsack method can obtain better allocation to the conventional CP/MISF method.

This method features applicability to various granularity programs. By bringing in a stream-based computing model, factors depending on system and application can be measured easily. On the other hand, the stream-based computing model has a few disadvantages: for example, stream-based computing is basically limited to pipeline processing and the programming methodology of this model is different from that of conventional procedural languages. However, those problems can be improved by the specific programming system supporting this model.

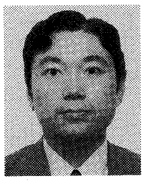
References

1) Aoyagi, Y., Uehara, M. and Mori, H.: Task Allocation Method Based on Static Evaluation for Heterogeneous Grained System, *Summer Workshop on Parallel Processing'97* (*SWoPP'97*), Vol.97, No.78, pp.7–12 (1997).
2) Aoyagi, Y., Uehara, M. and Mori, H.: A Case Study on Predictive Method of Task Allocation in Stream-based Computing, *The 12th International Conference on Information Networking* (*ICOIN-12*), pp.316–321, IEEE Computer Society (1998).
3) MPI Primer/Developing With LAM, OhioSupercomputer Center, The OhioState University (Nov. 1996).
4) Kasahara, H. and Narita, S.: An Approach to Supercomputing Using Multiprocessor Scheduling Algorithms, *Proc. IEEE 1st Int. Conf. on Supercomputing Systems* (1985).
5) Michalewicz, Z.: *Genetic Algorithms + Data Structures = Evolution Progrtams*, Springer-Verlag (1994).
6) Gaughan, P.T.: Data Streaming: Very Low Overhead Communication for Fine-grained Multicomputing, *Proc. 7th IEEE Symposium on Parallel and Distributed Processing* (1995).
7) Pospischil, G., Puschner, P., Vrchoticky, A. and Zainlinger, R.: Developing Real-Time Tasks with Predictable Timing, *IEEE SOFTWARE*, pp.35–44 (Sept. 1992).
8) Sakai, S.: Scheduling and Load Balancing in Parallel Computers, *J. IPS Japan*, Vol.27, No.9, pp.1031–1038 (1986).
9) Watts, J. and Taylor, S.: A Practical Approach to Dynamic Load Balancing, *IEEE Trans. Parallel and Distributed Systems*, Vol.9, No.3, pp.235–248 (1998).

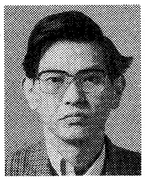
(Received January 5, 1998)
(Accepted July 1, 1999)



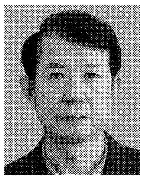
Yoichi Aoyagi received his M.E. and Ph.D. degrees from Toyo University in 1995 and 1999, respectively. He is currently working in Annex Information, Inc. His research interests are operating systems and programming languages. He is a member of IEEE, IPSJ and IEICE.



Minoru Uehara received his B.S. and M.S. degrees in electrical engineering from Keio University, Japan, in 1987 and 1989, and his Ph.D. degree in computer science from Keio University in 1995. He is currently an assistant professor of information and computer sciences at Toyo University. His research interests include distributed system, and programming language. He is a member of IEEE, ACM, and IPSJ.



Hideki Mori received M.S. and Ph.D. degrees all from Keio University in 1974 and 1978, respectively. He joined the faculty at Toyo University in 1978. He was a University Scholar of the Department of Computer Science of UCLA in 1984. He is a Professor of the Department of Information and Computer Sciences of Toyo University. His research interests include parallel architecture, parallel processing and fault tolerant computation. Most recently his research has emphasized on distributed parallel processing and fault tolerance systems. He is a member of IEEE, ACM, IPSJ and IEICE.



Akira Sato received Ph.D. degree from Waseda University in 1979. He joined the Research Laboratory of JNR in 1962, then he joined the faculty at Toyo University in 1984. His research interests include system simulation, parallel simulation and its applications. Most recently his research has emphasized on road traffic simulation, fuzzy graph modeling and its applications. He is a member of IPSJ and SICE.