A Case-Based Parallel Programming System

Katsuhiro Yamazaki and Shoichi Ando
Faculty of Science and Engineering, Ritsumeikan University
Nojihigashi, Kusatsu, Shiga, 525-8577 Japan
yamazaki@cs.ritsumei.ac.jp

Abstract

This paper describes how to reduce the burden of parallel programming by utilizing relevant parallel programs. Parallel algorithms are divided into four classes and a case base for parallel programming is developed by retrieving parallel programs in each class. Cases consist of indices, a skeleton, a program, parallelization effects and a history. Skeletons include the most important issues such as task division, synchronization, mutual exclusion, parallelization methods and threads. Parallel programs for image data storage, three dimensional spline, edge detection, thinning, knapsack problem and package wrapping algorithm are developed by retrieving the most relevant case and adapting it to the given problem. The experiment demonstrates that threads and synchronization can be reused from skeletons, and task division should be adapted by programmers.

1. Introduction

Although parallel computing is essential for solving large scale problems such as weather prediction and computer graphics, parallel computing is used only by experts. In order to spread parallel computing broadly, a breakthrough in parallel programming is essential, and various research activities that aim for reusability and portability are ongoing.

In parallel programming, load balancing among multiple processors and reduction of communication overheads between processors are most important. In addition, users are requested to fully understand parallelising methods of the target machine as well as task division and mutual exclusion of shared variables. C/Fortran and parallel directives are used in general and users have to specify task division and parallelization. Thus, much efforts are required for transforming from serial programs to parallel ones and for developing high performance parallel programs.

Meanwhile, Case-Based Reasoning(CBR) is expected to reduce the bottlenecks of knowledge acquisition, and has been applied for solving several practical problems such as legal judgment and fault diagnosis. Key issues in CBR include case representation, case retrieval and case adaptation. HYPO, a judgment system which supports decision making by referring to past similar cases, i.e. precedents, is the most successful system based on CBR [1].

This research investigates how to apply case-based reasoning to parallel programming and how to reduce the burden of parallel programming. This means that users reuse the structure of a relevant parallel program for a given problem as much as possible, and complete the program by supplementing it with additional information. This idea is generally used by primary and expert programmers, and thus the research aims to develop a practical parallel programming system by systematizing the idea.

Although the framework of the research is target machine and language independent, we assume that the target is C programming on a virtual shared memory parallel machine KSR1[6]. Since a single virtual address space is given in virtual shared parallel machines, the virtual shared memory paradigm can reduce the difficulties in parallel programming on distributed memory parallel machines.

In this research, the basic structures of parallel programs are prepared as skeletons. In order to retrieve similar skeletons for a given problem, the features of the problem are characterized as indices. Indices, a skeleton, a program, parallelization effects and a history construct a case which is stored into a case base. For a given problem, the system retrieves the most relevant case from the case base and tries to generate a parallel program by writing necessary information automatically or manually.

Parallel algorithms are generally classified into divide and conquer, processor farms, process networks and iterative transformation[5]. We have developed several parallel programs for each class and have implemented a case base for parallel programming[7][8]. In addition, we have not only investigated how to retrieve a relevant case and adapt it to a given problem, but also implemented the whole system on a Unix workstation.

Section 2 classifies parallel algorithms into four classes and shows parallel execution structures for each class.

Section 3 describes the system organization, the case base developed, and how to retrieve a relevant case and how to adapt it to a given problem. Section 4 describes how to develop a program for thinning by adapting a relevant case, bubble sorting. Section 5 evaluates the effectiveness of case-based parallel programming. Section 6 compares this research with related ones.

2. Classification of parallel algorithms

2.1. General classification of parallel algorithms

Parallel algorithms are generally classified into divide and conquer, processor farms, process networks and iterative transformation as shown in Figure 1. In divide and conquer, a problem is divided into subordinate problems, which are themselves recursively solved by dividing them further. The final result is obtained by recursively combining the solutions of the sub-problems.

In processor farms, a problem is divided into a number of independent computations, and the results of these computations are combined. The control of the operations is centralized, and the number of independent computations can be specified by the programmer or the number of available slaves.

Process networks are characterized by a division of computation into stages, with the data flowing through the stages. Stages can operate concurrently depending on the availability of input data. Thus, process networks are same as pipelining.

In iterative transformation, objects are transformed until the termination conditions are satisfied through several iteration steps.

2.2. BACS

BACS, Basel Algorithm Classification Scheme[2], was proposed to investigate methodological aspects such as algorithmic classification and complexity analysis. It tries to classify parallel algorithms in terms of mainly algorithm topologies, process structures, interaction mechanisms, data distribution and execution structures. Interaction mechanisms are classified whether it is done between two nodes (directly) or between three or more nodes(globally), and explicitly or anonymously. Mutual exclusion using mutex, for instance, is global implicit.

2.3. Parallel execution structures

KSR1 threads are based on POSIX standards. Threads are light weight processes and the overheads for generating them are small. Parallel processing is possible using multiple threads by allocating one thread to each processor.

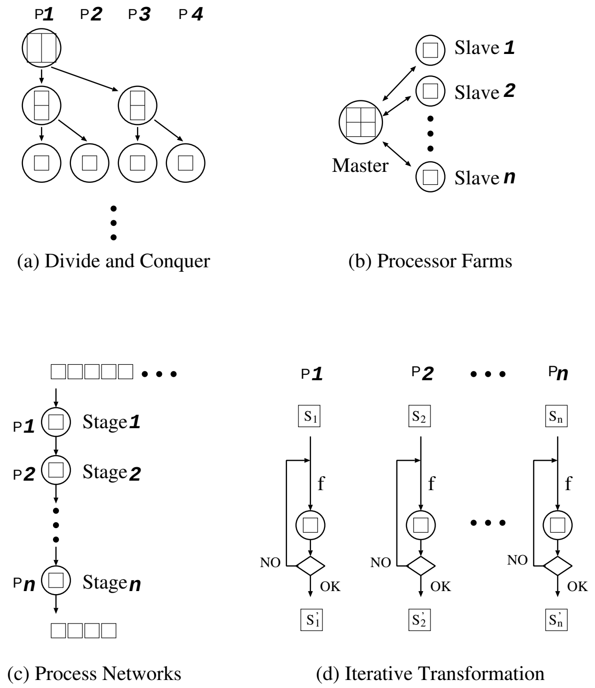


Figure 1. General classification of parallel algorithms.

Parallel regions, parallel sections and tiling are provided as parallelization methods in the KSR1. Multiple threads execute a same code segment in parallel regions and each thread executes each section in parallel sections. Thus, parallel regions correspond to SPMD and parallel sections correspond to MPMD. In addition, programmers can directly generate or remove threads using thread libraries.

The four classes of parallel algorithms are realized using parallel regions and thread libraries on the KSR1. Figure 2 shows the parallel execution structures of divide and conquer. In the parallel region type, the master activates a parallel region and multiple slaves are waiting for a task to be allocated. The master hands over a half of its task to the slave 1. Similarly each thread hands over a half of its task to an idle thread.

In processor farms, the master activates a parallel region and each slave executes a same task(Figure 3). In process networks, the master creates the top, middle and tail threads using thread libraries(Figure 4). Each thread is required to synchronize with the upper and lower threads. In iterative transformation, multiple threads are created using thread libraries and all threads execute a same calculation after having checked in the barrier A(Figure 5). They check out the barrier and the master judges if the termination conditions are satisfied. Two barriers are required, because slave threads which terminated the calculation can check out and check in again while the master is checking in. In divide and conquer, and processor farms, users can directly create and delete threads using thread libraries.

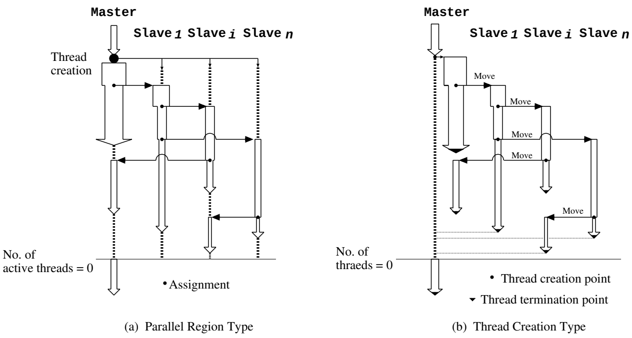


Figure 2. Divide and conquer.

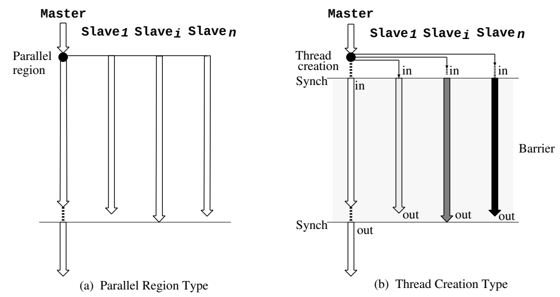


Figure 3. Processor farms.

3. A case-based parallel programming system

3.1. System organization

Figure 6 shows the system organization. In problem analysis, users define the application field and the specification of a given problem, and judge if there are data dependencies and termination conditions of loops. Based on these, users determine a class of parallel algorithms. Parallel structures for the selected algorithm are displayed and the user selects a parallel structure based on synchronization and parallelization methods.

The system retrieves the most relevant case from the case base using indices. Indices, a skeleton, a program and parallelization effects are presented to the user. The skeleton includes the most important issues such as task division, thread operations and synchronization. The user reuses thread operations and synchronization of the skeleton as much as possible, and adapts other issues such as task division to the problem.

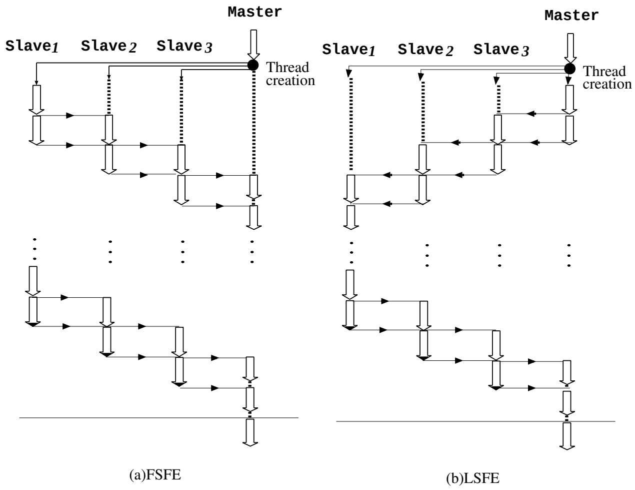


Figure 4. Process networks.

The system has been developed using Tcl/Tk on a Unix workstation. A case base, an indexing part, a case retriever, a case adaptater, and a user interface were implemented. The indexing part supports programmers to make indices by talking with the system. Figures and explanations for each index are displayed, and the programmer selects one of them. The case retriever retrieves the most relevant case to the given problem using indices and displays it to the programmer. Programmers can adapt the retrieved case using the case adaptater. The case base contains 22 parallel programs developed so far. Parallel programs are registered through the case registrar.

3.2. The case base

22 cases, i.e. parallel programs, were developed as shown in Table 1. A serial program was developed first, and then a parallel one was developed by considering parallelizing issues. Speedup was measured after the program was completed. Half of the cases were developed using relevant cases. In this case, the programmer completed the parallel program by supplementing the skeleton of the relevant case with additional information such as task division and unit calculations.

3.3. Indexing based on parallelization analysis

A user determines ten indices for a given problem by analyzing how to parallelize it. The user determines these

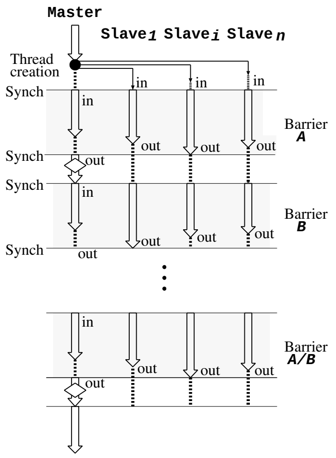


Figure 5. Iterative transformation.

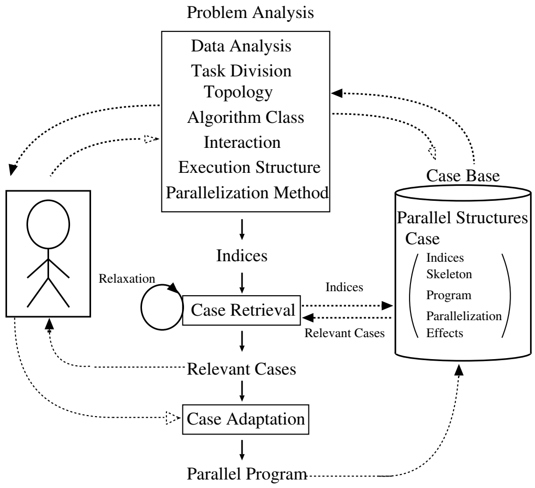


Figure 6. System organization.

Table 1. Case base.

Divide and conquer	Quick sorting, Integral calculation, Traveling salesman problem, Image data storage
Processor farms	Mandelbrot, String pattern matching (the KMP method, the BM method) Edge detection, Hough transform, Three dimensional spline, Ray-tracing, Vigenere cipher, Run-length encoding
Process networks	LU decomposition, Bubble sorting, Knapsack problem, Thinning
Iterative transformation	Solution of algebraic equations (the DKA method), Warshall algorithm, Package wrapping algorithm, Merge/split sorting, Romberg integration

indices in the following order that is same as human programming.

1. Applications: the user selects one from sorting, routing, graphs, image processing, string pattern matching, numerical calculations and the others.

2. Specification: definition of the problem is described using formulas or sentences.

3. Data structures: source and result data are determined in terms of (a) arrays or others, (b) one dimension, two dimensions or three dimensions, and (c) integer, floating, characters or structures.

4. Task division: partitioning and distribution of source and result data are determined. Partitioning is selected from an element, a raw, a column, elements, raws, columns and nothing. Distribution is selected from block, cyclic and copy.

5. Termination conditions: if the number of iterations of loops is variable, the conditions are described.

6. Topology: selects one from worker, tree, pipe, mesh, ring, master & worker, hypercube and others. Their structures are displayed on a screen.

7. Algorithms: selects one from divide & conquer, processor farms, process networks and iterative transformation based on data dependencies and termination conditions.

8. Parallelization methods: selects one from parallel regions, parallel sections, and pthread library.

9. Interaction: selects one from signal/wait, barriers, mutexes and nothing. The mechanism of these interaction are displayed.

10. Parallel execution structures: The structures for a selected algorithm and BACS expressions are displayed. The user selects one based on parallelization methods and iteration which the user wants to use.

8,9 and 10 should be determined in a body, because they are interrelated.

The definition of a given problem is described in the applications and specification. Typical data structures, task division, termination conditions, and topologies are given by the system using figures and explanations. The user selects one of them in each index or describes it. Arrays and structures are candidates as data structures. In task division, data partitioning and data distribution should be determined. For example, raw-wise, column-wise and mesh are possible for two dimensional arrays. Each partitioned element may be distributed in blocks or in cyclic. Sometimes the whole data may be copied.

Typical methods of algorithm class, parallelization methods and interaction are presented by the system, and thus the user can select one of them. The algorithm class that is the key issue in parallel programming is determined based on data dependencies and the termination conditions of loops. For example, processor farm can be used if no data dependencies and no termination conditions. Process networks can be applied if there are data dependencies but no termination conditions. Since the typical parallel execution structures are stored into the case base for each algorithm class, those for the determined algorithm class are presented to the user. The user selects the most suitable execution structure based on synchronization and the parallelization method. BACS execution structure is given by the system.

3.4. Case retrieval

This system retrieves the most relevant case by checking each index of a case. If all indices of a case match with those of a given problem, the system retrieves it, otherwise the system retrieves a relevant case by relaxing the conditions of indices.

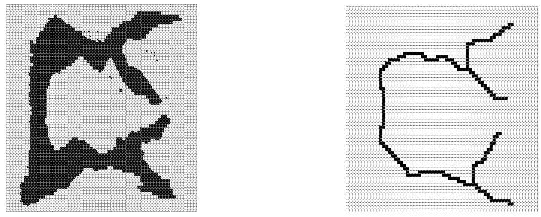
Relaxing of conditions is done by deleting an index one by one. The system retrieves a relevant case until the remaining indices of a given problem match with those of a case. The order of deleting indices is applications, specification, termination conditions, result data, source data, task division of result data, task division of source data, interaction, parallelization methods, topology and algorithms. An index that affects the structure of skeletons is deleted later in this order. This allows high reusability of skeletons and reduces the burden of parallel programming.

In addition, a case of same algorithm class is retrieved definitely in the worst case, because the algorithm class of a given problem is selected one of four classes. This is most important in that the user can reuse the main program of a case due to the same structure of parallel programs.

3.5. Case adaptation

Case adaptation is done by the programmer. Although the skeleton of a retrieved case can be almost reused, some adaptation is required in order to develop a final parallel program. Here we classify parallel programs into four parts that are threads, interaction, task division and unit calculations. Threads and interaction would be almost reused, because the algorithm class and the execution structure of a relevant case matches with those of a given problem. This means that the bone structure, that is the most important issue of parallel programs we believe, can be reused.

Task division can be reused to some extent if indices of task division are same. Otherwise, most of them should be adapted or newly described. Unit calculations seem to be almost reused from a serial program. In addition, a user can refer to the program body and speedup of a relevant case. Thus, the user can predict how much speedup the new program obtains.



(a)Original Image (b) Thinned Image

Figure 7. Thinning.

4. Thinning:an example of case adaptation

4.1. Definition

Detect a central line from a binary image by thinning the width of its area as shown in Figure 7.

4.2. Analysis of parallelization

Source and result $data$ are two dimensional array of integers. Figure 8 shows 17 mask patterns that consist of 3×3 grid. If the image matches with the patterns, the value of central point is changed from one to zero. The image plane is partitioned into blocks as raw-wise and column-wise.

As shown in Figure 9, each thread matches one block downward, rightward, upward and leftward and this process is repeated. A wavefront strategy which calculates from the left top to the right down is used. Process networks can be used, because values obtained by the previous match are used at the current match. Thus, the top left block(1) is processed by processor p1, and then two neighboring blocks(2) are processed by processors p1 and p2 simultaneously and so on. The topology is pipe. Threads are generated using thread libraries and they are synchronized with signal/wait.

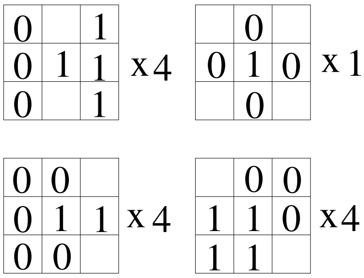


Figure 8. Mask patterns.

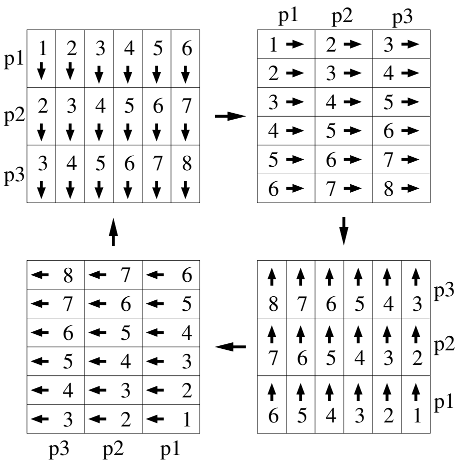


Figure 9. Task division.

Indices of thinning are shown in Table 2. Items with a circle are same in thinning and bubble sorting. Items with a triangle are similar in the two.

4.3. Bubble sorting:a relevant case to thinning

Bubble sorting is retrieved as the most relevant case to thinning. The smallest data, a bubble, comes to the top and this is repeated until all data are sorted as shown in Figure 10. Initial data are divided by blocks of elements. Process networks can be used due to pipeline processing. Top thread, middle threads and tail thread are created by thread libraries and synchronized with signal and wait.

4.4. Case adaptation

The skeleton of bubble sorting consists of main, top thread, middle thread and tail thread. The skeleton except the tail thread is as follows.

Table 2. Indices of thinning.

Applications	Image processing
Specification	Detect a central line from a binary image
△ Data structures	
	Source: Int. 2-dim array $data$
	Result: Int. 2-dim array $data$
△ Task division	
	$data$, blocks of rows/columns, block
◯ Termination conditions	-
◯ Topology	Pipe
◯ Algorithms	Process networks
◯ Parallelisation methods	Thread library
◯ Interaction	Signal & wait
◯ Execution structure	
top_thread : $Fix\{C, I^{p(lower\_thread)}\}$	
middle_threads : $Fix\{I^{p(upper\_thread)}, C, I^{p(lower\_thread)}\}$	
tail_thread : $Fix\{I^{p(upper\_thread)}, C\}$	

```
main
  {
   input data;   input the number of threads;
   initialize flag variables;
   for ( no. of threads − 2 ) create threads
   create a thread( tail_thread, thread no. )
   top_thread( 0 );
   print the results;
  }
top_thread( thread no. )
  {
   set the upper and lower edges;
   while ( upper edge ≤ lower edge )
     {
      while ( lower edge ∼ upper edge )
        move smaller values to the upper;
      adjust the upper edge;
      synchronize with a lower thread;
     }
  }
middle_thread( thread no. )
  {
   set the upper and lower edges;
   while ( 1 )
     {
      synchronize with an upper thread;
      compare the lowest value with the highest
      value of the upper thread;
      synchronize with an upper thread;
```

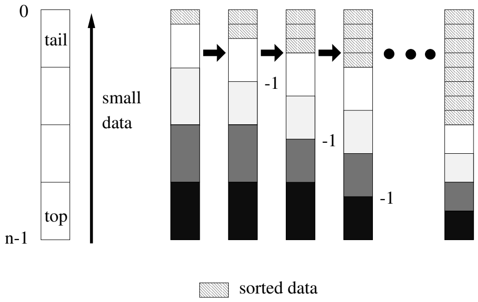


Figure 10. Bubble sorting.

```
    while ( lower edge -1 ~ upper edge )
      move smaller values to the upper;
    synchronize with a lower thread;
  }
}
```

The main is almost reused except task generation, and the order of thread creation is altered. In the bubble sorting, middle, tail and top threads are generated in this order(LSFE in Figure 4). This is altered in the order of top, middle and tail(FSFE in Figure 4). Initialization, collection of results and unit calculations are reused from a serial program. Synchronization with the upper and lower threads are reused from the skeleton. Task division should be adapted by the programmer. Parallelization effects of thinning is shown in Figure 11.

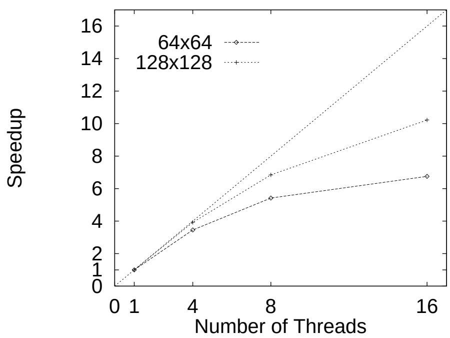


Figure 11. Parallelization effects of thinning.

5. Evaluation of case-based parallel programming

Six programs were developed by adapting relevant cases. Image data storage was developed using quick sorting.

Table 3. Reusability of cases.

(No. of lines)

		Threads	Synch.	Task Division	Unit Calc.
A	Reuse	30	9	3	22
	Adapt	0	0	18	0
	New	0	0	0	0
B	Reuse	8	1	0	107
	Adapt	0	0	11	3
	New	0	0	12	0
C	Reuse	8	1	6	60
	Adapt	0	0	0	5
	New	0	0	15	0
D	Reuse	7	36	0	222
	Adapt	4	0	24	0
	New	0	5	8	0
E	Reuse	11	9	9	32
	Adapt	0	0	9	0
	New	0	0	12	0
F	Reuse	10	7	0	67
	Adapt	0	0	2	4
	New	0	0	0	18

A: Image data storage, B: Three dimensional spline, C: Edge detection, D: Thinning, E: Knapsack problem, F: Package wrapping algorithm

Three dimensional spline was developed using the KMP method, edge detection was developed using three dimensional spline, knapsack problem was developed using LU decomposition, and package wrapping algorithm was developed using the DKA method. We have evaluated the reusability of cases for these six programs. Table 3 shows the number of lines that show the reusability of skeletons. Threads, synchronization and task division are reused from skeletons, while unit calculations are reused from serial programs.

Threads and synchronization can be completely reused in five programs. In thinning, the order of generating threads were altered and five synchronizations between the top thread and the tail thread were inserted to confirm the terminations of four direction calculations.

In terms of task division, image data storage, edge detection and knapsack problem can reuse the skeletons in part, but adaptation and new descriptions were required in general. Most of the unit calculations were reused from the serial programs. In package wrapping algorithm, shared variables were altered to local variables in each thread.

We believe that it is most important to decide which algorithm class can be applied to a given problem, and which execution structure can implement the algorithm.

This means that it is important and difficult for programmers to decide the most suitable execution structure and how to implement the structure using threads and synchronization. If thinning is registered to the case base, the case is reused. Hence, threads and synchronization can be almost reused from skeletons, provided that the case base is enriched. This means that the execution structure can be supported. If these structures are determined, the user just insert unit calculations and variable initializations into the program from a serial program. Since task division requires case adaptation, a supporting mechanism should be considered.

6. Related work

Cole proposed algorithmic skeletons, each of which describes the structure of a particular style of algorithm [3]. The divide and conquer skeleton, the iterative combination skeleton, the cluster skeleton and the task queue skeleton were presented. These skeletons serve to highlight "good" algorithmic style at a high level, encouraging the user to describe solutions well suited to parallel implementation.

Skeleton-oriented programming(SOP) was proposed based on BACS. Algorithmic skeletons, which are completed to a full program by the insertion of the calculation parts, are used as scalable execution and coordination patterns. In this approach, the synchronisation of processes is already specified in the main structure of the algorithm. Skeleton-oriented programming is very close to our research in that both ideas utilize algorithmic skeletons. The main differences between SOP and our research relate to program portability and skeleton retrieval. In SOP, a program skeleton which is the algorithm for the chosen programming language and the given virtual machine, is generated from algorithmic skeletons in order to realize program portability. However, users have to select the appropriate skeleton from the database. On the other hand, in our approach, relevant cases are automatically retrieved from the case base, although program portability is not considered.

Darlington et al. presented a methodology which is a compromise between the extremes of explicit imperative programming and implicit functional programming [4]. They use a repertoire of higher-order parallel forms, skeletons, as the basic building blocks for parallel implementations and provide program transformations which can convert between skeletons, giving portability between differing machines.

7. Conclusions

This paper has described how to reduce the burden of parallel programming by utilizing relevant parallel programs. We have developed a case-based parallel programming system which allows to retrieve the most relevant case from a case base and adapt it to a given problem.

The skeletons of cases include threads, synchronization and task division that are the most important and difficult issues of parallel programs. Image data storage, three dimensional spline, edge detection, thinning, knapsack problem and package wrapping algorithm were developed using relevant cases. The experiment showed that threads related issues and synchronization can be reused from the skeletons, and task division requires case adaptation by programmers.

We have been developing other new programs using relevant cases to verify the effectiveness of this system. Simplification of task division and automatic case adaptation should be investigated in the future.

Acknowledgments

The authors would like to thank heartily Professor John R. Gurd and Professor Haruo Niimi for valuable comments and discussions. We are grateful to Canon Supercomputing S.I.Inc. for allowing us to use the KSR-1.

References

[1] K. D. Ashley and E. L. Rissland. A case–based approach to modeling legal expertise. *IEEE EXPERT*, 3(3):70–77, Fall 1988.

[2] H. Burkhart et al. Bacs:basel algorithm classification scheme. Technical report, Universitat Basel, 1993.

[3] M. Cole. *Algorithmic Skeletons:Structured Management of Parallel Computation*. Pitman, 1989.

[4] J. Darlington et al. Parallel programming using skeleton functions. In *Parallel Languages and Architectures, Europe: Parle '93*, pages 146–160, 1993.

[5] F. A. Rabhi. Exploiting parallelism in functional languages:a paradigm-oriented approach. In *Second Workshop on Abstract Machine Models for Highly Parallel Computers*, pages 1–14, 1993.

[6] K. S. Research. *KSR Parallel Programming*, 1991.

[7] K. Yamazaki, S. Ando, and H. Asakura. A case-based parallel programming system. In *Joint Symposium on Parallel Processing(JSPP'97)*, pages 117–124, 1997.

[8] K. Yamazaki, K. Matsuda, and S. Ando. Case-based parallel programming. In *IPSJ 38th Programming Symposium*, pages 155–165, 1997.