C Parallelizing Compiler on Local-network-based Computer Environment

Kouichi ASAKURA, Toyohide WATANABE and Noboru SUGIE
Department of Information Engineering
School of Engineering, Nagoya University
Furo-cho, Chikusa-ku, Nagoya, 464-01, JAPAN

Abstract

Recently, the local-network-based computer system, in which some workstations are connected through the communication medium, is coming into practical use. The software development on this system is, however, very difficult for end-users because this system has complicated problems such as load balancing means, communication method among processes on different workstations and so on. In this paper, we propose a C-specific parallelizing compiler on this system in order to solve the above problems. Our compiler adopts the function call parallelization, and takes less communication overhead than DO loop parallelization.

1 Introduction

The parallelizing compiler, which parallelizes a sequential program so as to be suitable to various types of computer organizations, have been progressively studied for the improvement of system performance and processing efficiency [1, 2]. Although many parallelizing compilers were developed, many of them are related to FORTRAN programs. In FORTRAN parallelizing compilers, the parallelization of DO loop structure is the main issue of interest [3]. Many program restructuring methods such as node splitting, loop collapsing and so on, were proposed with a view to making the execution of DO loop structures efficient [4, 5].

These parallelizing compilers are mainly developed as the supercomputer-oriented paradigm. However, the current status in our computer environment is shifting over from centralized processing systems to distributed processing systems. Now that local-network-based computer systems, on which several workstations are loosely connected with a communication medium, are coming into practical use everywhere, the parallel/distributed processing mechanism for controlling different tasks simultaneously is more or less required.

In this paper, we address a parallelizing compiler for local-network-based computer environments. In particular, the program partitioning and restructuring methods are mainly discussed. Our compiler divides a sequential C program into several self-organized processes to be executed in parallel. The parallelizing compilers for C have been never investigated so actively. Although R.Allen et al. reported on a C-specific parallelizing compiler, they did not always investigate the compiling technique for C directly from a parallelizing point of view [6].

The rest of this paper is organized as follows. Section 2 shows the environment in which our parallelizing compiler is designed and overviews our compiler. Section 3 describes the program analysis phase and also defines some terms used in the following phases. We discuss the program partitioning algorithm and program restructuring algorithm with respect to the process generation and process allocation problems in Sections 4 and 5. Finally, the conclusion is reported in Section 6.

2 Distributed Environment and Parallelizing Strategy

Our C parallelizing compiler is designed and implemented on a distributed computer environment, in which several loosely coupled workstations are connected through the local area network. This distributed environment is different from the computer system on which the traditional parallelization researches have tried. In the traditional system, the communication costs such as data access times and synchronization overheads are disregarded or may be not problematic since processes in each processor are the same, every processor is accessible instantly to requisite data in the commonly shared memory, or processors can synchronize mutually through the high-speed communication channel.

However, the communication cost is one of the most important factors in our local-network-based computer environment because every processor can manage its own processes independently. Additionally, any particular hardware supporting equipment is not assumed. Therefore, our parallelizing compiler must decompose C programs in accordance with the network

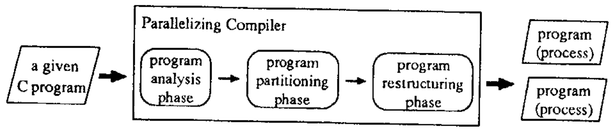

Figure 1: Configuration of our compiler

architecture optimally under the trade-off of the communication cost, the process granularity and the availability of processors.

The processing phases in our parallelizing compiler are shown in Figure 1. The program analysis phase analyzes firstly C programs, like the ordinary compiler. This phase aims mainly to check up the interdependency relationship among individual statements in order to examine the program fragments that can be executed independently. Next, the program partitioning phase analyzes the behavior of C programs with respect to the function calling relationship, and then partitions them into independently executable program units. Finally, the program restructuring phase constructs individually partitioned program fragments as completely self-organized subprograms.

3 Program Analysis

It is important to analyze the program structures with respect to the dependency among variables, calling sequences of functions, execution time of each function and so on, when we adopt the function-based program partition strategy. The analysis technique is almost similar to those assessed by the traditional compilation approaches or FORTRAN parallelization approaches [7, 8]. Also, concerning the construction of FORTRAN parallelizing compiler, the same view about program parallelization is already reported in [9]. However, we must pay attentions to the difference between C and FORTRAN. In FORTRAN, the parameter passing method for calling subroutines is "call by reference." Whereas, C applies "call by value" to the parameter passing for functions. As for our objective for parallelizing compiler, this difference shows that C is more manageable than FORTRAN. However, C is more complicated in comparison with FORTRAN because many syntactic characteristics are very flexible: that's, the scopes of variables, pointer variables, various data types, and so on.

Function call graph

This graph represents the calling structure among functions from a static point of view. The node indicates a function and the edge corresponds to the invoking relationship among functions. Figure 2(a) is an example of this graph. When we take notice of both ends of an edge, the upper node is defined as a parent and the lower node is a child. This graph does not only represent the global structure of program, but also is an analytic tool with respect to program partition. This function call graph is usually composed as a network-shaped form, as shown in Figure 2(a), because the same function may be often invoked from different other functions. The network-shaped form is not appropriate as the resource for the program partitioning phase. Thus, this form will be transformed into a tree-shaped form, as illustrated in Figure 2(b).

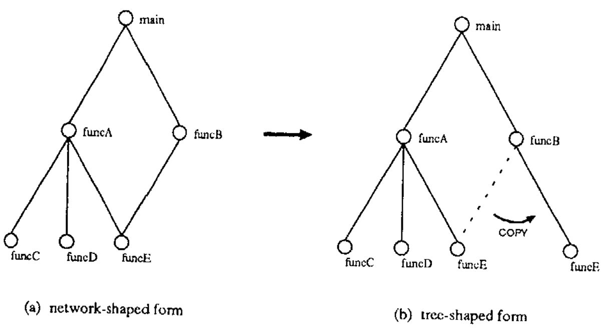

Figure 2: An example of function call graph

Estimated execution time of function

The execution time of each function takes an important role in order to divide a program into the fragments. However, it is difficult to estimate the execution time of function exactly even if the function is evaluated practically. H. Honda et al. reported an estimation method for the execution time of statements by counting the machine instructions in OSCAR project [10]. While, in our estimation, the execution time of function is basically dependent on the analytic evaluation for source codes of program. Namely, the execution time of function is approximately estimated as the number of tokens used in the function. Of course, it is necessary to estimate the execution times of individual functions in accordance with the behavior of program in the case that more precise results are required.

Connectivity among statements

Each statement of program is not always independent of other statements. Some statements must be executed before other statements. The execution order of statements is determined through the variable analysis. Namely, two statements that refer to or update the same variable must be controlled so as not to be executed simultaneously and the execution order of these statements cannot be absolutely changed. Also, the relocatable program region is calculated from these data. The relocatable program region is a segment surrounded with the upper and lower statements that have a connectivity relationship. A statement can be relocated somewhere in the relocatable program region with the same result. For example, the following program fragment has five statements.

```
1: a = b + c;
2: b = e + f;
3: g = a + h;
4: i = d + j;
5: k = g + i;
```

The execution order of this fragment is defined by the variable analysis: $1 \rightarrow 3 \rightarrow 5$ and $2 \rightarrow 4 \rightarrow 5$. Also, the relocatable program region of statement 3 is between 1 and 5: that is, the statement "`g = a + h;`" can be located somewhere in this relocatable program region without any change of the computation result.

Connectivity among functions

The dependency among mutually related functions is one factor to judge the behavior of program. If a function depends on another function too strongly, they are individable as independent processes. Namely, the dependency represents the possibility of the parallel execution of functions.

The dependency is defined as a numerical value, and the range is $0 \leq x \leq 1$ (x is the degree of dependency). The dependency value among two arbitrary functions is specified by the following equation.

$$\mathrm{DV}(f_i, f_j) = 1 - \frac{\text{simultaneously intersected execution time between } f_i \text{ and } f_j}{\text{execution time of } f_i},$$

where f_i is a parent and f_j is a child function. The numerator of the right side is calculated by the information of estimated execution time and connectivity among statements.

Number of function calls

This denotes how often a function is called by other functions. We cannot acquire the exact number of function calls at the compile step. Thus, this is estimated on the basis of the number of function call statements appearing statically in a program : for example, the number may be possibly approximated by the repeating number for the function call statements in the loop structure.

4 Program Partitioning

It is desirable that task loads assigned to individual processors are equally partitioned, corresponding to the computer environment. In the program partitioning phase, the parallelizing compiler must divide a whole program into simultaneously executable program fragments. The task granularity in our parallelization subject is specified appropriately with respect to the communication traffics among available program fragments, in addition to the concurrency among program fragments and number of available processors in the network environment. At least, the trade-off issue between communication traffic and process concurrency is important, in spite of difficult estimation factors, in order to make our parallelizing execution successful.

Our program partitioning strategy works as follows: first, looking upon individual functions as the minimum program fragments, and then merging the dependent functions into one process on the basis of the evaluation of function call graph and other analytic information. This program partitioning algorithm is shown in Figure 3. The input parameters of this algorithm are as follows: $t(i)$ is the estimated execution time of function f_i described in the section 3. $d(i,j)$ is the connectivity relationship between functions f_i and f_j, which is shown as $DV(f_i, f_j)$ in the previous section. $c(i,j)$ is the number of call statements of f_j in f_i, which is derived from the number of function calls. These variables are assigned to individual nodes and edges in our function call graph. $T(i)$ represents the totally estimated execution time of sub-tree whose node i is the root. $finish(i)$ is a flag variable to indicate whether the partitioning processing for f_i did finish or not.

This partitioning algorithm is effective from the bottom nodes to the upper nodes according to the function call graph. Namely, the mutually related functions, which correspond to nodes, are merged as a process, which is a sub-tree. The process composition procedure compares the estimated execution time $T(i)$ of a sub-tree with $T(j)$ of another sub-tree, and then judges whether the sub-tree with $T(j)$ can be merged into that of $T(i)$. This determination is performed in the expression (1). This is because it is effectual to merge two sub-trees if the shortened time is smaller than the overhead time (which is represented as OH in our algorithm). Figure 4 indicates the reducible execution time among two concurrently executable functions. In Figure 4(a), the reducible execution time is a part of $T(j)$ $(= t(i) \times (1 - d(i,j)))$. Whereas, in Figure 4(b), all of $T(j)$ are reduced when $T(j)$ is shorter than

```
INPUT
  • Function call graph.
  • $t(i)$ : estimated execution time of function $f_i$ .
  • $d(i,j)$: degree of connectivity between functions $f_i$ and $f_j$ $(i \neq j)$.
  • $c(i,j)$: number of function calls from function $f_i$ to $f_j$ $(i \neq j)$.
OUTPUT
  simultaneously executable processes as sets of mutual functions derived
  from function call graph.
ALGORITHM
  for each function $f_i$                ; initialization
    if function $f_i$ is leaf in function call graph then
      $T(i) \leftarrow t(i)$.
      $finish(i) \leftarrow TRUE$.
    else
      $finish(i) \leftarrow FALSE$.
    endif
  endfor
  $CS \leftarrow \{\}$.                  ;CS means Candidate Set.
  for each function $f_i$
    if $finish(i) = TRUE$ for all $f_j$ is child of function $f_i$ then
      add node $j$ to $CS$.
    endif
  endfor
  while $CS \neq \{\}$ do                ; main loop
    choose and remove some node $i$ in $CS$.
    $T(i) \leftarrow t(i)$.
    for each $f_j$ is child of function $f_i$
      shortened-time $\leftarrow \min\{T(j), t(i) \times (1-d(i,j))\}$.   ··· (1)
      if shortened-time is longer than $OH$ then
        output "The edge between function $f_i$ and $f_j$ is cut off."
        increasing-time $\leftarrow \max\{0, T(j) + OH - t(i) \times (1-d(i,j))\}$.
                                                                          ··· (2)
        $T(i) \leftarrow T(i) +$ increasing-time $\times c(i,j)$.
      else
        $T(i) \leftarrow T(i) + T(j) \times c(i,j)$.
      endif
    endfor
    $finish(i) \leftarrow TRUE$.
    for each $f_j$ is parent of function $f_i$
      if $finish(k) = TRUE$ for all $f_k$ is child of function $f_j$ then
        add node $j$ to $CS$.
      endif
    endfor
  enddo
```

Figure 3: Program partitioning algorithm

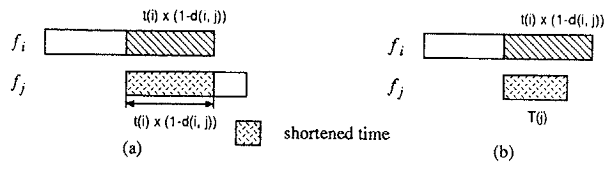

Figure 4: Evaluation in expression (1)

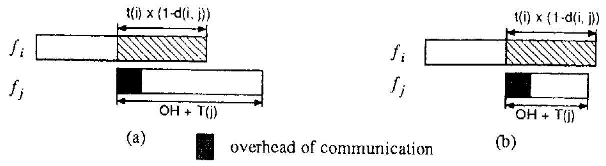

Figure 5: Evaluation in expression (2)

the estimated reducible execution time.

After the merging procedure, the execution time of sub-tree whose root node is f_i is calculated by the expression (2). The elapsed time for the execution of f_j is $T(j)+OH-t(i)\times(1-d(i,j))$ since the shortened time for parallel execution is $t(i)\times(1-d(i,j))$ as shown in Figure 5(a). On the other hand, the execution of f_j does not cause the increase of time if $T(j)$ is very small, and this expression becomes negative as shown in Figure 5(b).

5 Program Restructuring

The program fragments, which were separated in the program partitioning phase, are not always well organized as cooperative processes. Namely, they are a set of independent functions, but have not control mechanisms to synchronize with each other, transfer common data in coordination and so on by themselves. Therefore, in order to make up these program fragments as autonomous processes, they must be structurized sufficiently so as to generate together the same result that the original serial program puts out.

In this program restructuring phase, the cooperative processes to be executed independently on individual processors are composed on the basis of the function call graph and information about the connectivity among statements. The following algorithm is applied to this program restructuring: relocating statements, creating another process and inserting synchronous statements for function call statements.

i) Find out the program region in which a function call statement can be relocated , using the connectivity among statements.

ii) Move the function call statement to the upper location. Next, rewrite the program fragment so as to create a new process on another processor.

iii) Insert synchronous statements into the lower location of the relocatable program region.

Here, we show a brief example of program restructuring in Figure 6. In this example, we assume that

```
 1: int funcA(x, y)
 2: int x, y;
 3: {
 4:     int a, b, c, d;
 5:     int i;

 6:     a = funcB(x, y);
 7:     b = funcC(x);
 8:     c = b + x;
 9:     for(i=0;i<x;i++)
10:         c += x;
11:     b = funcC(y);
12:     c = c + b + y;
13:     for(i=0;i<y;i++)
14:         c += y;
15:     d = funcD(a);
16:     c += d;

17:     return c;
18: }
```

(a) original program

```
 1: int funcA(x, y)
 2: int x, y;
 3: {
 4:     int a, b, c, d;
 5:     int i;
        int sync = 0;

 6:     a = funcB(x, y);
        remotefunc(&sync, &d, FUNCD, a);
 7:     b = funcC(x);
 8:     c = b + x;
 9:     for(i=0;i<x;i++)
10:         c += x;
11:     b = funcC(y);
12:     c = c + b + y;
13:     for(i=0;i<y;i++)
14:         c += y;
        while(!sync)
            ;
16:     c += d;

17:     return c;
18: }
```

(b) rewritten program

Figure 6: An example for program restructuring

functions funcB and funcC are executed as subroutine calls, and that the function funcD creates a new process. We firstly find out two upper and lower statements for the relocatable program region with respect to the function call statement "d = funcD(a);." Thus, we must look for the statements that refer to or update the variable a or d because this function call statement refers to the variable a and updates the variable d. So, "a = funcB(x, y);" in line 6 and "c += d;" in line 16 are found. This relocatable program region is between lines 6 and 16. Next, this call statement is moved to the upper location of this region: after line 6 and before line 7, and then rewritten so as to create a new process. Here, remotefunc is a function for creating the new process on another processor. It is served as a library. Finally, the synchronous statement is inserted into the lower position: after line 15 and before line 16. The variable sync is a synchronous variable. This variable is reset initially to zero and changed by the software interruption when the process for funcD is finished. The subroutine for software interruption is also served as a library.

6 Conclusion

In this paper, we proposed a C-specific parallelizing compiler for local-network-based computer systems. Our compiler has the following features:

- The call statement parallelizing method is adopted. This method makes it possible that function call statements are executed simultaneously. This parallelizing method takes less communication time than DO loop parallelizing method;
- The program partitioning algorithm and program restructuring algorithm for function call statement parallelization are proposed. Especially, the program partitioning algorithm enforces to execute the statements in parallel only when the parallel execution of these statements reduces the total run time.

Acknowledgements

We are very grateful to Prof. T. FUKUMURA of Chukyo University, and Prof. Y. INAGAKI and Prof. J. TORIWAKI of Nagoya University for their perspective remarks, and also wish to thank Ms. K. SUGINO and our research members for their many discussions and cooperations.

References

[1] D. J. Kuck, E. S. Davidson, D. H. Lawrie and A. H. Sameh: "Parallel Supercomputing Today and the Cedar Approach," *Trans. on IEICE of Japan*, Vol.J71-D, No.8, pp.1361–1374 (1988).

[2] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung and D. Schouten: "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors," *Proc. of the 1989 Int'l Conf. on Parallel Processing*, Vol.2, pp.39–48 (1989).

[3] R. Cytron: "Doacross: Beyond Vectorization for Multiprocessors (Extended Abstract)," *Proc. of the 1986 Int'l Conf. on Parallel Processing*, pp.836–844 (1986).

[4] D. A. Pauda and M. J. Wolfe: "Advanced Compiler Optimizations for Supercomputers," *Comm. of the ACM*, Vol.29, No.12, pp.1184–1201 (1986).

[5] S. P. Midkiff and D. A. Pauda: "Compiler Generated Synchronization for Do Loops," *Proc. of the 1986 Int'l Conf. on Parallel Processing*, pp.544–551 (1986).

[6] R. Allen and S. Johnson: "Compiling C for Vectorization, Parallelization, and Inline Expansion," *Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pp.241–249 (1988).

[7] D. E. Maydan, J. L. Hennessy and M. S. Lam: "Efficient and Exact Data Dependence Analysis," *Proc. of the ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation*, pp. 1–14 (1991).

[8] G. Goff, K. Kennedy and C.-W. Tseng: "Practical Dependence Testing," *Proc. of the ACM SIGPLAN '91 Conf. on Programming Language Design and Implementation*, pp.15–29 (1991).

[9] R. Triolet: "Direct Parallelization of Call Statements," *Proc. of the SIGPLAN '86 Symposium on Compiler Construction*, pp.176–185 (1986).

[10] H. Honda, S. Mizuno, H. Kasahara and S. Narita: "Parallel Processing Scheme of a Basic Block in a Fortran Program on OSCAR," *Trans. on IEICE of Japan*, Vol. J73-D-I, No.9, pp.756–766 (1990) [in Japanese].