Hybrid Approach for Parallelization of Sequential Code with Function level and Block level Parallelization

K.Ashwin Kumar, Aasish Kumar Pappu, K. Sarath Kumar and Sudip Sanyal
{kakumar_b03,akpappu_b03,kskumar_b03,ssanyal}@iiita.ac.in

***Abstract*— Automatic parallelization of a sequential code is about finding parallel segments in the code and executing these segments parallely by sending them to different computers in a grid. Basically, parallel segments in the code can be found by doing block level analysis, instruction level analysis or function level analysis. Block is any continuous part of the code that performs a particular task. This paper talks about a hybrid approach that combines the block level analysis with functional level analysis for parallelization of sequential code and its illustrates its advantages over block level parallelization and function level parallelization performed independently. In this approach, segments of code are identified as basic blocks. These blocks are analyzed to identify them as parallelizable or dependent. Loops which are also identified as blocks are parallelized using existing loop parallelization techniques [1]. This information would be used for automatic parallel processing of the set of independent blocks on different nodes in the grid using Message Passing Interface(MPI). The system will annotate the MPI library functions to the program at appropriate positions in the source code to proceed with the automatic parallelization and execution of the program.**

Keywords: Block level parallelization, Instruction level parallelization, Function level parallelization, Loop level parallelization, Automatic parallelization

I. Introduction

Today a large number of techniques have been developed to convert sequential code into parallelized code. Some of them are Block level parallelization [2] in which blocks [3] are analyzed, or Instruction level parallelization in which instructions in the code are analyzed. The current paper has dealt with a new technique of combining block level parallelization and function level parallelization to build up on the existing techniques in automatic parallelization [4] of sequential code.

II. Related Work

Department of Mathematics and Computer Science, South Dakota School of Mines and Technology has worked on a project that focuses on parallelization of C code. Their work mainly concentrates on block level parallelization and does not take care of function level parallelization. Another approach called Cilk devleoped at MIT Laboratory for Computer Science in which programmer has to specify the procedures in the C code that have to be parallelized by inserting keywords in the source code. The Cilk runtime system takes the responsibility for scheduling them efficiently.

G.S.Tjaden and M.J.Flynn proposed the approach for detection and execution of parallel instructions [5]. A study made by David W. Wall from Digital Equipment Corporation, Western Research Laboratory showed the various limits of instruction-level parallelization [6].

The three basic techniques of parallelization, Instruction level parallelization, Block level parallelization and Function level parallelization suffer from these drawbacks:

- In Instruction level parallelization, there may be very high communication overheads or the overhead of creating threads.
- In Function level parallelization, the performance may not be satisfactory. For example suppose that two function calls are made inside another function and that both of them can be parallelized. Also suppose that one of the functions has one or more loops which are responsible for total complexity of the program. Therefore parallelization of the functions would not significantly speed up the code.
- The performance of block level parallelization is better than the previous two. However, if there are more than one function calls in a block and if these functions are responsible for the total complexity of the code then running such blocks in parallel may not yield good results.

We are interested in patching up some of the loop holes in the above techniques by combining both block level and function level parallelization. This approach was found to be better as the problems in function level parallelization and block level parallelization are nearly solved by considering function call as a part of block. Care is taken to ensure that there is only one function call in every block. Function level and block level analysis is done on these blocks and they are parallelized.

III. System Architecture

The system is built as a Pipe-and-Filter architecture. Architecture consists of eight phases as shown in the Fig 1

A. Preprocessing

This stage eliminates all the comments in the input source code. This helps in efficient and uninterrupted processing of

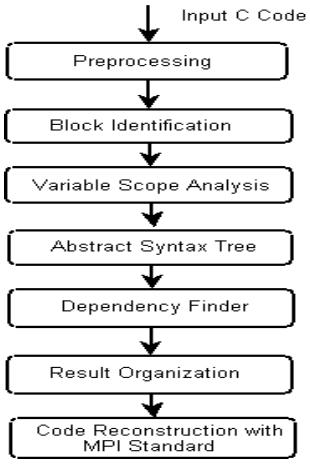

Fig. 1. System Architecture

the source code in the later phases.

B. Identification of blocks

This phase identifies blocks in the source code. We identify every block by the position of its beginning and end markers in the source code. Block identification is carried out for each function in the source code. A block in turn may contain one or more blocks. The block identification procedure is effective and perfect and avoids block conflicts. A block conflict is said to occur if blocks overlap. Identification of blocks is based on keywords in the source code which are responsible for the control flow in the code [3]. These keywords are recognized based on regular expression matching. We call these regular expressions as patterns. Patterns are the regular expressions that are used for identifying a particular order of occurrence of tokens in the source code.
Example :

```
[ \t]*while[a-z0-9A-Z=<>+/*++--_.![]()&&
  \t]+[ \t]*
```

is a pattern that recognizes a *while* statement.

1) Sequence of steps:

- Define patterns for recognizing the keywords occurring in a line in the source code.
- Look for matching patterns in the source code and for every occurrence of keywords, decide the boundaries of the block containing that keyword. There on, note down start and end line numbers of the particular block. This point is made clear in below Fig.2 and Fig.3

```
 1. void main() {
 2.  for(j=0;j<i;j++)
 3.    {
 4.      int j=1, t=1, l=1, w=0, r;
 5.      for(y=0;y<2;y++)
 6.        {
 7.          t++;
 8.        }
 9.      q=1;
10.    }
11.  for(k=0;k<3;k++)
12.  {
13.    int l=-1; w=0;
14.    w++;
15.  }
16.  for(m=0;m<3;m++)
17.  {
18.    int r=1;
19.  }
20.  while(o<1)
21.  {
22.   p=1;
23.  }
24.  getInch();
25.  getSource();
26.  }
```

Fig. 2. Example code

$block_0$	$1-26$
$block_1$	$2-10$
$block_2$	$4-4$
$block_3$	$5-8$
$block_4$	$7-7$
$block_5$	$9-9$
$block_6$	$11-15$
$block_7$	$13-14$
$block_8$	$16-19$
$block_9$	$18-18$
$block_{10}$	$20-23$
$block_{11}$	$22-22$
$block_{12}$	$24-24$
$block_{13}$	$25-25$

Fig. 3. Block sequencing Example in Figure 2

A block is simply a set of statements having single entry and single exit points. This set of statements may consists of compound statements, user-defined functions or a single expression. In this system, set of statements or a statement is identified as a block if it contains any of the following tokens:

- $while$
- for
- if
- $else$
- $\{$
- $\}$
- $userDefFunc()$

Since keywords are identified by regular expressions, we can add more keywords by increasing the number of regular expressions.

Basically, the core complexity of a program lies in a block of compound statements like a *for* loop. Therefore, parallelization of the source code must be at the block level. So block level parallelization can be accomplished in this phase.

C. Variable Scope Analysis

In this phase, each block is parsed to extract the variables in that block. A variable can be an identifier and also an user-defined function call. There are 4 types of flags alloted to each variable. Those are LIVE, DEAD, READ and WRITE. While extracting variables, each statement in the block is analyzed for assigning appropriate values to the four flags corresponding to a variable. A flag is a status bit appended with each variable occurred in a block. The flags are explained below.

LIVE: This flag value tells whether a variable is defined inside a block or not. If the value of flag is '1', then the variable is defined in this block. If '-1' is assigned to this flag, then the variable is defined outside the block.
DEAD: This flag suggests whether a variable will be killed after a block exit or not. If the value of flag is '1', then the variable is dead after this block, otherwise the variable is live even after the block.
READ: This flag suggests whether a variable is read in a particular block or not. If the value of flag is '1', then the variable is read in a block, otherwise the variable is not read in a block.
WRITE: This flag suggests whether a variable has been modified in a particular block or not. If the value of flag is '1', then the variable is modified in a block, otherwise the variable is passed unchanged in a block.

These variables are stored in a list, that is annexed to a block object.

This phase interacts with the block identification module to know start and end line numbers of a block. The output of this phase would be annotated block information for the further block level analysis in later phases. Block level variable analysis makes the scope finding procedure quick and efficient.

1) Sequence of steps:

- Each line in the block is tokenized and it is verified whether each token is an identifier or not. If it is an identifier, it is added to the list.
- If an expression is encountered, the variables are checked for their status i.e. READ or WRITE. On identifying the status of the variable its flags are updated, as shown in the Fig.4.
- An array e.g., a[] is a sequence of variables. Each element in an array is to be considered as a distinct variable and it is accessed using index or iterator of an array. If a[i] is a variable, then it may hold different numeric values at a given time. Therefore, the array variable refers to a distinct array element i.e. a distinct variable. Thus, it is necessary to track the value of index variable. This can be done using *watch*[1] procedure. There on, the value of the index variable is fetched and the corresponding array element is accessed, then its scope information can be found and stored. suppose a[i] in 1st block and a[j] in second block. For a[i] in 1st block and a[j] in 2nd block to be independent of each other, intersection of set of values that j takes for which a[j] was accessed and set of values i took for which a[i] was written must be a null set.
- These steps are repeated until the block end has occurred.

```
for(i=0;i<6;i++)
{
    int a;
    a = a*i;
}
```

Fig. 4. Example code snippet

VAR	L	D	R	W
i	-1	0	1	0
a	1	1	1	1
j	-1	0	1	1

Fig. 5. Variable scope information for above example code

This phase is independent of a block structure, it just depends on the start and end line numbers. This provides flexibility in finding the scope information for any portion of code.

The necessary and sufficient information required for the dependency analysis phase is provided by this phase.

D. Construction of Abstract Syntax Tree

Abstract syntax tree represents each function in the source code as a tree. The nodes are blocks and function definition being the root of the tree. This is logical representation of a function. The tree is built in such a manner that the code can be easily recontsructed in the last phase using this information. A node is a data structure which contains block information such as start and end line numbers, variable scope information, address of parent block and number of children it has. A parent block can be understood as a nesting block.

This phase takes list of blocks from block identification phase and passes logical representation of each function to the next phase.

Since the number of blocks is already known, the process of tree construction can be carried out in $O(n^2)$. Each node stores number of children to avoid back tracking for tree traversal. Algorithm for building the tree is shown as follows The example shown in Fig. 7 shows the creation of tree representation of a given code snippet in Fig. 6.

This logical representation of the function would be useful during the reconstruction or restructuring of the code.

E. Dependency Finder

The information provided by the Abstract Syntax Tree phase is a hierarchy of blocks, tree structure. A tree has levels and the considerable information from this tree would be blocks at the same level, as a input to this phase. At a time two sibling blocks are considered and dependency is found between them. A statement X depends on a statement Y if

[1]watch is a runtime procedure to track value of variables

Algorithm 1 Building Tree representation of code

NSF ⇐ No. of Scannable Functions
numchild ⇐ 0
index ⇐ 0
parent ⇐ -1
level ⇐ 0
block ⇐ blocksIdentifier(function.startLineNo,function.endLineNo)
NB ⇐ number of blocks in sequence
child$_0$ ⇐ new Node(block$_0$.$startLineNo$, block$_0$.$endLineNo$, numchild, index, parent, level)
root ⇐ child$_0$
index ⇐ 0
parent ⇐ 0
level ⇐ level+1
child$_1$ ⇐ new Node(block$_1$.$startLineNo$, block$_1$.$endLineNo$, numchild, index,parent,level)
child$_0$.$numchild$ ⇐ 1
for $j = 2$ to NB **do**
  for $k = j - 1$ to 0 **do**
    condition1 ⇐ block$_j$.$startLineNo$ > block$_k$.$startLineNo$
    condition2 ⇐ block$_j$.$endLineNo$ < block$_k$.$endLineNo$
    if condition1 and condition2 **then**
      numchild ⇐ block$_j$.$numchild$
      index ⇐ j
      parent ⇐ k
      level ⇐ level+1
      block$_j$ ⇐ new Node(block$_j$.$startLineNo$, block$_j$.$endLineNo$, numchild, index, parent, level)
      break
    end if
  end for
end for

- X can be executed after Y in a run of the program
- both X and Y read or write a common memory location

```
1. func()
2. {
3.   if(i>2)
4.   {
5.    i=i+4;
6.   }
7.   while(i>0&&i<2)
8.   {
9.    j++;
10.  i--;
11.  }
12. }
```

Fig. 6. Example code snippet

```
1-12
|
|->3-6
|    |
|    |->5-5
|
|->7-11
|    |
|    |->9-10
```

Fig. 7. Tree representation for above example code in Fig. 6.

Based on variable scope information of both the blocks taken at a time, dependency is found. Result is stored for each pair of blocks. Dependency finding stage is carried out in two intermediate stages.They are

1) Function Dependency Finder
2) Block Dependency Finder

Let us understand each of them individually

- Function Dependency Finder: In this sub-phase both of the blocks are checked whether they contain any user defined function call or not. If any one of them contains a function call, then that particular function's arguments and return variables are matched, with the variables in the second block and vice versa. To accomplish this verification procedure a table is maintained, that tracks the variables status i.e., READ and WRITE in each expression in the other block (block other than that which contains a function call). This table will suggest whether return value of the function call is used by the other block and also whether any variable in the other block are passed as arguments in the function call.
  After the above analysis, global variable analysis is done. The precondition for global variable analysis is both the blocks must contain different user-defined function calls. During this analysis, each function definition is scanned for any common access of global variables. If it is found that they access or modify any of the global variables in the program, then blocks containing these function calls are considered to be dependent.
  To understand Function Dependency Analysis, consider a function application $f(e_1, ..., e_n)$. Since the language

is function, the arguments $e_1, ..., e_n$ do not depend on each other. Thus, dependence analysis is trivial in a function language. We know that these expressions are independent. We also know that the function call "$f(...)$" does depend on the arguments to the call.

The standard way of representing functional programs for parallel execution is as *dataflow graphs*. The graphs show dependences between expressions in the form of what functions use what values. Each node of the graph denotes a function; edges connect the results of a function to where they are used. For example, the functional expression $(z + 2, x * y)$ would have the graph as shown in Fig.8

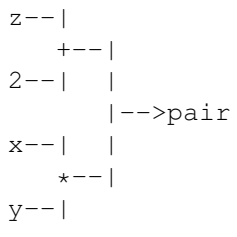

Fig. 8. Data flow graph for above functional expression

- Block Dependency Finder: If dependency status of the pair of blocks taken is pending, then those blocks are passed on to this phase. In this phase common variables occurring in each block are matched. The status flags are compared.

This phase interacts with abstract syntax tree phase to take pair of blocks as inputs. Output of this phase would be dependency status of a pair of blocks. Algorithm used is shown in Algorithm2

To give brief idea of functionality of this phase, in Fig.4 the blocks from 3 to 6 and 7 to 11 are dependent because they access and modify a common variable whose scope is visible to both the blocks. If flag sequence of a common variable in both the blocks is found in one of the following sequences, then the blocks are dependent on each other.

LIVE=1, DEAD=0, READ=1 and WRITE=0
LIVE=-1, DEAD=0, READ=1 and WRITE=0
LIVE=1, DEAD=0, READ=0 and WRITE=0
LIVE=-1, DEAD=0, READ=0 and WRITE=0
LIVE=-1, DEAD=1, READ=0 and WRITE=0
LIVE=1, DEAD=1, READ=0 and WRITE=0

F. Dependency Analysis

Dependency analysis phase takes input, pair of blocks from the list of blocks provided by the Abstract syntax tree phase and tests the dependency between the two blocks.

This phase accomplishes the task of finding maximum number of blocks that can be run in parallel, establishing transitive relations between the blocks. Two blocks are transitively related, if a block A is independent on block B and block B is independent on block C, then dependency between block A and block C is found. On finding out, this relationship between all the blocks in the source code, all the transitively related and directly related blocks are combined together, which can be considered as partition of a set of blocks. A partition of a set X is a set of nonempty subsets of X such that every element x in X is in exactly one of these subsets.The idea

Algorithm 2 dependencyFind():Finding dependency between block$_i$ and block$_j$ (i<j)

```
1:  flag ⇐ 0
2:  if block_i and block_j contain function calls and both are dependent then
      block_i and block_j are dependent
3:  else
4:    nov_i ⇐no of variables in block_i
5:    nov_j ⇐no of variables in block_j
6:    for i1 = 0 to nov_i do
7:      for i2 = 0 to nov_j do
8:        if block_j.variables_i1==block_i.variable_i2 then
9:          if block_i.flag_i2-0==1 and block_i.flag_i2-1==0 and block_i.flag_i2-2==1 and block_i.flag_i2-3==0 then
10:           flag ⇐ 0
11:         end if
12:         if block_i.flag_i2-0==-1 and block_i.flag_i2-1==0 and block_i.flag_i2-2==1 and block_i.flag_i2-3==0 then
13:           flag ⇐ 0
14:         end if
15:         if block_i.flag_i2-0==1 and block_i.flag_i2-1==0 and block_i.flag_i2-2==0 and block_i.flag_i2-3==0 then
16:           flag ⇐ 0
17:         end if
18:         if block_i.flag_i2-0==-1 and block_i.flag_i2-1==0 and block_i.flag_i2-2==0 and block_i.flag_i2-3==0 then
19:           flag ⇐ 0
20:         end if
21:         if block_i.flag_i2-0==-1 and block_i.flag_i2-1==1 and block_i.flag_i2-2==0 and block_i.flag_i2-3==0 then
22:           flag ⇐ 0
23:         end if
24:         if block_i.flag_i2-0==1 and block_i.flag_i2-1==1 and block_i.flag_i2-2==0 and block_i.flag_i2-3==0 then
25:           flag ⇐ 0
26:         else
27:           flag ⇐ 1
28:         end if
29:       else
30:         if flag==1 then
              break
31:         end if
            flag ⇐ 0
32:       end if
33:     end if
34:   end for
35: end for
36: if all variables of block_j satisfy above rules then
37:   print 'block_i and block_j are independent'
38: end if
```

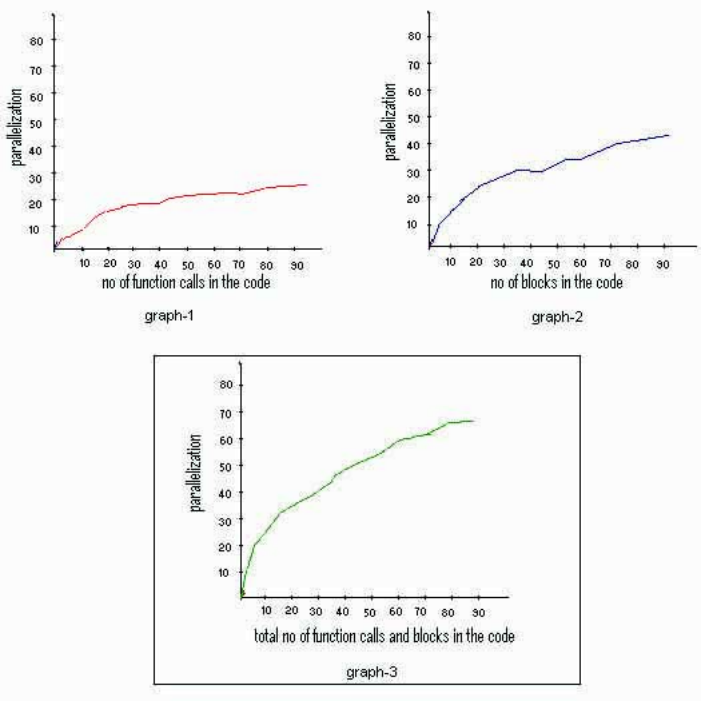

Fig. 9. Degree of parallelization for three above mentioned techniques

behind a partition of a set is, all the blocks in that partition would run in parallel independently. On the other hand, each partition runs sequentially.Important thing to note is to take care of Block control mechanism For example, consider three partitions of a set $\{A, B, C, D, E, F, G, H\}$ are $P1 = \{A, B\}$, $P2 = \{E, F, C\}$, $P3 = \{D, G, H\}$. Now P2 runs after the execution of P1, similarly P3 executes after the execution of P2. Whereas, in P1 both A and B execute concurrently, in P2 E, F and C execute concurrently and in P3 D, G and H execute concurrently.

Assume that total program takes 1 sec. When all the blocks are run sequentially, then execution time of P1, P2 and P3 would be 30%, 35% and 35% respectively. When the blocks in each of the partitions are executed concurrently, then the execution time of P1, P2 and P3 would be 15%, 11.67% and 11.67% respectively. This shows that the total execution takes only 38.4% of the original time.

G. Result Organization and Code Reconstruction with MPI standard

The MPI library calls are appended at those portions of the source code, where the program can be parallelized. These portions are identified from the output file.

IV. Result

We tried and tested this Algorithm for parallelization of sequential code using block level and function level parallelization by giving input programs with increasing number of blocks and function calls. Its result is compared to the one tested with block level parallelization algorithm alone and to the one tested with function level parallelization algorithm alone. It has been observed that the earlier is much more efficient when compared to the other two. The figure 9 show the results Graph-1 shows the degree of parallelization when function level parallelization is carried out. Graph-2 shows the degree of parallelization when block level parallelization is carried out and result is better than that of function level parallelization. Graph-3 shows the degree of parallelization when block level parallelization with function level parallelization is carried out and degree of parallelization is observed to be nearly double. Hence the approach was certainly found to be more effective.

V. Future Perspective

Future scope would be to implement loop level parallelizationt [11] and instruction level parallelization [9], to perform better. The loop level method parallelizes low-level loops in the code that are responsible for most of the computational cost. There are no restrictions when running with this method. However, the speedup factor may be significantly less than what can be achieved with domain level parallelization. The speedup factor will vary depending on the features included in the analysis since not all features utilize parallel loops. The idea given in the paper [12] of dealing with pointers and dynamic variables in Automatic parallelization will be extended to C. The idea will be extended to handle pointers in the C code. Power of instruction level parallelization will also be used to enhance the usability of the algorithm.

References

[1] Alian Darte, Yves Robert, Frederic Vivien *Loop Level Parallelization Algorithms* LIP, Ecole Normale Superieure de Lyon, F 69394 LYON Cedex 07, France

[2] Gregg Stubbendieck, Pete Gasper, Caleb Herbst, Jeff McGough, Chris Rickett *Automatic Parallelization of Sequencial C Code* Department of Mathematics and Computer Science, South Dakota School of Mines and Technology peter.gasper@gold.sdsmt.edu

[3] Muchnick,S. S. (1997). *Advanced Compiler Design & Implementation*. San Francisco, California: Morgan Kaufmann.

[4] Armstrong, Brian and Eigenmann, Rudolf *Challenges in the Automatic Parallelization of Large-scale Computational Applications* 20011121;

[5] G. S. Tjaden and M. J. Flynn. Detection and parallel execution of parallel instructions. IEEE Transactions on Computers C-19 (10), pp. 889-895, October 1970.

[6] Limits of Instruction-Level Parallelism David W. Wall Proceedings of the fourth international conference on Architectural support for programming languages and operating systems, pages 176–188. ACM Press, 1991.

[7] Aho, A. V., Sethi, R. Ullman, J. D. (1986). Compilers: Principles, Techniques, and Tools. Murray Hill, New Jersey: Bell Telephone Laboratories Inc.

[8] Research & High-Performance Computing Support Portable Batch System at McMaster http://www.rhpcs.mcmaster.ca/pbs/abaqusp.html

[9] Faraboschi, P. and Fisher, J.A. and Young, C., *Instruction scheduling for instruction level parallel processors*. Proceedings of the IEEE, vol 89, 20010000.

[10] Gu, Junjie and Li, Zhiyuan *Efficient interprocedural array data-flow analysis for automatic*. Software Engineering, IEEE Transactions on volume 26 20000330.

[11] Ricci, L. *Automatic loop parallelization: an abstract interpretation approach* Parallel Computing in Electrical Engineering, 2002. PARELEC '02. Proceedings.

[12] Manuel Hermenegildo School of Computer Science, Technical University of Madrid(UPM), Spain*Automatic Parallelization of Irregular and Pointer–Based Computations: Perspectives from Logic and Constraint Programming* Invited Paper at Europar'97, August 1997, LNCS, Springer–Verilag