Framework for Automatic Parallelization

Anala M R, Deepika Dash
R V College of Engineering
Department of CSE

Abstract— **Continuous research to increase the performance gain of the device in the field of computing has led to the advent of multi-core processors, which include more than one independent processing unit (core). This revolution on the hardware-side of the computer demands the software programmers to exploit the processing power by developing parallel programs. Programmers prefer writing serial applications since it is easier to put their ideas into the form of serial code than a parallel one. Hence, parallelizing such serial applications without much burden on the developer becomes paramount.**

This paper proposes a system that automatically parallelizes serial C code. The system proposed performs detailed dependency analysis, identifies the 'for' blocks that satisfy the criteria of being potential to be parallelized using the OpenMP framework, identifies the regions in such potential 'for' blocks which need to be run in a critical section. The correctness of the input code is ensured by this system. Also, post execution analysis i.e. execution of the input program to monitor the usage of resources is avoided which makes this system faster than some of the existing systems.

Keywords- OpenMP;for loop; parallel

I. INTRODUCTION

A single processor was used in legacy systems where instructions were executed sequentially without any overlapping. To improve the performance of such systems, the technique of partially overlapping instructions called pipelining was used [1]. In order to further improve their performance, clock speeds of each processor had to be increased, which was costlier than designing a system with multiple processors.

Hence, multicore architecture, which involves more than one processor on a single silicon die, was invented. This enhanced the performance of the computer without increasing the running frequency of each processor. It includes additional benefits like reduction in power consumption [2] and reduction in size and weight. According to Flynn's Model [3], computer architecture can be classified into 2 dimensions. First is the number of data streams which can be processed concurrently. Second is the number of instruction streams the computing device supports.

The proposed algorithms are designed to be applied on Single Instruction Multiple Data Stream class of computers. In order to achieve parallelism in computers, using Parallel Computing Platforms is one of the available methods. Parallel Computing Platforms enable abstraction of the underlying architecture from the developer. However, parallel computing platforms have certain limitations because of the potential software bugs [4] accompanying them like synchronization, communication and race conditions. Examples of parallel computing platforms available are OpenMP, CUDA, MPI etc.

Automatic Parallelization enables abstraction of parallel computing platforms from the developer thereby reducing the efforts of the developer in order to parallelize serial programs. It also enables migration of serial programs to multicore systems. It can also be used to parallelize existing serial programs. The algorithms proposed in the paper performs modification of given input program in order to parallelize "for" loops present in the program. It enhances the performance of the program while maintaining the correctness of the program when executed in parallel.

II. BACKGROUND

In the algorithms proposed, OpenMP [5] is used to parallelize for loops in C program. The "for" loops can be parallelized using OpenMP by making use of for directive. This directive can include data scheduling clauses, synchronization clauses, sharing attribute clauses, initialization clauses, reduction clauses, etc. These clauses include one or more variables denoting the property of the variable(s) when the loop is parallelized. Some of the clauses used in the paper are explained briefly below.

A. Data Scope Attribute Clauses:

–**Private:** Data within parallel region is private to each thread.
–**Lastprivate:** Data is private to each thread, but the data will be copied to global variable from the last iteration of the loop.
–**Shared:** The data within parallel region is shared between all threads.
–**Reduction:** Variable has local copy in each thread. At the end of all iterations, local copy of each thread is combined using the operation specified.

III. RELATED WORK

This paper is in continuation of our previous paper [6]. The importance of high-performance computing and the comparative study of different models of computation is discussed in [7]. There are many other research areas [8, 9] where the serial algorithms are to be parallelized automatically to reduce the burden of programmer. The following tools have been developed in order to perform automatic parallelization of sequential code.

A. Par4All

Par4All [10] is designed to parallelize programs in C and Fortran. This tool enables parallelization on heterogeneous systems comprising of HPCs and GPUs.

978-1-7281-0114-9/18/$31.00 ©2018 IEEE
DOI 10.1109/HiPCW.2018.00026

B. ParallWare

ParallWare is a commercial tool built for automatic parallelization. It works as a source-to-source compiler [11]. Automatic parallelization is achieved by inserting OpenMP clauses to the serial C code. Dependency elimination by the technique of variable renaming is not inculcated.

C. Intel C++ Compiler

Intel C++ compiler [12] comes with an inbuilt automatic parallelization tool which can parallelize the code to be compiled. It is accompanied by an internal performance evaluation tool which is used to determine profit gain by parallelization and use this data to parallelize the given code.

D. Cetus

Cetus [13] is an open source tool designed to automatically parallelize the input serial C code using OpenMP APIs. Model-based and profile-based techniques are used to determine profitability of parallelizing the loops. This tool limits itself to the already declared variables, thereby not removing the dependencies through the variable renaming technique. Recursive function calls are not parallelized using OpenMP 'task' construct.

The algorithm proposed recognizes multiple initializations and makes minor changes to the program in order to make the loop compatible with OpenMP APIs. The compile time cost model used in this algorithm does not require post execution analysis. The concept of register renaming is applied to variables in order to eliminate dependencies and maximize parallelism.

IV. METHODOLOGY

Code analyzers are used in most of the existing automatic parallelization tools to identify parts of the given code that can be parallelized. The proposed algorithms enable analysis of "for" loops in C code. It uses OpenMP "for" directive to parallelize them. Data structures used in the proposed algorithms are -Variable Table and Loop Table.

Variable Table: Name of the variable, Number of array dimensions, list of *private_loop_ids* (list of *loop_ids* in which indicates that the variable is marked private in loops and operations performed on that variable a□ects it).

Loop Table (stores the details of outermost for loops): *loop_id*, *loop_var*, Start constant, End constant, Comparison type, Increment type (indicates the type of operation in increment section of loop header), Increment value, *Valid* flag (is set if the loop satisfies the OpenMP format of "for" loop header), *private_var_list*, *shared_var_list*, *lastprivate_var_list*, *reduction_var_list, parent_loop_id* (in case of nested loops, *loop_id* of its parent).

Statement Table: *stmt_id* (Unique ID given to each statement in the C program), variables access list (contains a list of variables accessed in the statement along with the type of access read/write) and *cond_nesting_level* (nesting level of this statement in a conditional block i.e. "if" and "switch").

The Algorithm 1 shows the major steps involved in analyzing the for loops in the input serial C code.

Algorithm 1 Algorithm for "for" loop Analysis

Input: Serial C code
Output: C Code ready for parallel execution

1: Analysis and Modification of "for" loop header
2: Dependency Graph Generation
3: Critical Section Identification
4: Identification of parallelizable loops
5: Data Scope Analysis of variables
6: Generate OpenMP Clause to parallelize for loop.

A. "for" Loop Header Analysis

OpenMP restricts itself to a certain format of "for" loop headers in order to parallelize it. OpenMP Version 2.5 has 5 restrictions on the "for" loop header. These restrictions are described below.

- Data type of loop variable *(loop_var)* must be signed integer. This variable must be loop invariant.
- Comparison expression follows the format as shown in (1).

loop_invariant_integer indicates a variable or expression involving variables which are not modified in the loop.

loop var <, ≤, >or ≥ loop invariant integer (1)

- Increment part of the loop header must increment or decrement the loop var by a loop invariant value.
- Loop variable (*loop_var*) must be incremented if the comparison operator is <or ≤, similarly loop variable (*loop_var* must be decremented if the comparison operator is >or ≥.

"for" loop headers that satisfy all the above conditions proceed to the next step of the Algorithm 1. The details of this step are shown in the Algorithm 2. Additional data about the loop is determined such as start constant and end constant of the iteration variable. This data is gathered by parsing 3 parts of "for" loop header: initialization section (which determines loop var and start constant), comparison section (which determines end constant and comparison type) and increment section (which determined increment type). If any of the sections is not present, then the loop is marked as non-parallelizable (Valid flag is unset). In case of multiple initializations in the loop header, all initializations are moved above the loop header and initialization of the loop var is generated using variable renaming. Similarly, if the loop var is not initialized within the header, initialization statement is added in order to meet the restriction of loop header as specified by OpenMP. All the data gathered in this step is stored in Loop Table.

Algorithm 2 Analysis and modification of "for" loop header

Input: "for" loop header
Output: Modified "for" loop header

1: Parse the condition section in the "for" loop header and determine *loop_var*, *end_constant* and *comparision_type*.
2: Parse increment section of the "for" loop header and determine *type_of_operation* (increment or decrement) and *loop_invariant_value*.
3: Parse initialization section of the "for" loop and check for initialization of *loop_var* and initialization value.
4: **if** multiple initialization sections present **then**
5: Shift all initialization before the "for" loop header and rename *loop_var*.
6: **end if**

B. Dependency Graph Generation

The custom parser parses each statement of the input serial C code and identifies the type of access on every variable in it, if any, and stores the same in the *variable_access_list* along with the *statement_id* in the Statement Table. Once all the statements are parsed, the Algorithm 3 is applied in order to generate the dependency graph.

This algorithm scans the *variable_access_list* of each statement entry in the Statement table entry by entry. Once a variable's *curr_access_type* is obtained by calling the procedure getAccessType(). The function getAccessType() returns the access on the variable passed to it in the statement with *stmt_id* passed to it. The *curr_stmt* is obtained by invoking the procedure getPrevStmt(). The *curr_stmt* is used as a temporary variable to maintain the statement id from which the destination access needs to be checked.

The procedure getPreviousAccess() returns the *stmt_id* of the statement on which the statement passed is dependent on due to the access on the variable passed to it. This *stmt_id* is stored as the *dest_stmt*. While finding this previous statement, the procedure iterates through the program in decreasing order of the conditional nesting level i.e. once the conditional nesting level is decreased, the statements having *cond_nesting_level* more than it are skipped. This way, the dependency between the statements in conditional blocks and outside those blocks are e□ciently maintained.

A dependency edge is added from the *curr_stmt* to the *dest_stmt*. The edge added contains the type of dependency between those statements and the variable causing this dependency. The dependency type is obtained by the procedure getDependencyType(). The *src_access_type* and *dest_access_type* are passed to this procedure. If *src_access_type* is READ then an edge is added only when the *dest_access_type* is WRITE, the dependency type returned is Read After Write (RAW). Similarly, when *src_access_type* is WRITE and the *dest_access_type* is READ, then the dependency type returned is Write After Read (WAR). When src access type is WRITE and the dest access type is WRITE, then the dependency type returned is Write After Write (WAW). The *curr_stmt* is updated to the *dest_stmt* so that edges are added to all the statements on which the source statements depend. This process is continued until all the statements within the current function are checked using the procedure getPreviousAccess().

Algorithm 3 Dependency Graph Generation

1: **for all** Statements (s) in input program **do**
2: **for all** Variables (v) accessed in statement s **do**
3: *curr_access_type* ← getAccessType(v,s)
4: *curr_stmt* ← getPrevStmt(s)
5: **while** (*dest_stmt* ← getPreviousAccess(v,*curr_access_type*,*curr_stmt*)) exists **do**
6: Add edge from statement s to statement *dest_stmt* with dependency type ← getDependencyType(*curr_access*,getAccessType(v,*dest_stmt*))
7: *curr_stmt* ← *dest_stmt*
8: **end while**
9: **end for**
10: **end for**

C. Critical Section Identification

The Algorithm 4 marks statements as critical with respect to "for" blocks. This step considers one "for" block in Potential Block Table at a time and iterates through every statement within the "for" loop body. If there is a write on a shared variable in a statement, then that statement and every statement that is directly or indirectly dependent on it inside the current "for" loop body are added to the critical section with respect to the "for" block under consideration.

Algorithm 4 Critical Section Identification

1: **for all** "for" blocks in Potential Block Table **do**
2: **for all** Statements (s) in the "for" loop body **do**
3: **if** Statement s contains WRITE on a shared variable in the "for" block **then**
4: Add statement s to critical section with respect to that "for" block
5: Add all statements within that "for" block that are dependent on the statement s directly or indirectly to critical section with respect to that "for" block
6: **end if**
7: **end for**
8: **end for**

D. Identification of parallelizable loops

After dependency graph is constructed using the method described above, analysis is made on each of the potential blocks identified in order to determine whether the loop under consideration is parallelizable or not. WAW and WAR dependencies are false dependencies. They can be eliminated using Register Renaming [14] concept at application level. Hence dependencies are reduced. Further tests are made to eliminate dependencies due to array variables. The tests include Omega test [15], Gaussian elimination [16], CRI test [17], The I test [18], Banerjee test [19] etc. After all possible eliminations of dependencies, a check is made to determine whether there exists any loop within the dependency graph consisting of statements of the "for" loop body. If such a loop exists and at least one of them is not marked critical, then the algorithm concludes that the loop cannot be parallelized, hence the loop is marked non-parallelizable by unsetting the valid flag.

E. Data Scope Analysis

As indicated in the Algorithm 5, the system considers each variable with the type of access (read / write) at a time. If the variable is being read, the context of the statement under consideration is identified (whether the statement is inside a potential block or not). If the variable is inside a potential block then current loop id is retrieved. reduction var list, shared var list and private var list are checked for the current variable. If it is already present then no further analysis is required, else the variable is added to the shared var list of the current "for" block entry (indicated by loop id). loop id is updated to its parent loop id. This process is repeated until the current "for" block does not have any parent. If the statement is not inside a potential block then for each loop id present in private loop ids, remove the variable from the private var list, add the variable to lastprivate var list and remove the loop id from variables private loop ids list.

If access on the variable is write, context of the statement is identified (whether the statement is inside a potential block or not). If the statement is inside a potential block then current loop id is retrieved. The reduction var list, shared var list and private var list are checked for the current variable. If it is already present in any one of them then no further analysis is required, else the variable is added to the private var list of the current "for" block entry (indicated by loop id). This operation is followed by addition of loop id to the private loop ids of the current variable under consideration. loop id is updated to its parent loop id. This process is repeated until the current "for" block does not have any parent. If the statement is not inside a potential block then all loop ids present in private loop ids of the current variable are deleted from that list.

If a variable processed using the method described above is added to shared var list, another check is made for adding the variable to reduction var list. This check is invoked using the procedure "CheckForReductionCapability()". This procedure checks for the variable access present within the "for" block under consideration. If there is only one statement in which the variable is present along with operation that can be used under "reduction" clause (for eg: +=,-=, etc.) then the variable is removed from shared var list and added to reduction var list along with the operation.

F. Generation of OpenMP Clauses

The final step of "for" loop parallelization is insertion of "#pragma omp for" directive. The addition of the directive is done only if the Valid flag of the "for" loop under consideration is set. The directive along with the appropriate clause(s) is added at the proper places to achieve parallelism using OpenMP parallel computing platform. This step begins scanning the Loop Table and checking for all the parallelizable loops (whose Valid flag is set). The "#pragma omp parallel for" clause is added before the "for" loop header. The data scope attribute clauses are added accordingly.

A configurable option is provided to the user to enable parallelizing nested loops, called Enable Nested. If this option is set, then all the parallelizable loops in the potential block table are parallelized irrespective of the state of their enclosing "for" loops. If the option is unset, then all enclosing "for" loops of the current "for" loop under consideration are checked if their Valid flag is set i.e. the "for" loop is parallelized. If so, then the current loop is not parallelized. This way, only the parallelizable loop at the lowest possible nesting level is parallelized, maintaining the overhead that might be created due to thread overhead.

Algorithm 5 Data Scope Analysis

```
1: Initialise curr_for_id ← getCurrentLoopID()
2: if variable is READ then
3:    if statement is inside a "for" loop then
4:       while hasParent(curr_for_id) is true do
5:          if variable not in private_var_list of the Loop then
6:             if curr_for_id not in variable.private_loop_ids then
7:                Remove variable from private_var_list of loop entry with loop_id = Lid
                  ∀ Lid ∈ variable.private_loop_ids
8:                Remove curr_for_id from variable.private_loop_ids
9:             end if
10:            Add variable to shared_var_list of loop entry of curr_loop_id
11:         else if curr_for_id not in variable.private_loop_ids then
12:            Remove variable from private_var_list and add to lastprivate_var_list of loop
               entry with loop_id = Lid ∀ Lid ∈ variable.private_loop_ids
13:         end if
14:         curr_for_id ← getParentLoopID(curr_for_id)
15:      end while
16:   end if
17: end if
18: Initialise curr_for_id ← getCurrentLoopID()
19: if variable is WRITE then
20:   if statement is inside a "for" loop then
21:      while hasParent(curr_for_id) is true do
22:         if variable not in private_var_list and shared_var_list and private_var_list of
            the Loop then
23:            if variable.array dimensions is 0 then
24:               Add variable to private_var_list of loop entry of curr_loop_id
25:               Add curr_loop_id to variable.private_loop_ids
26:            else
27:               Add variable to shared_var_list of loop entry of curr_loop_id
28:               CheckForReductionCapability(variable,curr_loop_id)
29:            end if
30:         else if curr_for_id not in variable.private_loop_ids then
31:            Remove curr_for_id from variable.private_loop_ids
32:         end if
33:         curr_for_id ← getParentLoopID(curr_for_id)
34:      end while
35:   end if
36: end if
```

Finally, the generated directive(s) along with the clauses is added as a new statement at its proper position into the output file.

V. Experimental Results

For the loop present in "foo()" function, it is evident from the dependency graph in Figure 1 there is a self loop in statement "b[i] = b[i-1] + i;" and this statement is not under critical sections and hence the loop is not parallelized by the system. In case of the loop present in "main()" function there are loops in dependency graph as shown in Figure 2. But all the statements in the loop are marked as critical which makes the loop parallelizable. Hence the loop inside "main" function is parallelized and required statements are marked critical.

This algorithm is applied on di□erent programs and the parallelized code is executed on Intel Core i7-2670QM CPU @ 2.20GHz, 64-bit Instruction Set, 6 MB Cache, with 4 cores and 8 threads (hyper threading), running Windows OS. Execution time of programs executed in parallel (with 8 threads) and serial (1 thread) are measured for these programs.

Figure 3 shows the execution time in seconds for serial and parallel execution. Serial (1 thread) are measured for these programs. Figure .3 shows the execution time in seconds for serial and parallel execution.

Input Code

```
#include<stdio.h>
#define SIZE 100

int foo(int b[],int n)
{
    int i;
    for(i=0;i<n;i++)
    {
        b[i] = b[i-1] + i;
    }
    return a[i];
}

int main()
{
    int i,j,k;
    int sum ,max;
    int a[SIZE];
    for(j=0,i=j,max=i,sum=i;i<SIZE;i++)
    {
        k = foo(a,j);
        if(max < a[i]){
                    max = a[i];
                }
                sum += a[i];
                k = j;
        j = k + 1;
    }
    printf(''k = %d, Sum = %d",k,sum);
}
```

Parallelized Code

```
#include<omp.h>
#include<stdio.h>
#define SIZE 100

int foo(int b[],int n)
{
        int i;
        for(i=0;i<n;i++)
        {
                b[i] = b[i-1] + i;
        }

        return a[i];
}
int main()
{
        int i,j,k,i_1;
        int sum ,max;
        int a[SIZE];
        j=0;
        i_1=j;
        max=i_1;
        sum=i_1;
        #pragma omp parallel for private ( i )
            lastprivate ( k ) shared ( max, j, a, n, b )
             reduction ( +:sum )
        for(i = i_1;i<SIZE;i++)
        {
                #pragma omp critical
                {
                        k = foo(a,j);
                        if(max < a[i])
                        {
                                max = a[i];
                        }
                }
                sum += a[i];
                #pragma omp critical
                {
                        k = j;
                        j = k + 1;
                }
        }
        printf(''k = %d, Sum = %d",k,sum);
}
```

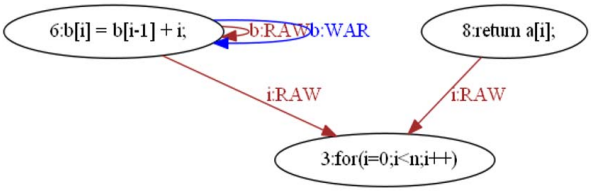

Figure 1. Dependency Graph for "for" loop inside "foo()" function

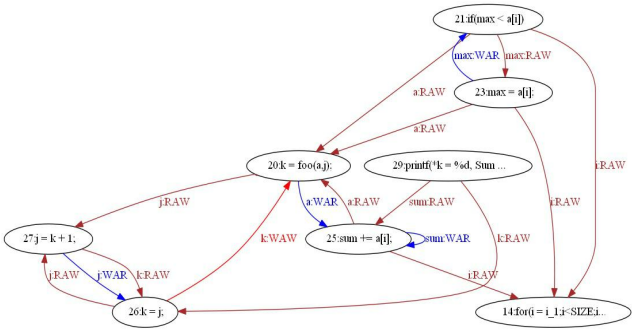

Figure 2. Dependency Graph for "for" loop inside "main ()" function

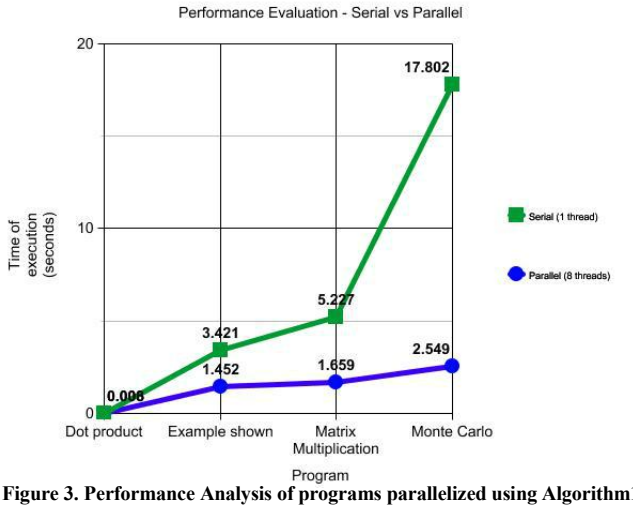

Figure 3. Performance Analysis of programs parallelized using Algorithm1

VI. CONCLUSION

Using OpenMP APIs, the proposed algorithm parallelizes "for" loops in a simple and e□cient way. It creates dependency graphs that gives a clear picture of data and control dependencies that are present in the program which is used to determine whether the loop is parallelizable or not. This ensures the logic of the program is unaltered while parallelizing the loop and hence enhancing the performance of the program. The algorithm also tries to modify the loop header to make it suitable for parallelizing using OpenMP.

VII. Future Work

Sometimes parallelizing "for" loops can degrade the performance of the program due to the overhead that is involved in synchronization between threads. This drawback can be eliminated by performing profitability tests on the "for" loops after marking statements as critical. The concept used in the proposed algorithm can be extended to parallelize "while" and "do while" loops. In case of nested loops, loop interchange can enhance the amount of parallelization achieved. Hence loop interchange can be added to the algorithm.

VIII. Acknowledgement

The authors of this paper thank Ms. Meghana Babu and Mr. Amit Bhat for their valuable contribution during the course of the project.

References

[1]. David Patterson and John L. Hennessy, "Computer Architecture, A Quantitative Approach (Fifth Edition)", Morgan Kaufmann, 2011, ISBN 978-0123838728.

[2]. R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen, "Single-ISA Heterogeneous Multi-core Architectures: The Potential for Processor Power Reduction", Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36), Dec. 2003, ISBN: 0-7695-2043-X.

[3]. D.B. Skillicorn, "A Taxonomy for Computer Architectures", Computer, Vol. 21, No. 1 1 ,1988, INSPEC Accession Number: 3321122. pp. 46-57.

[4]. Hennessy John L, Patterson David A, Larus James R, "Computer organization and design: the hardware/software interface (2. ed., 3rd print. ed.)", San Francisco: Kaufmann, 1999, ISBN 1-55860-428-6.

[5]. Amit G Bhat, Meghana N Babu, Anala M R, "Towards automatic parallelization of 'for' loops", 2015 International Advance Computing Conference(IACC), DOI: 10.1109/IADCC.2015.7154686, pp:136-142.

[6]. Patrick Mukala, "Arithmetic Deduction Model for High Performance Computing: A Comparative Exploration of Computational Models Paradigms", International Journal of Information and Network Security (IJINS), Vol 3, No 3, 2014.

[7]. Hong Li, Bin Gong, He-Bei Gao, Chang-Ji Qian, "Parallel Computing Properties of Tail Copolymer Chain", Indonesian Journal of Electrical Engineering and computer science, e-ISSN: 2087-278X, Vol. 11, No. 8, August 2013, pp: 4344 - 4350.

[8]. Xue-Feng Jiang, "Application of Parallel Annealing Particle Clustering Algorithm in Data Mining", Indonesian Journal of Electrical Engineering and computer science, Vol 12(3), 2014, pp:2118-2126.

[9]. S. Akhter and J. Roberts, "Multi-Core Programming", Intel Press, 2006.

[10]. M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. O. McMahon, F. X. Pasquier, G. Pan, and P. Villalon, "Par4all: From convex array regions to heterogeneous computing", IMPACT'12, Paris, France, Jan. 2012.

[11]. Qingye Jiang ; Young Choon Lee ; Arenaz, M. ; Leslie, L.M., "Optimizing Scientific Workflows in the Cloud: A Montage Example in Utility and Cloud Computing (UCC)", 2014 IEEE/ACM 7th International Conference on 8-11 Dec. 2014, INSPEC Accession Number:14888469, pp: 517-522.

[12]. X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito, and E. Su, "Intel OpenMP C++/Fortran Compiler for Hyper–Threading Technology: Implementation and Performance", Intel Technology Journal, 2002.

[13]. Bluemke, I., Fugas, J., "C code parallelization with paragraph", ICIT, 2010 2nd International Conference on Information Technology (ICIT), ISBN: 978-1-42448182-8, pp. 163-166,

[14]. Sima, D., "The design space of register renaming techniques, in The design space of register renaming techniques", ISSN: 0272-1732, pp: 70-83.

[15]. William Pugh, "A practical algorithm for exact array dependence analysis", Magazine - Communications of the ACM, vol. 35 issue 8. pp. 102-114.

[16]. Lin, H.X., "Designing parallel sparse matrix algorithms beyond data dependence analysis", IEEE International Conference on Parallel Processing Workshops, 2001, page(s): 7-13, ISBN: 0-7695-1260-7.

[17]. Birch, J., Psarris, K., "Discovering Maximum Parallelization Using Advanced Data Dependence Analysis", HPCC 08, 10th IEEE International Conference on High Performance Computing and Communications, pp.103-112, ISBN:978-0-7695-3352-0.

[18]. X. Kong, D. Klappholz, and K. Psarris, "The I Test: An Improved Dependence Test for Automatic Parallelization and Vectorization", IEEE Transactions on Parallel and Distributed Systems, 1991, pp. 342-349 vol.2 issue 3, ISSN: 1045-9219.

[19]. U. Banerjee, "Dependence Analysis for Supercomputing" Boston: Kluwer Academic, 1988, ISBN: 0898382890.1.

Appendix

A. Parallel program

```
#define row_col 1000;
void matrix_mul()
{
int row_size,col_size,i,j,k,index,num;
row_size=row_col;
col_size=row_col;
 /*-----------------------------------------------------------------*/
int** mat = (int**)malloc(row_size * sizeof(int*));
#pragma omp parallel for
  for (index=0;index<row_size;++index)
      {
            mat[index] = (int*)malloc(col_size * sizeof(int));
      }
```

```
      int** mat2 = (int**)malloc(col_size * sizeof(int*));
      #pragma omp parallel for
      for (index=0;index<col_size;++index)
           {
            mat2[index] = (int*)malloc(row_size * sizeof(int));
           }
     #pragma omp parallel for private(j)
      for(i=0;i<row_size;i++)
           {
            for(j=0;j<col_size;j++)
               {
                  mat[i][j]=i+j;
                  mat2[i][j]=i*j;
               }
           }

      int** mat3 = (int**)malloc(row_size * sizeof(int*));
     #pragma omp parallel for
      for (index=0;index<row_size;++index)
           {
            mat3[index] = (int*)malloc(row_size * sizeof(int));
           }
          for(i=0;i<row_size;i++)
          {
           for(j=0;j<row_size;j++)
             {
                mat3[i][j]=0;
             }
          }

#pragma omp parallel for private(j,k) collapse(3)
for(i=0;i<row_size;i++){
            for(j=0;j<row_size;j++){
                      for(k=0;k<col_size;k++){
                         mat3[i][j]+=mat[i][k]*mat2[k][j];
                    }
            }
      }
}

void main()
{
printf("\nMATRIX MULTIPLICATION: Enter matrix size:");
float st,et,start,end;
printf("----------------------------------------")
          start=omp_get_wtime();
          matrix_mul();
          end=omp_get_wtime();
printf("time=%lf\n",end-start);
}
```