A Source to source Automatic Parallelizing Compiler for a Cluster of Workstations

S. Sanjeeth Pallav Kumar Baruah

Abstract: The large number of sequential programs that takes lot of computational time with single processor needs to be ported onto the cluster of workstations to harness its computational power and to decrease the running time of the programs. One way to achieve this is to rewrite the sequential programs that fit in message-passing paradigm, but this is a time consuming and error prone process. This paper describes a parallelizing compiler that automatically converts the sequential programs in to parallel programs for a COW and distributing the data in such a way that the communication between the processors is minimized. Since nested for loops are the core of scientific and engineering applications, which access large arrays of data, this paper deals with parallelizing the perfectly nested for loops present in the source

code.

Keywords: Parallelizing compiler, dependence, Transformations, communication analysis, data distribution.

1. INTRODUCTION

There exists a lot of sequential programs written for material science, computational fluid dynamics, Image processing and many other scientific disciplines, which takes lot of time when executed in a single processor. In order to overcome this one can take to parallel processing. One way is to rewrite the whole program. The major problem is the difficulty of writing parallel software. Anyone other than the experts in parallel programming would have to not only learn parallel programming but also develop an understanding of the additional complexities in developing parallel software, e.g. data distribution or synchronization problems, that is largely irrelevant to, and takes valuable time away from, the actual research issues.

One solution to the above problems lies in tools that can automatically or semi automatically replace these efficient sequential programs by equivalent, more efficient parallel programs, from a library. One such tool is MPIIMGEN[6], which converts any sequential program written in *C* language for Image Processing applications into a parallel version. But the major disadvantage of that kind of approach is that they are confined to a particular domain like image processing as in the case of MPIIMGEN.

Another approach is to construct an *automatic parallelizing compiler* that converts any sequential program into a parallel program. Most of the existing parallelizing compilers generate their parallel code for shared memory or distributed shared memory systems. There are very few parallelizing compilers, which generate code for COW. POLARIS [8], is an optimizing source to source translator, which converts sequential

Department of Mathematics and Computer Science, Sri Sathya Sai Institute of Higher Learning PrasanthiNilayam, Anantapur District, Andhra Pradesh, INDIA -515 134.

sanjeeths@gmail.com, baruahpk@yahoo.com

FORTRAN 77 program into a parallel program for shared memory and distributed shared memory systems. PARAFRASE-2 [8] is a source to source translator, which converts sequential FORTRAN 77 or C or CEDAR FORTRAN programs into a parallel program for Multithreaded, Shared Memory, and Distributed Shared Memory architectures.

THE PARALLELIZING COMPILER [7] is a compiler constructed using SUIF[8] framework. It generates a message passing parallel code that can run in a cluster of workstations. Here programmer specifies data decomposition specification and compiler makes use of these specifications for distribution the data. It will be difficult for a programmer to specify the best decomposition specification. Otherwise the generated parallel program might take longer time than the corresponding sequential program, due to communication overhead. Also since this compiler doesn't support communication optimization, the performance of the generated parallel code completely depends on the programmer specified data decomposition. These draw backs are overcome in the compiler that we present here. In our work we aim to automatically convert the sequential code into a parallel code capable of running on a cluster of workstations (COWS) and also to distribute the data automatically that minimizes the communication. Unlike the above mentioned attempts we try to achieve parallelism at a higher granularity .The performance results we present at the end of this paper speak volumes about the effectiveness of the strategies adopted in developing the compiler. In Section I we discuss the structure of our compiler. Section 2 to section II explains our parallelization algorithm, data distribution strategy, communication optimization method etc. in brief. Performance analysis of our I compiler is shown in the section 12.

1: Structure

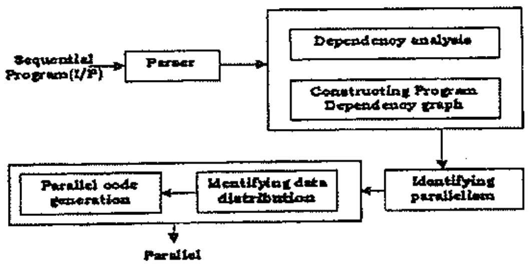


Figure 1. Structure of our Complier

The above figure shows the structure of our parallelizing compiler. The Input to our compiler is a sequential c program and the output is a message passing parallel c code.

2. PARALLELIZATION ALGORITHM

Here we are presenting a new step by step approach for automatic handling of the parallelization. Our compiler parallelizes perfect nested for loops present in the source code. The Parallelization Algorithm involves the following steps

I. Normalization
2. Finding Data dependence and dependence graph construction 3. Finding Strongly Connected Components and Statement Re-ordering.
4. Finding Independent components
5. Identifying the hidden parallelism using loop partitioning and loop skewing
6. Finding out a proper schedule for parallelism.
7. Communication Analysis
8. Finding out the data distribution .
9. Code generation

This algorithm parallelizes the nested loops in the given program. For a k nested for loop, the innermost for loop is level 1 and outermost for loop is level k. In the parallelization algorithm, steps 3 and 5 are performed for statement at all the levels.

.3. NORMALIZATION

A *for* loop is said to be in a *standard* form if it satisfies all the following conditions:

- The initialization part should always be of the form a= 0, where a is the loop index.
- The incrementer should be a unit increment ie. the incrementer should increment its value by 1 only, normalizing the for loop is nothing but converting the for loop which is in non-standard form to a standard form. The main advantage of this conversion is that while doing program analysis we need not consider any issues due to other types of for loops (eg. triangular). Also code generation becomes easier. It also makes implementation simple.

4. FINDING DATA DEPENDENCE AND DEPENDENCE GRAPH CONSTRUCTION

Dependency Analysis is very important step and is present in all the parallelizing compilers. The dependence analysis for arrays or matrix are performed using ZIV or SIV or MIV tests[2] based on the array complexity. Our algorithm finds the dependence between all the statements in the for loop and constructs a dependence graph. Here the edge between the two statements contains the dependence information which includes

- *The Dependent variable.*
- It is the variable on which the statements are dependent.
- Dependent on which loop.
- The set of for loops on which the dependency exists.
- *Type of Dependency.*
- Specifies the type of dependency ie flow or anti or output.
- *Dependence distance* as explained in [2].

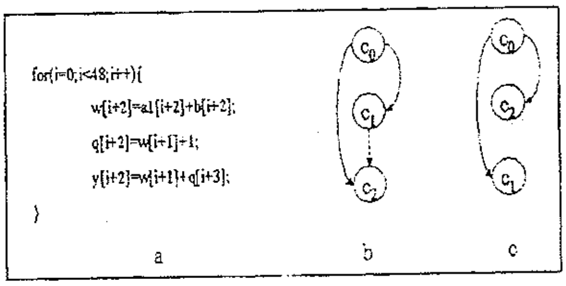
5. FINDING STRONGLY CONNECTED COMPONENTS AND STATEMENT RE ORDERING

The *Strongly Connected Component(SCC)[4]* are the maximal SCRs (SCRs that are not proper subsets of any other SCR).

One of the important steps in our compiler is *loop fission.* It partitions independent statements inside a loop into multiple loops with identical headers. It is used to separate statements that may be parallelized from those that must be executed sequentially. For applying loop fission, we first need to find the cycle in the dependency graph and place all the vertices involved in the cycle in the same component. So we use *SCC* for applying loop fission.

Only finding *SCC* is not enough for applying loop fission since there might be some anti-dependencies between the *SCC's*. It is required to remove all the anti-dependencies between the *SCC's*. This can be achieved by ordering the *SCC's* such that, there is only flow dependency between the components.

For example in the figure 2, the anti-dependency is shown by dotted lines and flow dependency by ordinary straight line. The edge between the component C1 and C2 is anti-dependent. So interchanging the order of the components C1 and C2 removes the anti-dependence. After statement reordering the components



```
for(i=0;i<48;i++){
        w[i+2]=a1[i+2]+b[i+2];
        q[i+2]=w[i+1]+1;
        y[i+2]=w[i+1]+q[i+3];
}
```

Figure 2: Statement Reordering a) sample for loop b) scc components with their dependence edges c) After applying statement Reordering

contains only flow dependence, hence all the components can be placed in separate for loop and now for each component we can identify the parallelism separately.

6. FINDING INDEPENDENT COMPONENTS

Let C be set of scc components and C_i be a subset of C, C_i is said to be a independent component if there exist no path between C_i and $\forall c_i$; where $i \in \{2, , |C|\}$. Once *finding SCC's* and *statement reordering* are performed, there may be a set of Independent components that can be executed parallely.

```
1. for each component $c_i$ from set of SCC components do
   a) for each of the component $c_j$ from set of scc components
         i) do if there is any path or an edge between $c_i$ and $c_j$ then place $c_i$ and $c_j$ in the same group.
   b) if there is no path between $c_i$ and any other components then place it in a seperate new group
```

Figure 3: Algorithm for Finding Independent Components

In the algorithm shown in the figure 3, each component is checked with all the other components for presence of a path. If there exists a path between any two components then, those two components are placed in the same group. If two components are independent, then those components are placed in two different groups. Here total number of groups gives the number of independent components. To achieve parallelism at more granularity we should extract parallelism from each independent component.

7. IDENTIFY THE HIDDEN PARALLELISM USING LOOP PARTITIONING AND LOOP SKEWING

Some program may contain parallelisms that are invisible to us; they are termed as hidden parallelism. These hidden parallelisms can be exposed using loop skewing{5} and loop partitioning{5}. This section explains only the algorithm of these two transformations.

Loop *skewing* is a loop transformation, which adjusts the iteration space of two perfectly nested loops by shifting the work per iteration in order to expose parallelism. The figure 4(a) shows the algorithm for loop skewing. Loop skewing can be applied, only if there is no cycle between the two selected components. But since we have removed all the anti-dependence, there won't be any cycle. So, we only need to test for presence of loop skewing.

```
1. While all the components are analyzed do
   a) Take two adjacent components $c_i$ and $c_{i+1}$ check if loop skewing can be applied if so then combine the components and record skew distance.
                              (a)
1. While all the components are analyzed do
   a) Take two adjacent components $c_i$ and $c_{i+1}$ check if gcd of their cross-iteration dependencies is greater than 1 if so then combine the components and record the type of parallelization and the gcd value.
                              (b)
```

Figure 4:a) Skewing Algorithm b) Loop Partitioning Algorithm

The next technique that we used to expose hidden parallelism is loop partitioning. It works by computing greatest common divisor (GCD) of the cross iteration dependence distances. The algorithm for loop partition is shown in the figure 4(b). This algorithm is similar to that of loop skewing. The gcd value specifies the number of processes that can be used for parallelizing that component. So it should be more than 1.

8: Finding out a proper schedule for parallelism.

Scheduling is allocating tasks to each process such that all the processes can execute simultaneously. Basically, there are two ways of parallelizing the *for* loops they are *statement parallelization* and *Iteration parallelization*.

Statement parallelization is executing more than one statement parallely and *iteration parallelization* is executing set of iterations of a statement parallely. For Scheduling we use either statement parallelization or iteration parallelization.

Before generating the schedule we find out the type of parallelization (statement or iteration) that can be applied to the components in all the levels of the *for* loop. The type of parallelization can be combination of both statement and iteration, that is, a set of statements can be executed simultaneously and each statement in that set can be executed parallely using iteration parallelism. Also statement or iteration parallelism can be combined with loop skewing and loop partition. In our algorithm, statement Parallelization takes precedence over iteration parallelization, since executing different statements in different processes parallely yields more effective parallelization than executing a statement parallely. The algorithm for scheduling is given in the figure 5.

```
1. for each component c_i from the set of components at level j
    a) If the type of parallelization for component c_i is "not parallelizable" then go to next level and get the set of components in that level from where the component c_i is derived and do step with j = j - 1.
    b) else based on the type of parallelization find out the ranks for executing the components.
c) do data distribution.
c) do code generation.
```

Figure 5: Algorithm for finding a proper schedule

In this algorithm we will start from level k, that is, the outermost loop. We visit all the components one by one. Each component Ci is checked for the type of parallelization. If the component cannot be parallelized then go to next level, find out the set of components from where the component Ci is derived, that is, we find out the set of components that contain the statements present in Ci. Do the same process for the derived components. This process continues till we reach level 1. If the component is parallelizable, then number of ranks (processor numbers) that are allocated to the component depends on the type of parallelization and available number of processors. Also Ranks will be allocated based on the communication cost. Next section deals with allocation of ranks.

9. COMMUNICATION ANALYSIS

In this section, we will discuss about our communication algorithm. This algorithm eliminates redundant communication and also resolves unnecessary communication.

Eliminating Redundant Communication:

Root process should hold all the data in the beginning and after completion of the computation. Once all the operations that is performed on the data is completed, the modified data has to be returned back to the root process. Sending the modified data can be performed at the end or after executing each component. Sending data at the end will make improper use of the processor time and also at the end of the computation, there will be lot of communication between root process and other processes. This will lead to some processes becoming idle for long time and also there is a chance of data being lost in the network because of improper buffering. So, we adopt the view that it is necessary to send the modified data after executing each component. But there is a chance that redundant data might be sent by the processes, which occurs due to output dependencies. So, our algorithm does the following: if there is any output dependencies between two components, the variable involved in the output dependencies will not be sent to the root processes, because the component which was the cause for the output dependency will anyway modify the data and send it to root process.

Resolving Unnecessary Communication:

To resolve unnecessary communication, a component will be executed by a process, where the communication cost required is the minimum. Executing in some other process will lead to more and frequent communication between the processes. Allocation of the ranks to the components depends on the communication cost. To handle this we maintain a "processor status" structure that specifies the set of valid data, it is holding. The Structure of the "processor status" is shown in the figure 6.

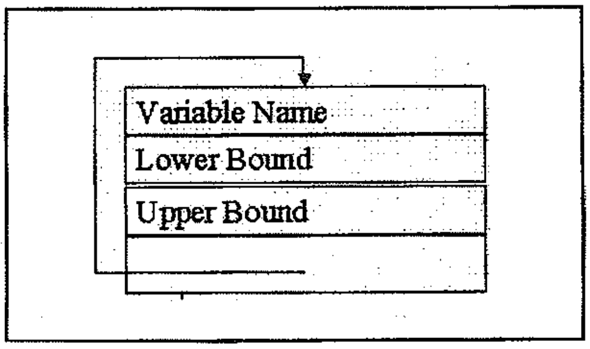


Figure 6. Processor Status Structure

Each entry of Processor Status contains the variable name, lower bound and upper bound. There is one entry which points to the same structure. This entry is to handle matrices, where it specifies the lower bound, upper bound of 2nd dimension in 2D matrix and so on. Each processor will have lot of entries specifying the chunks of data it is holding. While allocation of ranks, we will find out the communication required by each processor (which was not allocated) for executing the component, then we will allocate those ranks whose communication cost is less.

```
1. for each processor status do
    a) for each use in the component do
    b) if the use is present, then find out which portion of the data is unavailable and no. of data required.
    c) if the use is not present, then whole data is required, calculate the no. of data required. Update the no. of data required.
2. Return the processor number that has the minimum no. of components.
```

Figure 7. Algorithm for communication optimization

The algorithm is given in the figure 7. Here the no. of data is number of data required multiplied by bytes occupied by each data. The three steps, namely, Scheduling, Data distribution and Code generation are done simultaneously for each component. That is, as soon as the data distribution strategy is found for a component, the code is generated for that component.

10: Finding out the data distribution Strategy

Once the proper parallelization schedule is identified, the next step is to distribute the data that are required by individual process. The algorithm is given in the figure 8. Here for each of the components we find their corresponding "def' and "use". Then for each use, check whether the required data is present. If it is not present, find out the segment that is not available and generate the MPI calls. The distribution for array or matrix can be block or block cyclic, it depends on the way the given component is parallelized.

```
1. for each of the USE in the component
    a) if the all the required data is present locally,then there is no communication required.
    b) else find out the portion of the data that is required and generate the send and recv calls.
```

Figure 8. Algorithm for data Distribution

In the example shown in the figure 9, data distribution for b is decided by parallelized loop, that is, if the outer most loop is parallelized then we use block distribution (set of rows from array b are sent to different processes) and if the inner loop is parallelized we use block cyclic distribution (set of columns from array b are sent to different processes). Figure 10 shows the mpi code generated for the serial code given in the figure 9. In both the cases the block size depends on the number of processes that are used for parallelizing the component. Here for block distribution we use MPI send and MPI Recv and for block cyclic we use MPI Type Vector for constructing the data type and we use that data type in the send and recv calls.

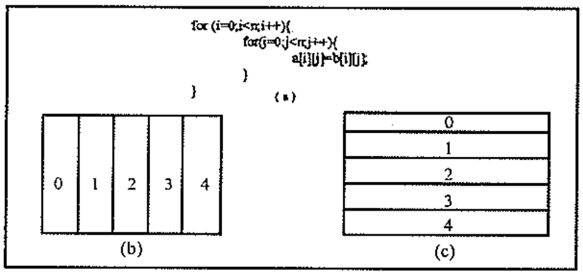


Figure 9: Array Data Distribution a)sample for loop b) array distribution for array b if inner loop is parallelize c) if outer loop is parallelized

```
If(rank==0){
   MPI_Send(b[50][0],500,MPI_INT,1,0, MPI_COMM_WORLD);
   for(i=0;i<50;i++){
     for(j=0;j<100;j++){
        a[i][j]=b[i][j]
   }}}
If(rank==1){
  MPI_Recv(b[50][0],500,MPI_INT,0,0, MPI_COMM_WORLD,
&status);
  for(i=50;i<100;i++){
    for(j=0;j<100;j++){
       a[i][j]=b[i][j]
  }}}
```

(a)

```
If(rank==0){
  MPI_Type_vector(100,50,100,MPI_INT,&column);
  MPI_Send(b[0][50],1,column,1,0,MPI_COMM_WORLD);
//same for loop
}
If(rank==1){
 MPI_Type_vector(100,50,100,MPI_INT,&column);
 MPI_Recv(b[0][50],1,column,0,0,MPI_COMM_WORLD,&status);
  //same for loop
}
```

(b)

Figure 10 : Sample generated mpi code a) Block Distribution b) Block Cyclic Distribution

11. CODE GENERATION

This is the last phase in our compiler. We generate a SPMD message passing code. As soon as data distribution is identified for the component, code is generated for that component with MPI calls added to it. Scalar variables are replicated in all the processors. So, we use MPI Bcast for broadcasting any scalar variables to all the processors. We also use MPI calls for sending and receiving the data. The mpicalls that are used by our compiler are MPI Send; MPI Recv, MPI Bcast, MPI Type Vector, MPI Reduce.

12. PERFORMANCE ANALYSIS

In this section we discuss the performance of the compiler and the communication algorithm we used. For identifying the performance of our compiler, we take a Matrix multiplication benchmark program. Also we take a sequential program containing operations, that are most commonly used in various fields (Mathematics, physics and ImageProcessing) for discussing the effectiveness of our communication analysis algorithm. The associated speed up for the generated parallel program with respect to the serial program, that was given as input to the compiler is determined. Also we compared the predicted or theoretical speed up with the actual speed up. The machine configurations of the nodes in the cluster of workstations, on which the parallel programs were tested, are as follows:

- Each node in the cluster is a 2.4 GHz Intel Pentium 4 processor with 256MB RAM with a 100 mbps LAN

Programme 1 : Matrix Multiplication

Programming analysis was done for matrix with different sizes. The timing on different number processing nodes is given in Table 1.

Figure 11 shows a plot of the speed obtained vs number of processors, for node with different sizes. The speed up predicted is almost linear for 4 processors only. Behind is very large for 500*500 matrix and 800*800 matrix. The reason for the deviation is because the communication cost increases with the processors in number of processes. Therefore the speed up. The figure 12 shows the plot between actual and predicted speed up for a 500*500 matrix multiplication. Here actual speed up is found to be greater than the predicted speed up. This is because, predicted value gives an approximate

Proc no	Speed up			
	500*500		1000*1000	
	Predicted	Actual	Predicted	Actual
1	1	1	1	1
2	1.97	1.96	1.97	1.97
3	2.70	2.71	2.89	2.87
4	3.47	3.53	3.78	3.80
5	3.75	3.8	4.63	4.68
6	4.12	4.34	5.42	5.51

Table 1 : Speed up for Matrix Multiplication

speed up, and it doesn't take care of the communication overlap. Communication overlap reduces the execution time. Hence the speed up increases.

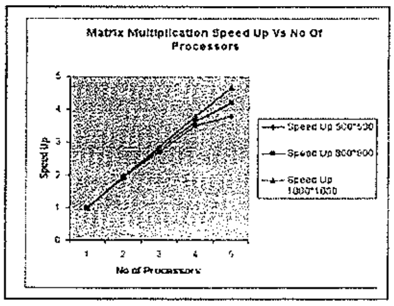


Figure 11: Plot of Speed Up Vs No Of Processors for Solving Matrix Multiplication

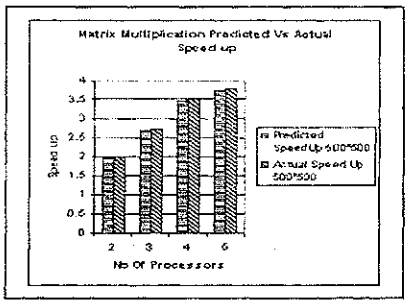


Figure 12: Plot of Predicted Vs Actual Speed Up for 500*500 Matrix

Program 2: Two-Dimensional Convolution

The plot for speed up vs no of processors and comparision of predicted and actual speed up for convolution is shown below

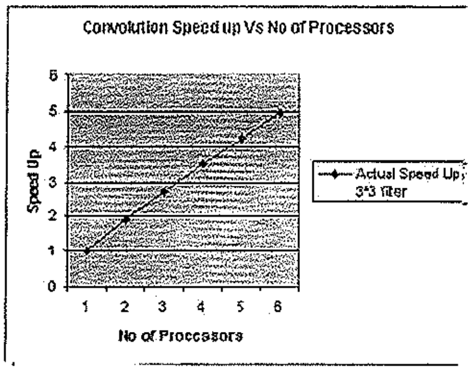


Figure 13: Plot of Speed Up Vs No Of Processors for convolution.

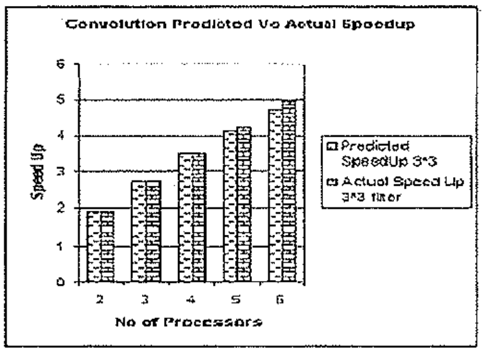


Figure 14: Plot of Predicted Vs Actual Speed Up for convolution.

Program 3

For identifying the performance of our communication algorithm, we will consider two versions of the parallel program generated by our compiler. One using our optimized communication algorithm (with resolving unnecessary communication and Eliminating Redundant Communication) and the other with an unoptimized communication alogrithm. This Program consists of the following operation.

- Finding the numerical integration for three different functions
- Matrix Multiplication
- Convolution
- Solving Wave Equation
- Solving Heat Equation
- Solving Flow Equation

The timing analysis for the generated parallel program with optimized communication and without optimized communication on a cluster of workstations is shown in Table 2.

Proc No	Speed up	
	Without optimized communication	With optimized communication
1	1	1
3	2.91	2.95
4	3.88	3.93
5	4.62	4.81
6	5.27	5.59
7	5.78	6.28
8	6.13	6.716

Table 2 : Speed up for Program 2

Figure 14 shows a plot of speed up obtained vs number of processors for the sequential program. From the graph we can see that the speed up obtained by an optimized version of the parallel program is far better than an unoptimized one.

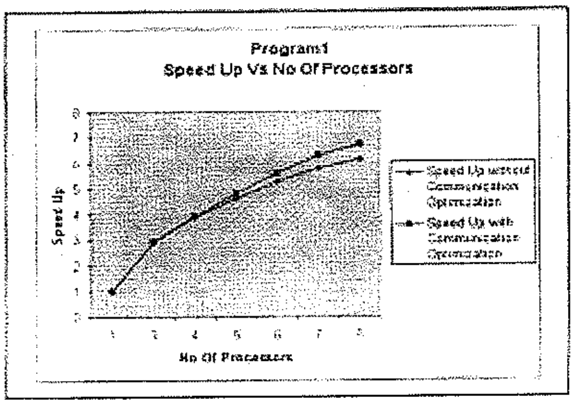


Figure 14: Plot of Speed Up Vs No Of Processors for Program3

Program 4

This Program consists of the following operation

- Applying a Median Filter
- Matching 5 Template images.

The plot of Speed up vs No of processors with and without communication optimization for the above program is given below.

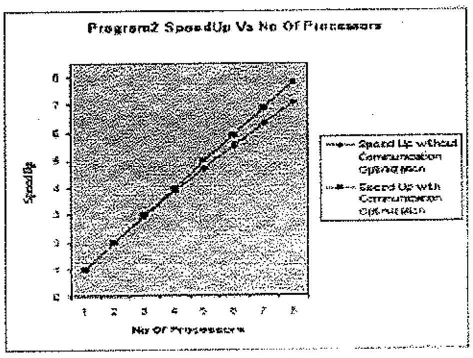


Figure 14: Plot of Speed Up Vs No Of Processors for Program3

The speed up was linear and also parallel program with communication optimization yields better speed up than non-optimized one.

Thus we have shown that our communication algorithm yields better speed up than compared to an unoptimized one.

CONCLUSION

A compiler that converts a sequential code into a parallel code, which can be run on a cluster of workstations, was developed. The performance of the generated parallel codes was evaluated and it was found that the generated parallel codes achieved near linear speed up for computationally intensive operations containing for loops. The core issues of data distribution and communication optimization are addressed. The strategies adopted are found to be performing satisfactorily. Unlike other parallelizing compilers for cluster workstations, here we have an automatic data distribution and communication optimization mechanism built into the compiler.

Acknowledgement: This work was carried out in the A.I.Center, Department of Mathematics and Computer Science, Sri Sathya Sai Institute Iof Higher Learning, supported by the Institute's Academic Research Cell. Authors dedicate this work to the Chancellor of the Institute Bhagawan Sri Sathya Sai Baba.

References:

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

[2] Goff, G. Kennedy, K., and Tseng, C (1991, June). "*Practical Dependence Testing*", roceedings of the SIGPLAN Conference on Programming Language Design and Implementation, Toronto, Canada.

[3] Kathryn S McKinley, "*Automatic and Interactive Parallelization*", P.hd Thesis, Rice University, Houston, Texas March 1994. [4] Michael Wolfe, "*High Performance Compilers For Parallel Computing*", Addison-Wesley Publishing Co., 1996.

[5] Utpal Banerjee, Rudolf Eigenmann and Alexandru Nicolau, "*Automatic Program Parallelization* ", February 1993.

[6] Vinod Varma U and Pallav Kumar Baruah "*MPIIMGEN- A code transformer that parallelizes image processing code on COW*", IEEE International conference on cluster computing, Sept. 20-23 2004, San Diego, California

[7] Yatin Nayak, "*A parallelizing compiler for cluster of workstations* ", Report, April 2000, Department of Computer Science and Engineering, IIT Kanpur.

[8] Kathryn S. McKinley, J. Eliot B. Moss, Sharad K. Singhai, Glen E. Weaver, Charles C. Weems, "*Compiling for Heterogeneous Systems: A Survey and an Approach* ", Department of Computer Science, University of ,'- Massachusetts, Amherst. CMPSCI Technical Report 9582, October 1995.