Evaluating Performance using Message Passing Interface

By:
Jasmina Vasiljevic
Ramya Mohan
Bassim Ibrahim

Professor Nagi Mekhiel

December 9th 2008-12-09

Abstract

Combinatorial optimization problems are common and inherently posses exponentially growing execution times, which often do not lead to global minimums. One common such problem is the placement algorithm in an FPGA. In order for a netlist to be mapped onto an FPGA such that the trace lengths between all the logic blocks are minimized, a suitable physical location has to be determined for each logic block. In this project we implement the simulated annealing algorithm on a placement problem. In order to improve performance, we parallelize the algorithm, and run it on a number of machines using MPI. We examine four different parallelization techniques. Based on preliminary results, we chose the best one, Independent Sets [jas2] , and fully implemented it. As a result, we achieved speedups very close to the number of machines used.

Table of Contents

1. Message Passing Architecture

1.1 Introduction

Message passing systems provides alternative methods for communication and movement of data among multiprocessors. A message passing system typically combines local memory and the processors at each node of the interconnection network. There is no global memory so it is necessary to move data from one local memory to another by means of message passing. This is typically done by send/receive pairs of commands, which is written into the application software. Each processor has access to its own local memory and can communicate with other processors using the interconnection network as shown in Figure 1.1. These systems eventually gave way to internet-connected systems where the processor/memory nodes are cluster nodes, servers, clients, or nodes in grater grid.

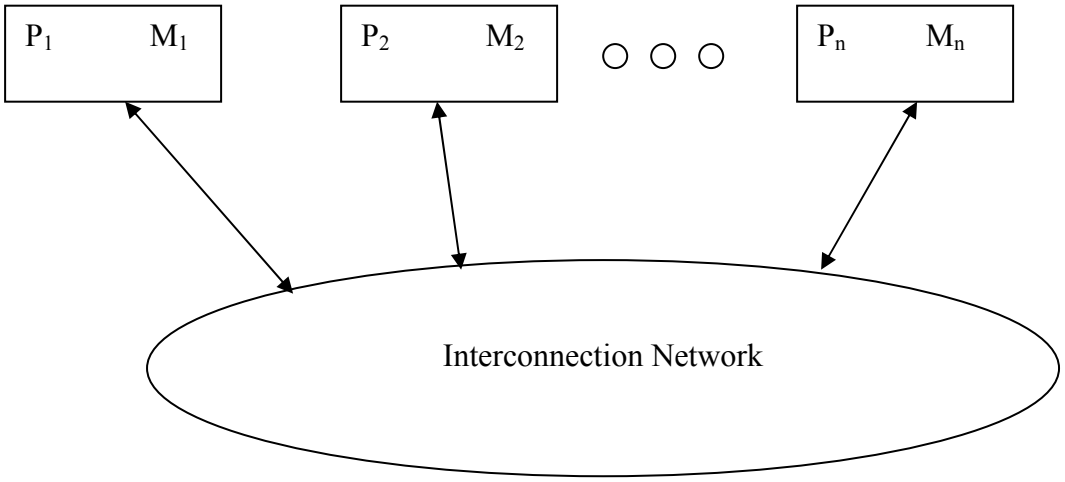


Figure 1.1 Message Passing Systems

1.2 Introduction to Message Passing

A message passing architecture is used to communicate data among a set of processors without the need for global memory. The basis for the scheme is that each processor has its own local memory and communicates with other processors using messages. The elimination of the need for a large global memory together with its synchronization requirement, gives message passing schemes an edge over shared memory schemes.

Figure 1 shows the main components of the message passing architecture. There are n nodes in the figure. A node N_i consists of processor P_i and a local memory M_i. Each processor has its own address space. Nodes communicate with each other by links (external link) and via an interconnection network. Two important factors must be considered in designing message passing interconnection networks: *link bandwidth* and *latency*. The link bandwidth is defined as the number of bits that can be transmitted per unit of time (bits/s). Network latency is defined as the time to complete a message transfer through the network.

Traditionally, software has been written for serial computation to be run on a single computer having a single Central Processing Unit (CPU). A problem is broken into a discrete series of instructions. Instructions are executed one after another .Only one instruction may execute at any moment in time Figure 1.2.

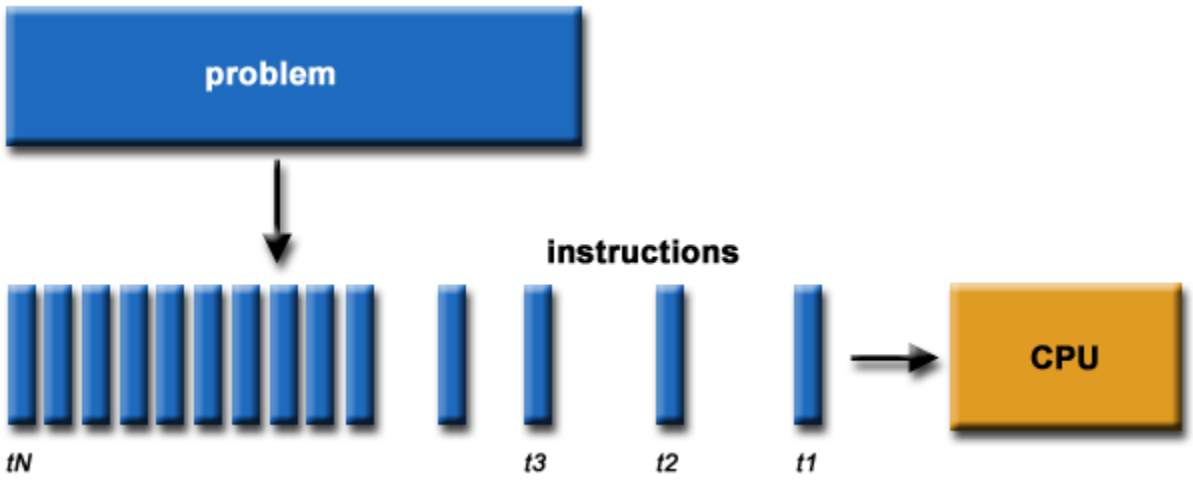


Figure 1.2 Serial computation

In executing a given application program using message passing, the program is divided into concurrent processors; each is executed on a separate processor Figure 1.3. If the number of processes is larger than the number of the number of processors, then more than one process will have to be executed on a processor in a time-shard fashion .processes running on a given processor use *internal channels* to exchange messages among themselves. Processes running on different processors use the external channels to exchange messages. Data exchanged among processors cannot be shared; it is rather copied (using send/receive messages). An important advantage of this form of data exchange is the elimination of the need for synchronization constructs, such as semaphores, which result in performance improvement. In addition, a message passing scheme offers flexibility scalable in accommodating a large number of processors in addition to being readily scalable.

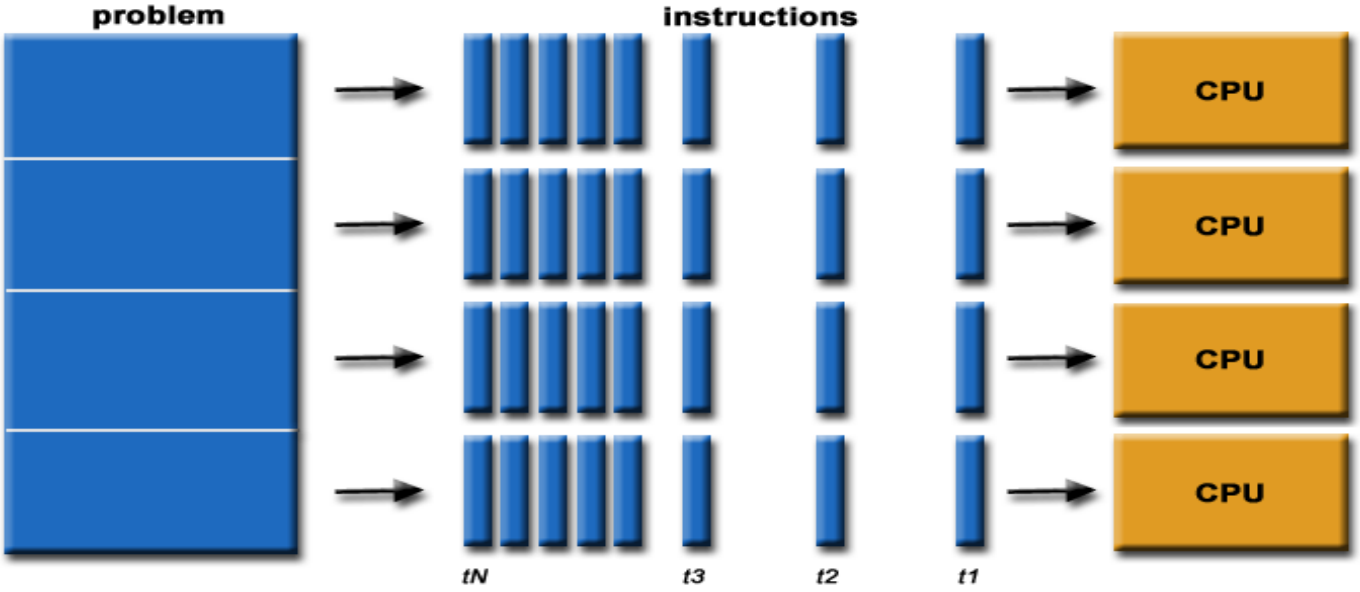


Figure 1.3 Parallel computation

1.3 Advantages of Parallel Programming

- Need to solve larger problems
  - more memory intensive
  - more computation
  - more data intensive

- Parallel programming provides
  - more CPU resources
  - more memory resources
  - solve problems that were not possible with serial program
  - solve problems more quickly

1.4 Parallel Programming goals

- Reduce execution time :
  - computation time
  - idle time - waiting for data from other processors
  - communication time - time the processors take to send and receive messages
- Load Balancing
  - divide the work equally among the available processors

- Where possible - overlap communication and computation
- Many problems scale well to only a limited number of processors

1.5 Message:

A message is made up of an array of elements of a particular MPI data type. The basic MPI data types correspond to the basic C or Fortran data types. The data types corresponding to C are tabulated in Table 1.

MPI Datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED_INT	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Table 1 MPI data type corresponding to C

The data type specified in a receive call must match the data type specified in the send call. However, if different processors store the data type in different ways, MPI is able to deal with this.

In addition to the data being passed, each message contains a communication envelope. This contains specific information that enables the messages to be distinguished. In other words, the envelope provides information on how to match sends to receives.

2. Communications

2.1 Point-to-Point

MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation.

- There are different types of send and receive routines used for different purposes. For example:

  - Synchronous send
  - Blocking send / blocking receive
  - Non-blocking send / non-blocking receive
  - Buffered send
  - Combined send/receive
  - "Ready" send

- Any type of send routine can be paired with any type of receive routine.
- MPI also provides several routines associated with send - receive operations, such as those used to wait for a message's arrival or probe to find out if a message has arrived.

2.1.1 Blocking vs. Non-blocking

- Most of the MPI point-to-point routines can be used in either blocking or non-blocking mode.
- **Blocking:**

  - A blocking send routine will only "return" after it is safe to modify the application buffer (your send data) for reuse. Safe means that modifications will not affect the data intended for the receive task. Safe does not imply that the data was actually received - it may very well be sitting in a system buffer.
  - A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send.
  - A blocking send can be asynchronous if a system buffer is used to hold the data for eventual delivery to the receive.
  - A blocking receive only "returns" after the data has arrived and is ready for use by the program.

- **Non-blocking:**

    - Non-blocking send and receive routines behave similarly - they will return almost immediately. They do not wait for any communication events to complete, such as message copying from user memory to system buffer space or the actual arrival of message.
    - Non-blocking operations simply "request" the MPI library to perform the operation when it is able. The user can not predict when that will happen.
    - It is unsafe to modify the application buffer (your variable space) until you know for a fact the requested non-blocking operation was actually performed by the library. There are "wait" routines used to do this.
    - Non-blocking communications are primarily used to overlap computation with communication and exploit possible performance gains.

2.2 Collective Communication

Collective communication is defined as communication that involves a group of processes Figure 2.1. The functions of this type provided by MPI are the following:

- Barrier synchronization across all group members
- Broadcast from one member to all members of a group
- Gather data from all group members to one member
- Scatter data from one member to all members of a group
- A variation on Gather where all members of the group receive the result
- Scatter/Gather data from all members to all members of a group (also called complete exchange or all-to-all)
- Global reduction operations such as sum, max, min, or user-defined functions, where the result is returned to all group members and a variation where the result is returned to only one member
- A combined reduction and scatter operation
- Scan across all members of a group (also called prefix)
-

2.2.1 All or None

- Collective communication must involve **all** processes in the scope of a communicator. All processes are by default, members in the communicator MPI_COMM_WORLD.
- It is the programmer's responsibility to insure that all processes within a communicator participate in any collective operations.

2.2.2 Types of Collective Operations

- **Synchronization** - processes wait until all members of the group have reached the synchronization point.
- **Data Movement** - broadcast, scatter/gather, all to all.
- **Collective Computation** (reductions) - one member of the group collects data from the other members and performs an operation (min, max, add, multiply, etc.) on that data.

2.2.3 Programming Considerations and Restrictions

- Collective operations are blocking.
- Collective communication routines do not take message tag arguments.
- Collective operations within subsets of processes are accomplished by first partitioning the subsets into new groups and then attaching the new groups to new communicators

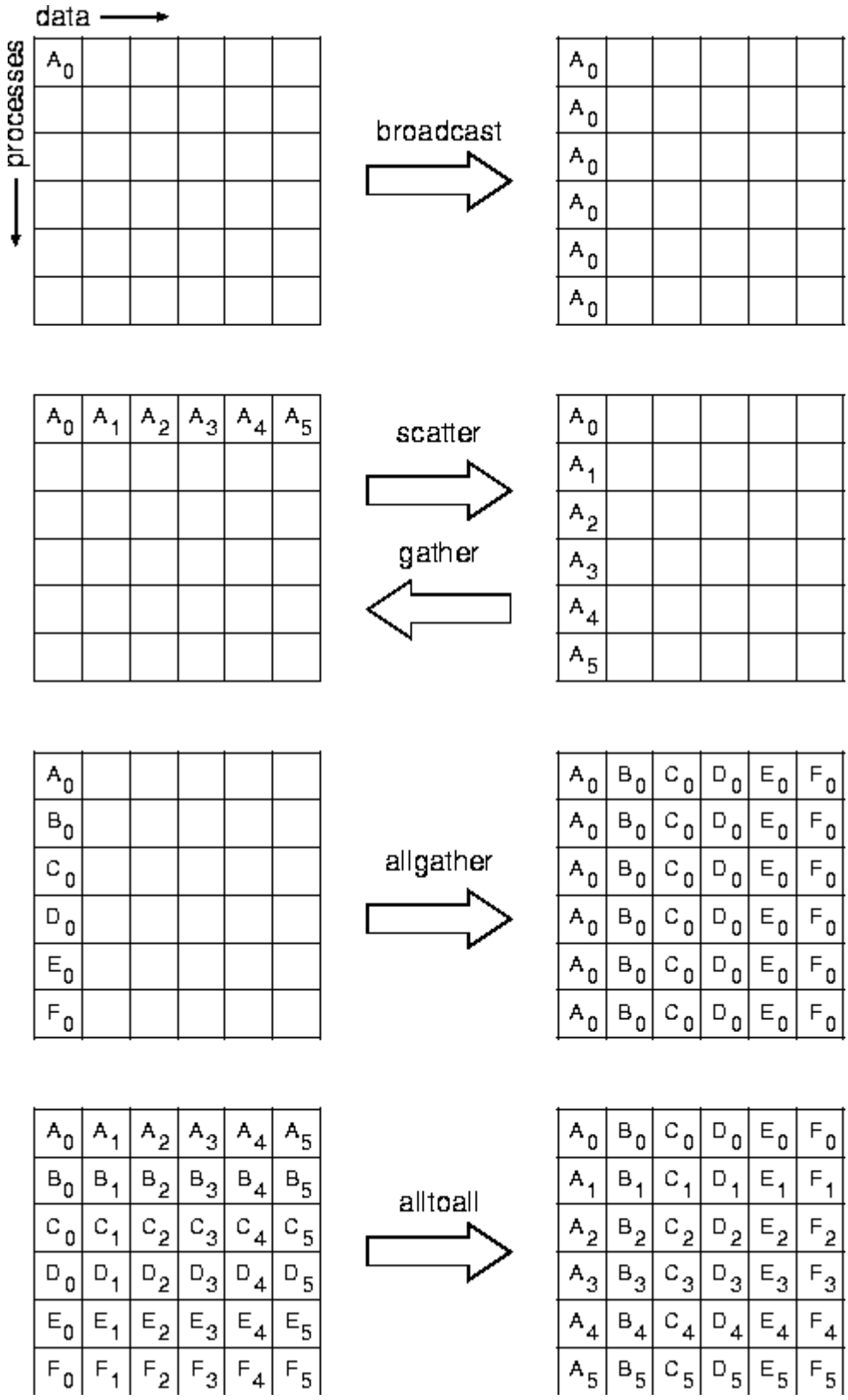


Figure 2.1 Collective communications

2.3 MPI Programming:

The complete MPI specification consists of about 129 calls. However, a beginning MPI programmer can get by with very few of them (6 to 24). All that is really required is a way for the processes to exchange data, that is, to be able to send and receive messages.

The following are basic functions that are used to build most MPI programs.

- All MPI/C programs must include a header file mpi.h.
- All MPI programs must call MPI_INT as the first MPI call, to initialize themselves.
- Most MPI programs call MPI_COMM_SIZE to get the number of processes that are running
- Most MPI programs call MPI_COMM_RANK to determine their rank, which is a number between 0 and size-1.
- Conditional process and general message passing can take place. For example, using the calls MPI_SEND and MPI_RECV.
- All MPI programs must call MPI_FINALIZE as the last call to an MPI library routine.

So we can write a number of useful MPI programs using just the following 6 calls MPI_INIT, MPI_COMM_SIZE, MPI_COMM_RANK, MPI_SEND, MPI_RECV, MPI_FINALIZE.

2.4 Compiling and Running MPI Applications:

The details of compiling and executing MPI/C protgram depend on the system. Compiling may be as simple as

```
g++ -o executable filename.cc –lmpi
```

However, there may also be a special script or makefile for compiling. Therefore, the most generic way to compile MPI/C program is using mpicc script provided by some MPI implementations.

To execute MPI/C program, the most generic way is to use a commonly provided script mpirun. Roughly speaking, this script determines machine architecture, which other machines are included in virtual machine and spawns the desired processes on the other machines. The following command spawns n copies of executable.

mpirun -np n executable

2.5 Installing MPICH2

- Get source code from:
  http://www.mcs.anl.gov/research/projects/mpich2/downloads/index.php?s=downloads
- Unpack downloaded file mpich2.tar.gz:
  tar xfz mpich2.tar.gz
- Create install directory
  mkdir ~/mpich2-install
- Configure: MPICH2:
  configure \-prefix=~/mpich2-install |& tee configure.log
- BuilMPICH2:
  make |& tee make.log
- Install the MPICH2 commands:
  make install |& tee install.log
- Add the bin folder to the PATH
- At Ryerson EE department you must do this:
- Edit **.myzshrc** and add the following:
  PATH=~/mpich2-install/bin:$PATH
  export PATH
- This will set the path variable so that all the mpi binaries would be accessible directly.
- It is necessary for running the programs on all the machines.
- Check that everything is in order at this point by doing:
  which mpd
  which mpicc
  which mpiexec
  which mpirun
- Create a file .mpd.conf in home directory:
  touch .mpd.conf
  chmod 600 .mpd.conf
- Add a secretword to the file:
  echo "secretword=mr45-j9z" >.mpd.conf
- Add host names to config file mpd.hosts
- To avoid the password prompt:
  cd ~/.ssh
  ssh-keygen -t rsa
  cp id_rsa.pub authorized_keys
- Start the mpi daemon
  mpdboot -n 5 -f mpd.hosts

- Test the ring:
  mpdtrace
- Test how long it takes a message to circle this ring with:
  mpdringtest
- Exit All mpd daemons
  mpdallexit

3. Net List

Netlist is a text description of the circuit connectivity. It is basically a list of connectors, a list of instances, and for each instance, a list of the signals connected to the instance terminals. Attribute information is also present in the netlist. The increasing complexity of digital signal-processing (DSP) algorithms in embedded applications, including image and control processing algorithms, requires high processing power to satisfy the real-time constraints often imposed by such applications. This processing power can be achieved by parallel processing devices and parallel multiprocessor architectures. The word netlist in the field of electronic design describes the connectivity of an electronic design. Netlists usually convey connectivity information and provide nothing more than instances, nets, and perhaps some attributes. If they express much more than this, they are usually considered to be a hardware description language such as Verilog, VHDL, or any one of several specific languages designed for input to simulators. Netlists are meant to convey connectivity information, and if there is more data than this basic information in the description, then it may not be just a netlist.

There are several kinds and classes of "netlist":
1. Physical
2. Logical

and also, netlists can of two major classes:
1. Instance based
2. Net based.

Netlists can also be
1. Flat
2. Hierarchical

Hierarchical Netlists can be
1. Folded
2. Unfolded.

3.1 Contents and Structure of a Netlist

Most netlists either contain or reference descriptions of the parts or devices used. Each time a part is used in a netlist, this is called an "instance". Thus, each instance has a "master", or "definition". These definitions will usually list the connections that can be made to that kind of device, and some basic properties of that device. These connection points are called "ports" or "pins", among several other names.
An "instance" could be any form of electronic device. Instances have "ports". In any consumer electronic device, these ports would be the two (or three!) metal prongs in the plug. Each port has a name, e.g. "Neutral", "Hot" and "Ground" of plug. Usually, each instance will have a unique name, so that if you have two instances of the same descriptions, they can be identified (e.g instanca1, instance2). Besides their names, they might otherwise be identical.

Nets are the "wires" that connect things together in the circuit. There may or may not be any special attributes associated with the nets in a design, depending on the particular language the netlist is written in, and that language's features.

Instance based netlists usually provide a list of the instances used in a design. Along with each instance, either an ordered list of net names are provided, or a list of pairs provided, of an instance port name, along with the net name to which that port is connected. In this kind of description, the list of nets can be gathered from the connection lists, and there is no place to associate particular attributes with the nets themselves. E.g SPICE

Net-based netlists usually describe all the instances and their attributes, then describe each net, and say which port they are connected on each instance. This allows for attributes to be associated with nets. E.g EDIF.

3.2 Hierarchy

In large designs, it is a common practice to split the design into pieces, each piece becoming a "definition" which can be used as instances in the design. For example, in an electronic device, the definition not only includes the information about the ports but also includes a full electrical description of the internals of the device, with the motors, switches, etc., inside it. A definition that includes no instances would be referred to as "primitive", or "leaf", among other names; and a definition that includes instances would be "hierarchical".

A "folded" hierarchy allows a single definition to be represented several times by instances. An "unfolded" hierarchy will not allow a definition to be used more than once in the hierarchy. Folded Hierarchies can be extremely compact. A small netlist (for instance, just a few hundred instances) could describe connections with tens or hundreds of thousands of instances this way. For example, definition "A" is a simple primitive memory cell. If definition "B" contains 32 instances of "A", If definition "C" contains 32 instances of "B". Further suppose "D" contains 32 instances of "C", and "E" contains 32 instances of "D". At this point, the design contains a total of 5 definitions (A through E), and 128 total instances. Yet, E describes a circuit that contains 1,048,576 instances of "A"!

A "Flat" design is one where only instances of primitives are allowed. Hierarchical designs can be "exploded" or "flattened" into flat designs via recursive algorithms. "Explosion" can be a very apt term if the design was highly folded (as in the previous example). Also, folded designs can be "unfolded", by creating a new copy (with a new name) of each definition each time it is used. This will generate a much larger database if the design was highly folded, but will also preserve the hierarchy.

By providing a list of the instance names as one descends a folded hierarchy from the top definition to the primitives, one can derive a unique hierarchical path to any instance. These paths can be used to tie a flat design description to a folded hierarchical version of the same design.

3.3 Netlist Output

Sample output of the first 10 N has been show below.

```
The Value of N=1000
Output[1]
Input[1]
Input[43]
Input[16]
Input[1341]
Output[2]
Input[134]
Input[1423]
Input[869]
Input[2148]
Output[3]
Input[783]
Input[2801]
Input[3799]
Input[1099]
Output[4]
Input[1777]
Input[436]
Input[2793]
Input[2257]
Output[5]
Input[166]
Input[661]
Input[3262]
Input[2742]
Output[6]
Input[3057]
Input[3310]
Input[3838]
Input[878]
Output[7]
Input[1707]
Input[3810]
Input[3358]
Input[3692]
Output[8]
Input[3243]
Input[1804]
Input[2419]
Input[2646]
Output[9]
Input[2399]
Input[2197]
Input[2880]
Input[456]
```

4. Case Study

4.1 FPGA Placement Problem

Programming an FPGA is a process composed of the following steps:

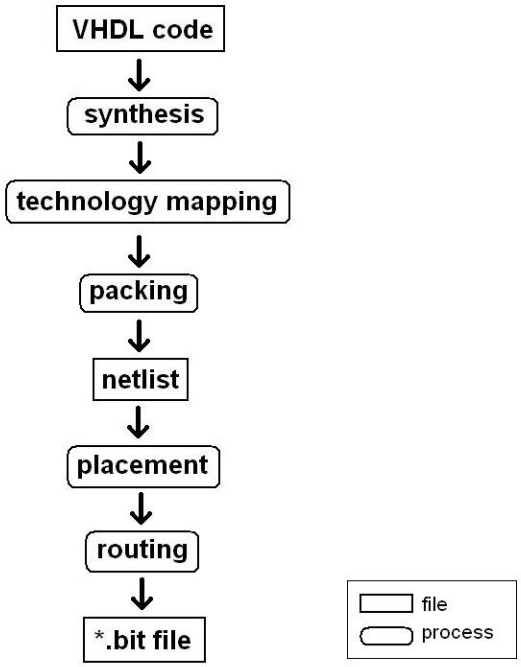


Figure 4.1 – The Compilation Process

First, the VHDL code is compiled into a netlist; this is called the synthesis process. Second, the netlist is technology-mapped into logic gates. Packing then places the logic gates into logic blocks. Next, a placement algorithm determines the physical location of the logic blocks in the FPGA. Finally, routing between the logic block connects the gates and completes the compilation cycle. Each of these steps is performed by a separate algorithm, and often by a combination of processes. The place and route algorithms optimize the placement so that the trace length between the logic units is minimized. This is necessary in order to meet common requirements, such as timing and area.

In this project, we will examine and implement a simulated annealing algorithm to perform placement.

An FPGA can be represented by an array, where each array cell represents a logic block. In our implementation, for simplicity, we assume that each logic block has a single element: one logic gate. Each logic block can store a small portion of an equation, or logic, such as a look-up table. Most common look-up tables in today's mainstream FPGAs have four inputs. Studies showed that this is one of the optimum input sizes[Jas1].

After the packing step, a netlist is generated. In our implementation we use a netlist as an input to the algorithm. The placement is performed for the given netlist, and the output of the algorithm is theoretically ready for routing.

4.1.1 Importance of Placement

Placement is a combinatorial optimization problem. As the number of logic blocks grow, the algorithm execution time grows exponentially. The trend in FPGA development has shown a consistent increase in size, and as a result, a huge increase in compilation time. Today's top of the line FPGAs, such as the Xilinx Virtex 5, can take up to 6 to 8 hours in order to compile a full algorithm (one that takes up the whole FPGA). This is an extremely long time. As the FPGAs grow further, so will the compile time, which makes this problem a very good candidate for parallelization.

4.2 Simulated Annealing

4.2.1 Introduction to Simulated Annealing

Simulated Annealing (SA) is an algorithm used on combinatorial optimization problems. It is generally reliable for finding the global minimum or close to a global minimum. The name annealing comes from metallurgy, the process of controlled heating and cooling of a material in order to increase the size of molecular crystals, and reduce defects. The controlled temperature ensures that you apply energy to the problem and slowly cool it, in order for the elements to find their optimum arrangement.

Simulated Annealing is used for a variety of applications, including chemistry, biology and computer science. In this project, we will use the simulated annealing algorithm to perform a placement of a netlist onto an array representing an FPGA.

4.2.2 The Algorithm

The SA algorithm is presented below.

```
S = Random Placement();
T = Initial Temperature();
R = Initial Range;

While ( T > 0 ) {    //outer loop
      While (moves per T are not reached) {  //inner loop
            Snew = Move(S, R);
            if cost (Snew) < cost (S); {
                  S = Snew;
            else
                  a = random(0, 1);
```

```
                If (a < e –Δc /T) {
                    S = Snew;
                }
            }
            T = UpdateTemp();
            R = UpdateRange();
}
```

First, the netlist is randomly placed onto the array. The array, as previously mentioned, represents an FPGA.

There two loops in this algorithm. The first loop, represents each temperature step. The temperature in this algorithm plays a very important role, and simulates the cooling process. The temperature decreases with each step. Commonly, the decrease is exponential, such as e^{-x}.

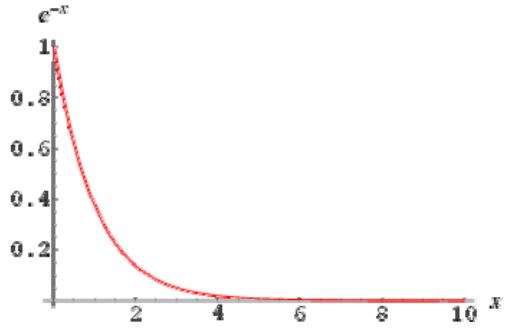


Figure 4.2 Exponential Temperature Decrease

The inner loop performs moves on the logic blocks, or array elements, and calculates the cost. The moves are performed by swapping the location of two logic blocks. Cost, in this case, is represented by the total length of all the traces required to connect the netlist. In this algorithm, cost is the focus, and the task is to optimize it by find the global minimum.

For each switch and cost calculation, the algorithm has to decide if the new placement and the new cost should be kept or ignored. If the new placement is better then the previous one, meaning the cost is lower then before, the new placement is kept and the cost updated.

But, if the new cost is higher then the previous, the algorithm can choose to keep or to ignore it. This decision is performed by a guessing method. The program generates a random number, and then compares it to a threshold. If the random number is less then the threshold, the new, more expensive cost is kept. On the other hand, if the generated random number is bigger then the threshold, the cost is ignored. In this case, the threshold is a function of temperature. As the temperature lowered with each step in the outer loop, the probability of the higher cost being accepted lowers.

This means that at first, when the temperature is high, increasing cost is likely to be accepted. But, when the temperature lowers, only improvements in the cost will be kept, and any increases ignored.

This method helps reach the global minimum. It is compared to giving an element more energy to overcome a local minimum, and hopefully reach the global one.

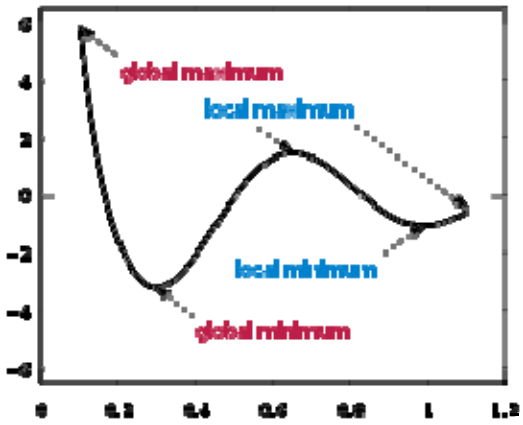


Figure 4.3 Global and Local Minima

Another variable used is Range. It defines the allowed span when the logic blocks are to be switched. Range is also a function of temperature. At the beginning, when the temperature is high, logic blocks can be switched across the whole FPGA. As the temperature decreases, only block close to each other can be swapped.

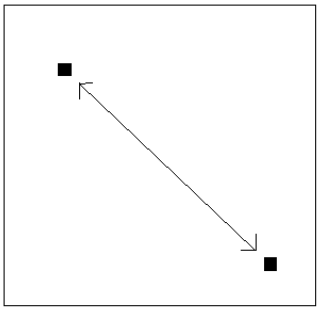


Figure 4.4 Long Range: high temperature switching

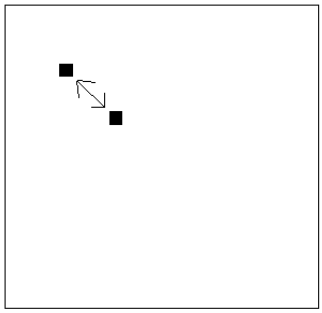


Figure 4.5 Short Range: high temperature switching

4.2.3 Implementation Details

The two dimensional array structure is outline below. The [source] is the output of the logic block, and the [sink-s] are the inputs of the logic blocks it is connected to. Each element in the square brackets is expressed in terms of its x and y position in the array. Essentially, each line in the array represents a trace, with one source (the output of the logic block), and multiple sinks (logic block inputs).

[source], [sink 1], [sink 2] , [sink 3], [sink 4], [sink 5], [sink 6]

[xpos, ypos], [x-d1, y-d1], [x-d2, y-d2], [x-d3, y-d3], [x-d4, y-d4], [x-d5, y-d5], [x-d6, y-d6]
[xpos, ypos], [x-d1, y-d1], [x-d2, y-d2], [x-d3, y-d3], [x-d4, y-d4], [x-d5, y-d5], [x-d6, y-d6]
[xpos, ypos], [x-d1, y-d1], [x-d2, y-d2], [x-d3, y-d3], [x-d4, y-d4], [x-d5, y-d5], [x-d6, y-d6]
[xpos, ypos], [x-d1, y-d1], [x-d2, y-d2], [x-d3, y-d3], [x-d4, y-d4], [x-d5, y-d5], [x-d6, y-d6]
.
..
...

The initial placement of the netlist logic blocks, into the two dimensional array is completely random.

The x, y positions represent the physical location in an FPGA.

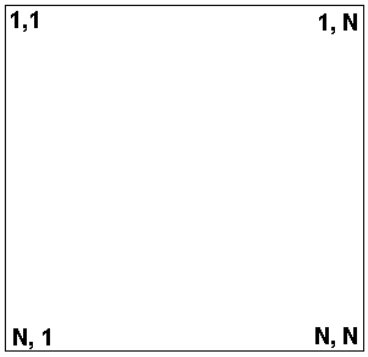


Figure 4.6 The FPGA Array

4.2.3.1 Cost: Semi Perimeter Length

The cost is calculated by a method called The Semi Perimeter Length. The locations of the source and all the sinks are analyzed. For each line in our array, yhe minimum and maximum is selected, for both, the x and the y axis. Then, the difference between the minimum and maximum values is calculated and added together, which results in the final cost.

[xpos, ypos], [x-d1, y-d1], [x-d2, y-d2], [x-d3, y-d3], [x-d4, y-d4], [x-d5, y-d5], [x-d6, y-d6]
[xpos, ypos], [25, 68], [260, 2], [40, 83], [38, 11], [148, 211], [5, 187]

cost_x = max(x-d1, x-d2, x-d3, x-d4, x-d5, x-d6) - min(x-d1, x-d2, x-d3, x-d4, x-d5, x-d6)
cost_y = max(y-d1, y-d2, y-d3, y-d4, y-d5, y-d6) - min(y-d1, y-d2, y-d3, y-d4, y-d5, y-d6)

cost = cost_x + cost_y

This cost function is performed for every line in the array, and every time a logic block switch is made by the algorithm.

4.2.3.1 Switching Blocks

In theory, there are two types of logic block switching which can be applied to this algorithm: displacement and exchange. In our implementation, we will only use cell exchange. This is because we assume that the FPGA is 100% full and there are no empty elements, hence we can only swap logic blocks between themselves, which is called 'exchange'.

We choose a random x, y array position, and we mark this as the first logic block. Then we chose the second random x,y position, within the specified range for the step in question. This is marked as the second logic block. These two logic blocks are then swapped. This is done by switching all of the sink destinations which are generated by the two outputs of the logic blocks in question. Then, the whole array is searched, and if the x,y positions for the two selected logic blocks are encountered, they are updated with their new array placement.

4.3 Partitioned Placements: Parallelization of the Algorithm

Four different parallelization techniques were attempted:

1. Sequential Proposals
2. Independent Sets
3. Partitioned Sets
4. Partitioning by Area

The following is an explanation of each with a description of advantages and disadvantages.

4.3.1 Partitioning By Area

In the original implementation of the algorithm, the range decays exponentially, since it is a function of temperature. In our implementation, we divide the algorithm into two phases dependent on the range. Instead of decreasing the range slowly with temperature, we perform the logic block switching in two different ranges: on the full array, and on quarter of the array, as is shown in Figure XX. This is because we want to parallelize the algorithm, into four segments. This method is called Partitioning by Area, and it is a Fine-Grained Parallel type of move [jas2].

The reason we chose to keep the range at full in Phase 1 is because we want to make the most use out of this phase before we jump into Phase 2.

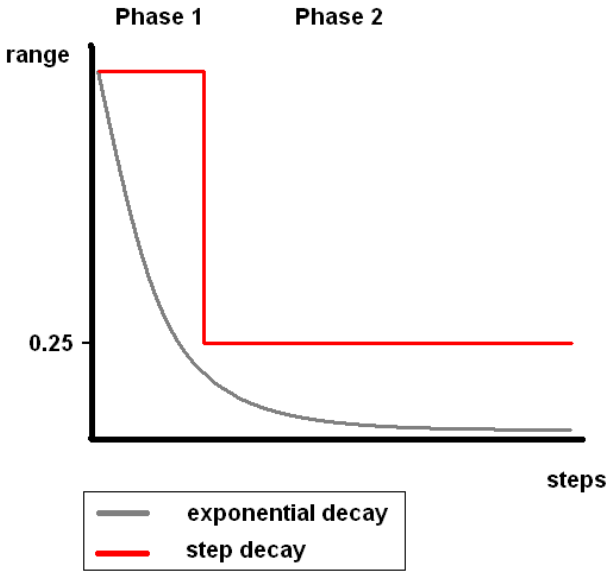


Figure 4.7 Two Phases of Range and Temperature

4.3.1.1 Phase 1

Because in Phase 1, the range spans the entire array, the switching can't be split onto different machines. This phase is performed only on the main server, or node '0'. The length of this process depends on how many temperature steps we allocate to be performed in it.

Clearly, this phase is the bottle neck of the algorithm, but it is necessary since without it the output placement quality would be significantly degraded. In other words, the error of the output placement would be very large, when measured against the sequential, single processor algorithm benchmark.

4.3.1.2 Phase 2

In Phase 2, the placement array is divided into four tasks, as is shown in Figure 4.8.

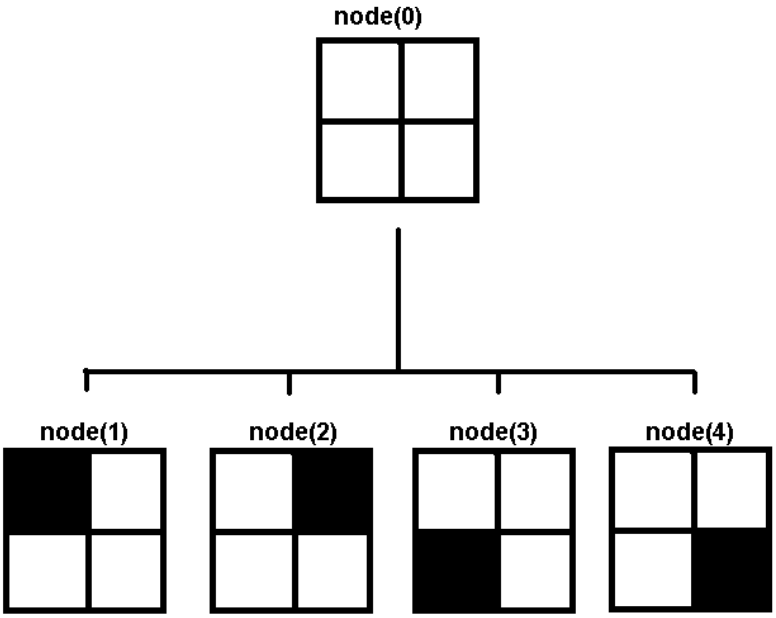


Figure 4.8 Workload Divisions for the Partitioning by Area Parallelization

Initially, node(0) has the full array. At the beginning of Phase 2, the full array is sent to four other machines. The range is now one quarter of the full array, and each machine can modify one designated part. Every machine will be working independently, and there will not be any sharing of elements. Logic block swaps are now performed by each machine only in the designated region. Figure 4.8 shows the four different regions, with the black mask indicating the range of allowed moves for each node.

The point of this partition is to allow each node to work individually on optimizing each quarter, and to achieve better performance. But, even though moves are limited to the specified range, some nets still cross the boundaries into the other quarters. The cost function still spans the entire FPGA. As a result, after a certain number of moves, the calculated cost by each node, has a certain amount of error. In order to remove this error, each node has to update the entire array. If the updating is done periodically, the error can be minimized, and the placement can still be optimized by reaching a close to global solution. The update structure of the array is shown in Figure 4.9 below.

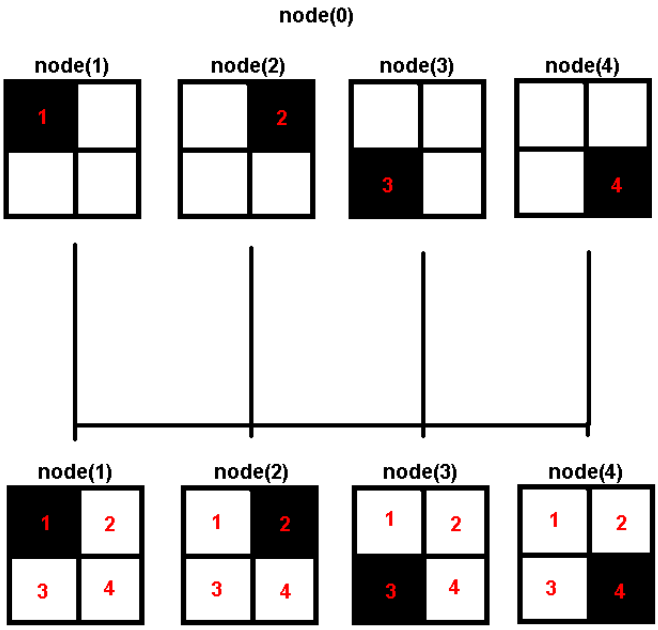


Figure 4.9 The Update Step

In this phase, the number of moves between the updates determines the performance increase of parallelization, since the updates are very time consuming when compared to the swapping steps. In order to maximize the performance, the number of updates needs to be minimized. On the other hand, if we decrease the number of array updates too much, we might sacrifice the quality of the final placement due to error.

4.3.2 Partitioned Sets

In this implementation, we create partitioned sets, where each set is a self-contained problem. One way to partition the sets is to use a partitioning algorithm. The problem would be partitioned at the start, in such a way as to minimize the connected nets between the individual sets. The partitioning would be executed at the beginning of the program and then once during every update between the machines. We tried to find a suitable implementation of the partitioning algorithm. When we estimated the processing time for the partitioning and compared it to the execution time of the SA, we realized that the overhead was too large and decided against this method.

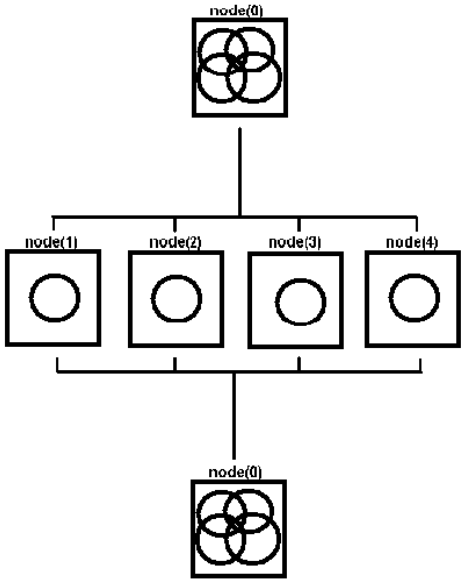


Figure 4.10 Workload Division for Partitioned Sets

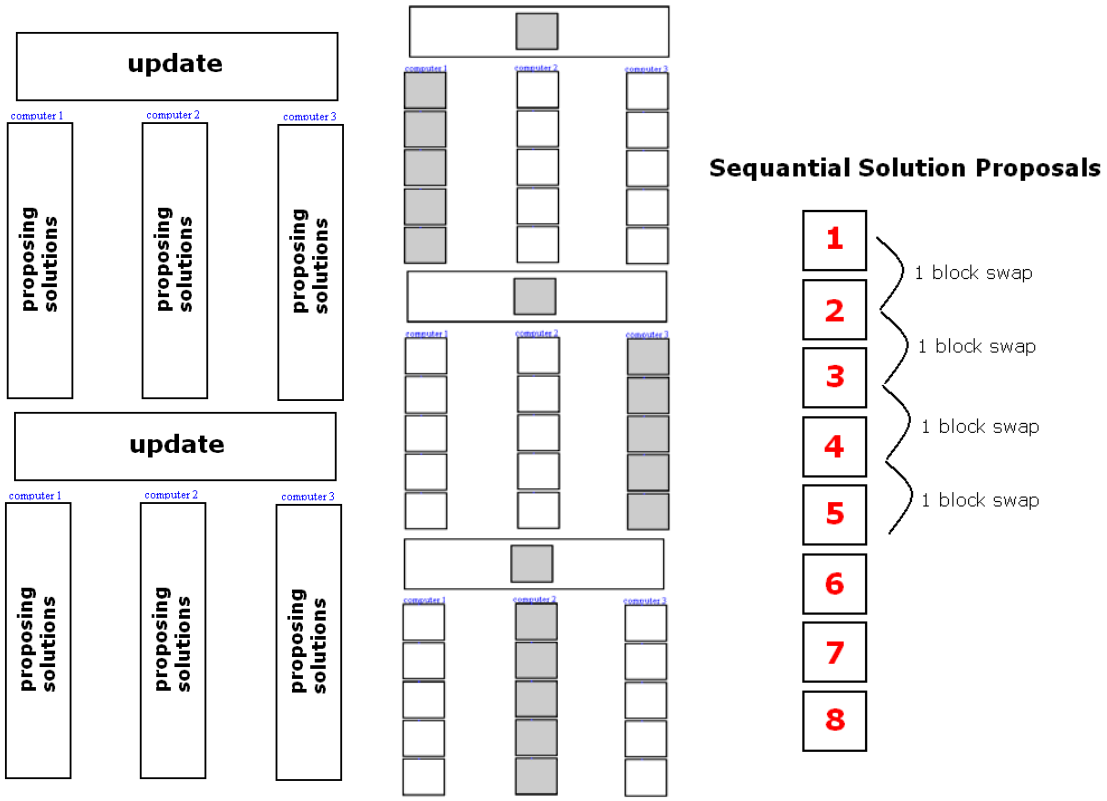
4.3.3 Sequential Proposals:

Instead of passing different problems to each computer, we can pass the same array and ask each computer to analyze it and to try and find a solution. During the update step, all the computers are asked to present their solutions, and the best one is chosen. In our case, the best solution is the one with the lowest cost. Once the solution is chosen, the solution array is updated to all the computers using the broadcast function. This way the overhead is significantly smaller then before. The update step involves comparing a couple of numbers, and then broadcasting the same array. This is very good for scalability.

In the sequential proposals method, each machine performs swaps on the array, and decides whether or not to keep the swap. Then it moves onto the next swap. During the update step, each machine has only one array to present for best-comparison. This method was implemented and tested vigorously.

The resulting speedup of optimization was not significant. Because the swaps are randomly chosen, it is difficult to find a swap which actually decreases cost.

```
for (0 to temp_steps)
        for (0 to computers)
                for (0 : swaps_per_step)
                        propose_new_solution()
                        decide_if_solution_is_accepted()
                end for
                choose_best_solution()
        end for
end for
```



4.11 The Structure of Sequential Proposals

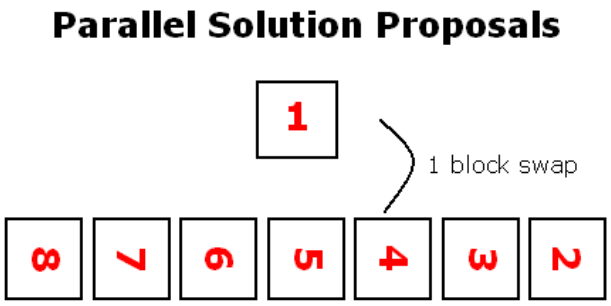
4.3.4 Independent Sets

This method is very similar to the one previously described, except for the way each computer proposes solutions. Each machine receives an array, and proposes a number of solutions which can be done on that array, without incrementing the swaps. Out of all the proposed solutions, each machine chooses the best one, and proposes it during the update step. This way, at every update step, the array changes the most by one block swap.

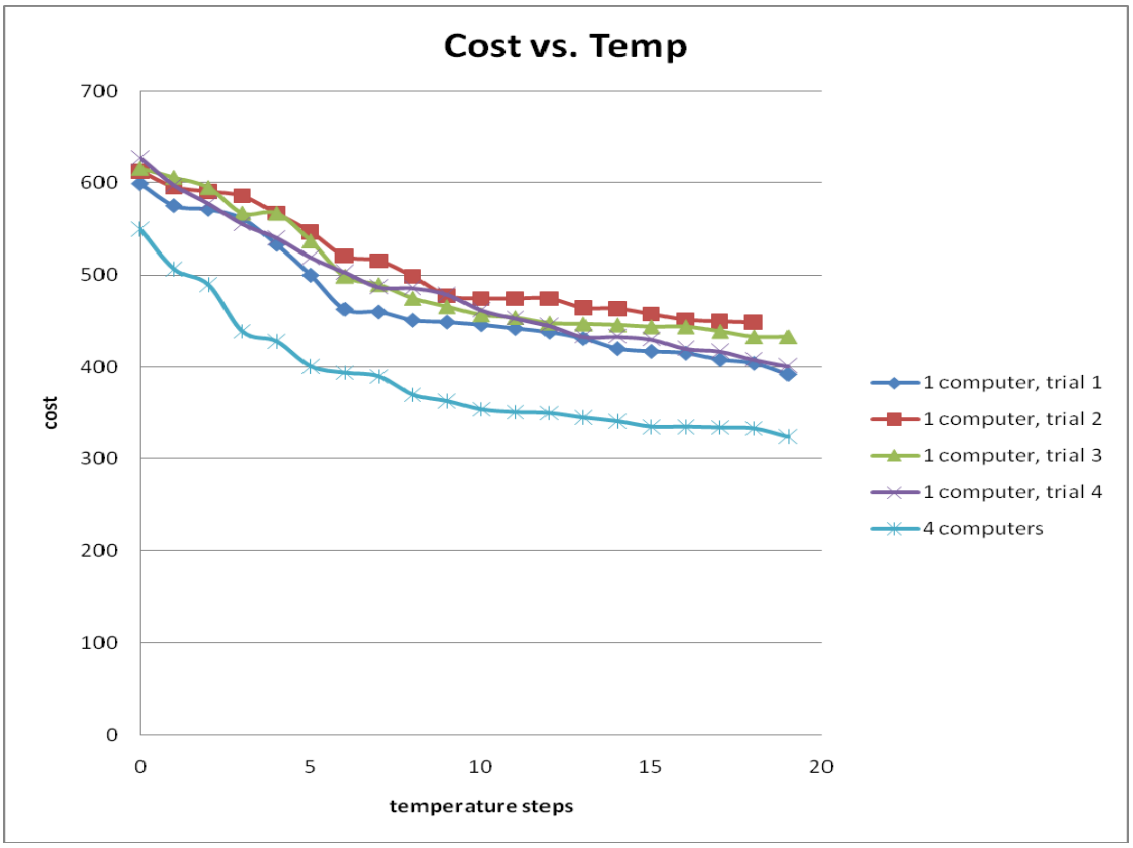
This method was coded and implemented, and testing showed extremely promising results.

The figure below shows the speedup with respect to the Independent Sets parallelization architecture. The cost decreases with each temperature step. Four lines on the graph represent the cost as it is decreasing on a single machine. Each machine randomly chooses a block swap. Because of the random factor, the proposed next step is far from optimal. The bottom line is the algorithm running on four machines. Each machine proposes a step, and out of the four proposed, only the best one is implemented. This decreases the cost much faster.

```
for (0 to temp_steps)
        for (0 : swaps_per_step)
                for (0 to computers)
                        for (0 to comp_steps)
                                propose_new_solution()
                                choose_best_solution()
                        end for
                end for
                decide_if_solution_is_accepted()
        end for
end for
```



4.12 Solution Proposals

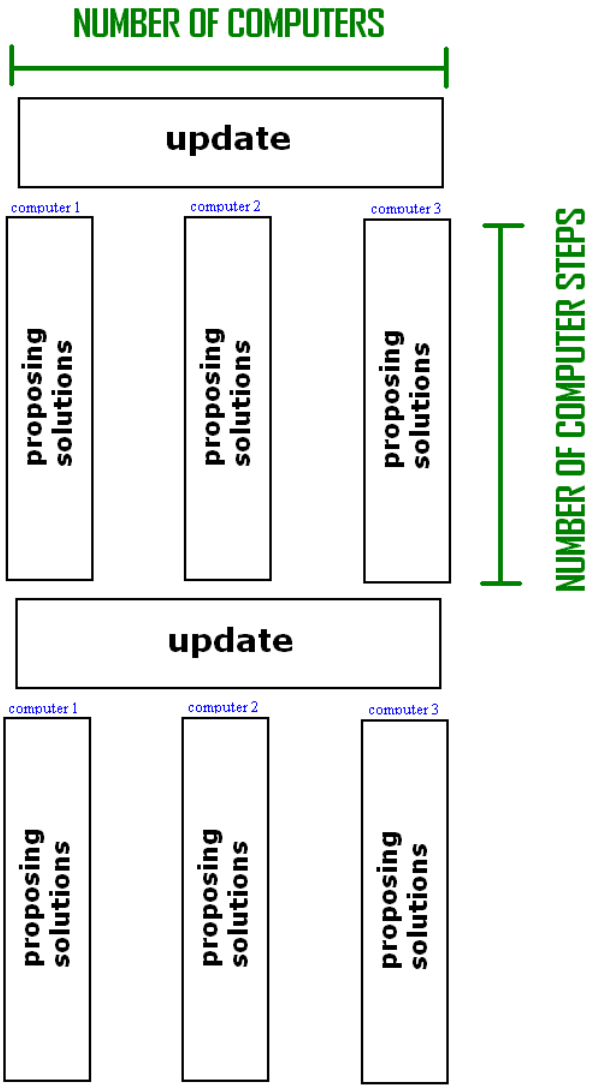


4.13 Performance of the Sequential Proposals Parallelization

5. Variables

We choose three variables for our analysis, and compared how they influence the overall speedup of the algorithm:

1. number of computers
2. number of computer steps
3. size of array



4.14 Variables

The following is a simple calculation of how three factors influence the number of proposed solutions:

1 computer 1 comp_step 20 temp_steps 5 swap_steps ---------- 100 proposed solutions	10 computer 100 comp_step 20 temp_steps 5 swap_steps ---------- 100 000 proposed solutions

5.1 Constants

We set temp_steps to 20, and swap_per_step to 5. These were kept constant throughout all the tests.

5.2 Trials Summary

The following is a list of all the trials we included in the results and analysis:

Array Size	Number of Computers	Comp_steps
100	1	1
100	1	2
100	1	3
100	1	4
100	1	5
100	1	10
100	1	15
100	1	20
100	1	30
100	1	40
100	1	70
100	1	100
100	1	120
100	1	150
100	1	300
100	2	1
100	2	2
100	2	3
100	2	4
100	2	5
100	2	10
100	2	15
100	2	20
100	2	30
100	2	40
100	2	70
100	2	100

100	2	300
100	10	1
100	10	2
100	10	5
100	10	10
100	10	50

Array Size	Number of Computers	Comp_steps
2500	1	1
2500	1	5
2500	1	10
2500	1	10
2500	1	50
2500	1	100
2500	1	200
2500	2	1
2500	2	20
2500	2	50
2500	2	100
2500	2	200
2500	5	1
2500	5	20
2500	5	50
2500	5	100
2500	5	200
2500	10	1
2500	10	5
2500	10	20
2500	10	50
2500	10	100
2500	10	200

Array Size	Number of Computers	Comp_steps
10000	1	5

10000	1	50
10000	1	200
10000	2	20
10000	2	50
10000	2	100
10000	2	200
10000	2	500
10000	2	1000
10000	10	20
10000	10	50
10000	10	200
10000	10	1000
10000	20	20
10000	20	50
10000	20	200
10000	20	1000

6. Results

6.1 Array Size: 100

Figure 5.1 shows how the cost decreases with swap steps performed on the solution array. The three lines represent the cost optimization with one, two and ten computers. We see that all three trials converge to the same global minimum cost, but the more computers we use, the faster we get there.

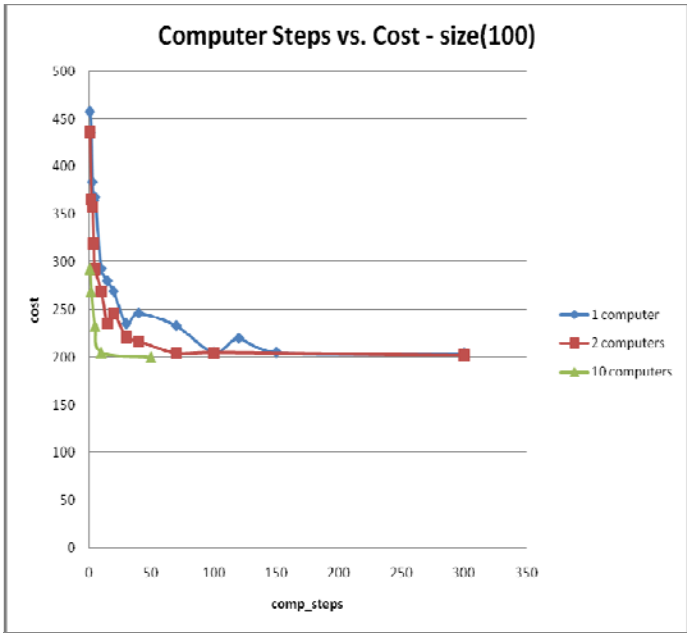


Figure 5.1 Cost Optimization

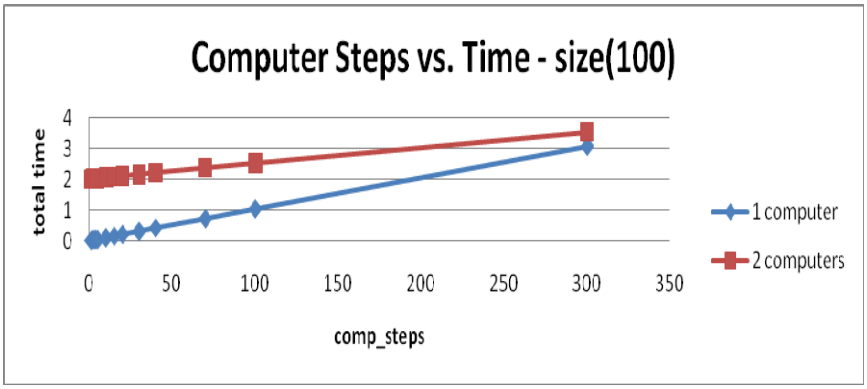


Figure 5.2 Performances of One and Two Computers

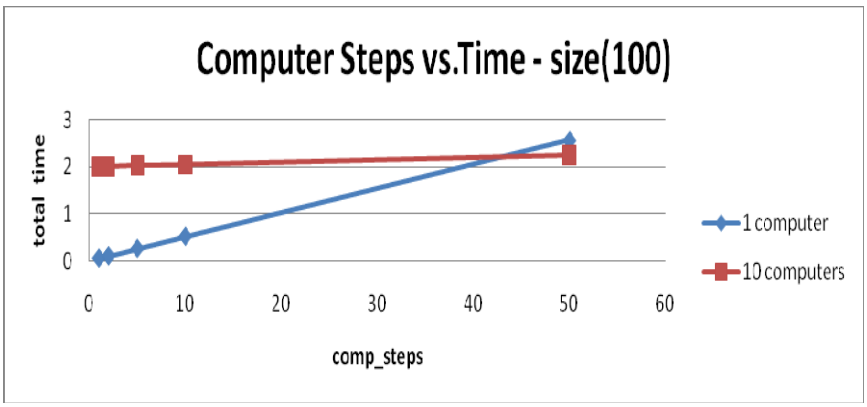


Figure 5.3 Performances of One and Ten Computers

In Figures 5.2 and 5.3 we see show the execution times of one, two and ten computers. We see that, for this array size, using one computer is always faster then two. This is because the problem set is too small and the overhead of communication diminishes any possibility for speedup. When we use ten computers, we see that only after a certain amount of comp_steps, we reach identical execution time. This means that we found the lower bound for this set up.

In conclusion, when using ten computers with an array size of 100, we need a minimum of 50 comp_steps. Generally, this array size is not a good candidate for parallelization.

6.2 Array Size: 2 500

Similarly as before, in Figure 5.4 we see that as we use more computers the cost decreases faster.

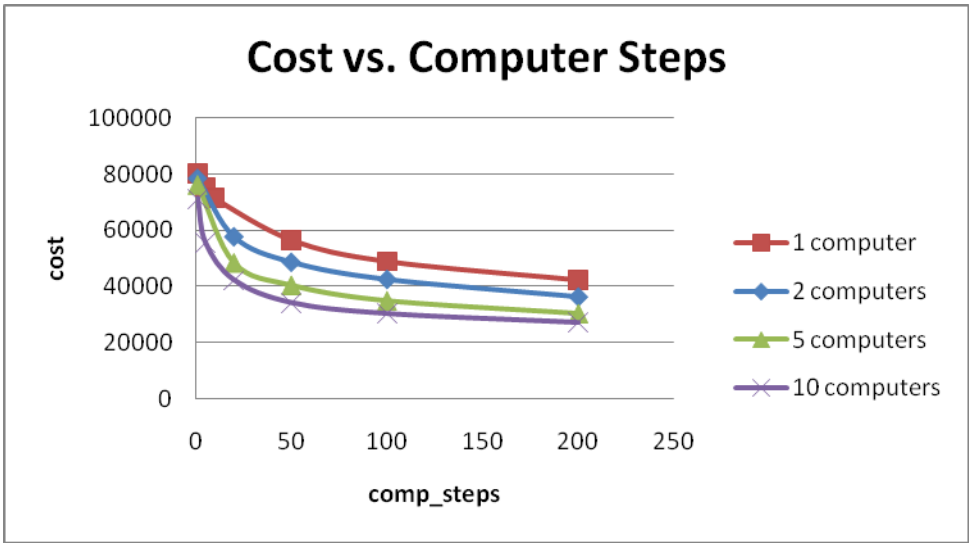


Figure 5.4 Cost Optimization

In the following three figures, we see similar results as before. The more computers we use, the more parallelization we extract. For this array size, one computer is also faster then two. But when we use five computers, we show a lower comp_steps bound of 100, at which point the execution times are equal and everything above is faster with five computers. With ten computers, the lower bound is also present, but it decreases down to around 50 comp_steps.

In conclusion, we see that with larger problems sets we get better speedup results.

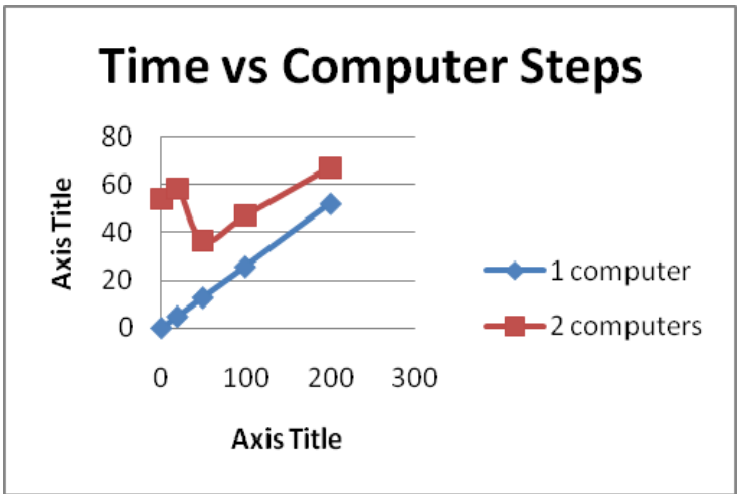


Figure 5.5 Performances of One and Two Computers

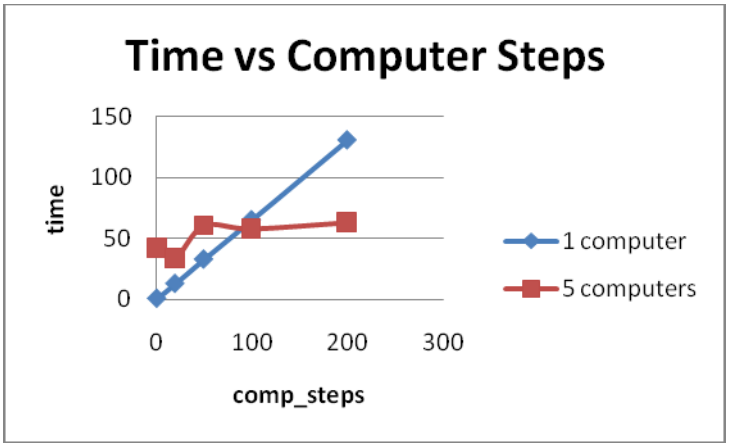


Figure 5.6 Performance of One and Five Computers

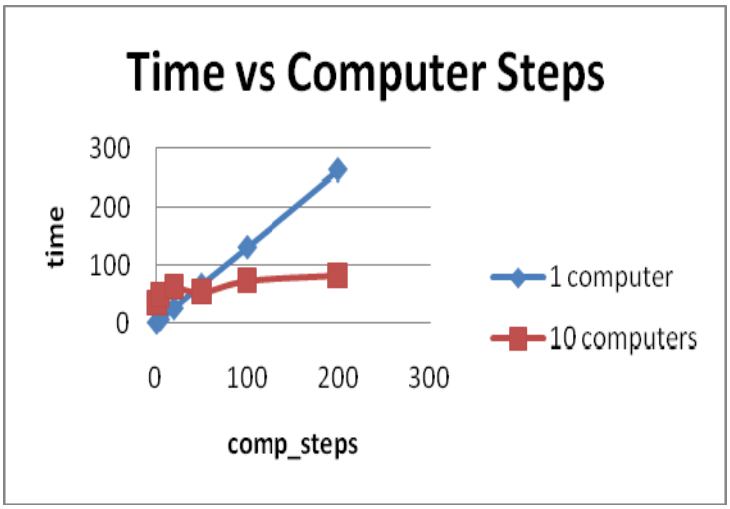
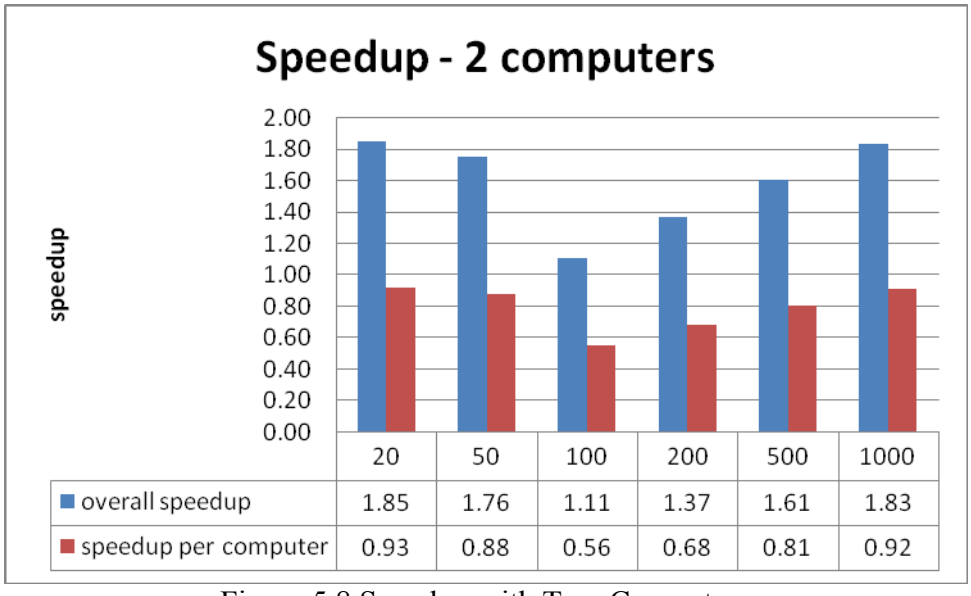


Figure 5.7 Performances of One and Ten Computers

6.3 Array Size: 100 000

With this array size we achieved the best results. They are outlined in Figures 5.8, 5.9 and 5.10. We see that with the really large comp_steps, such as 1000, the speedup factor approaches the number of computers. We also show the speedup factor for each additional computer.



	20	50	100	200	500	1000
overall speedup	1.85	1.76	1.11	1.37	1.61	1.83
speedup per computer	0.93	0.88	0.56	0.68	0.81	0.92

Figure 5.8 Speedup with Two Computers

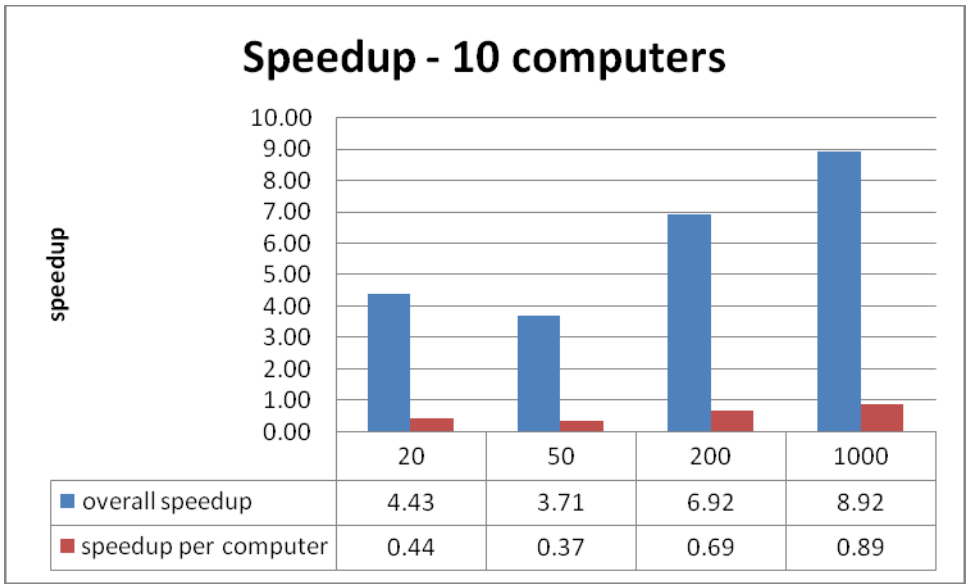


Figure 5.9 Speedup with Ten Computers

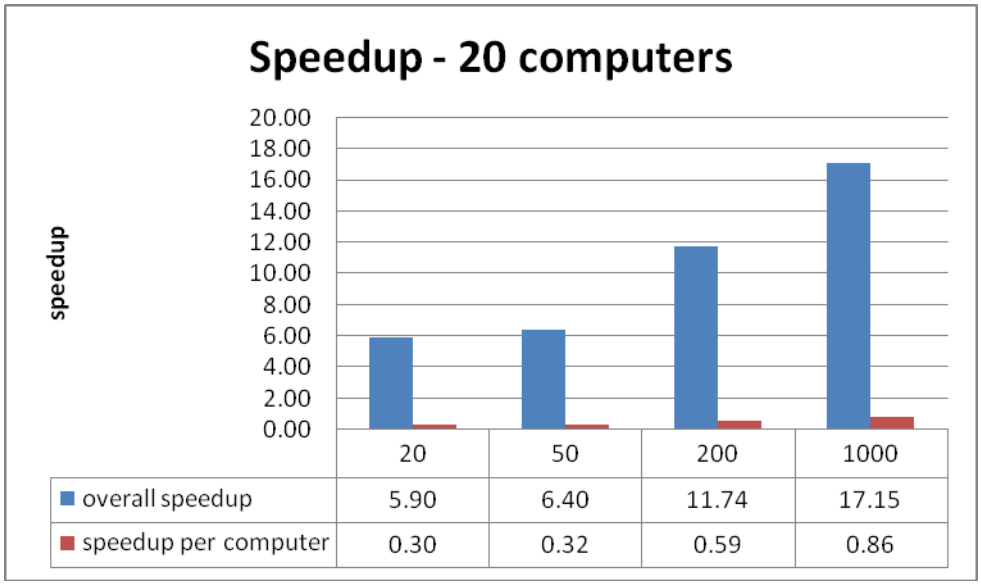


Figure 5.10 Speedup with Twenty Computers

For this array size, we performed an overhead analysis. In Figure 5.11, we show the total time it takes to execute the algorithm. The processing time is shown in blue, and the communication between the computers is shown in red. Clearly, as the workload for each machine increases, the ratio between communication and processing becomes more favourable for parallelization.

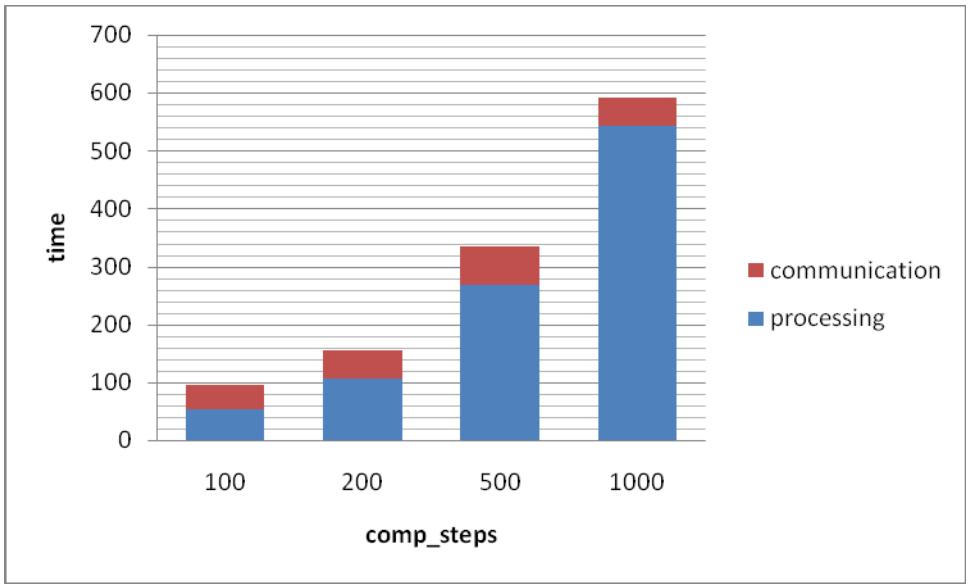


Figure 5.11 Communication Overhead

7. Conclusion

We performed and in-depth analysis of various ways of parallelizing the simulated annealing algorithm. We examined four different parallelization techniques, and based on preliminary results, we chose the best one and performed a full implementation.

We studied three variables: array size, number of computer steps, and number of computers. We measured the effect each one had on the speedup of the algorithm and drew conclusions.

Generally, the larger the problem set, the more suitable it is for parallelization. We showed that smaller problems sets did not perform well. The array size of 100 was much faster on one computer then on two.

Secondly, the more work each computer did, the better the overall speedup was. When the computer steps were 1000, the speedup was very close to the number of computers used. The speedup was measured by the amount of time it would take one machine to generate a certain result as opposed to the number of machines used.

Thirdly, we discovered that when the number of computer steps is high, increasing the number of computers used directly influences the speedup and provides excellent results.

Finally, we formalized the lower boundary of the speedup for each parameter.

The table below summorizes the best speedup results for an array size of 100 000 and comp_steps of 1000.

Number of Computers	Maximum Speedup
2	1.83
10	8.92
20	17.15

8. References

[jas1] “Architecture of Field-Programmable Gate Arrays: The Effect of Logic Block Functionality on Area Efficiency”, J. Rose, R.J. Francis, D. Lewis, P. Chow

[jas2] “High-Quality, Deterministic Parallel Placement for FPGAs on Commodity Hardware”, Altera Corporation, A. Ludwin, V. Betz, K. Padalia, Feb 2008

[jas3] “Strategies for a Massively Parallel Implementation of Simulated Annealing”, F. Baiardi

[jas4] “Parallel Simulated Annealing Algorithms for Cell Placement on Hypercube Multiprocessors”, P. Banerjee, M.H. Jones, J.S. Sargent

[jas5] “Parallel Algorithms for Chip Placement by Simulated Annealing”, F. Darema, S. Kirkpatrick, V. A. Norton.

[jas6] “Placement by Simulated Annealing on a multiprocessor”, TCAD, pp. 534-549, Jul 1987

[jas7] “An evaluation of parallel simulated annealing strategies with application to standard cell placement”, TCAD, vol 16, pp. 398-410, Apr. 1997.

[jas8] “Parallel simulated annealing strategies for VLSI cell placement”, in VLSID, (Bangalore, India), pp. 37-42, 1996.

[Bas_1].”Advanced Computer Architecture and Parallel Processing”, Hesham El-Rewini, Mostafa Abo_El_Barr, John Wiley 2005

[Bas_2]. “Parallel Computer Architecture”, David E.Culler, Jaswinder Pal Singh, Anoop Gupta, Morgan Kaufmann, 2003.

[Bas_3]. "Beginner’s Guide to MPI", Dixie Hisley and Lori Pollock, University of Delaware,August 2006

[Bas_4]. MPICH2: http://www.mcs.anl.gov/research/projects/mpich2/

[Bas_5]. MPI Forum: http://www.mpi-forum.org

[Bas_6]. Running C/C++ Program in parallel using MPI http://www.cs.rpi.edu/~seole/doc/mpi.pdf

[Bas_7]. http://www.nada.kth.se/kurser/kth/2D1263/4_lecture10.pdf

[Bas_8].” An introduction to the Message Passing Interface (MPI) using C “, http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.shtml

[Bas_9]. “MPI Programming Guide”, http://www.nersc.gov/vendor_docs/ibm/pe/am106mst02.html

[Bas_10]. “MPI: A Message-Passing Interface Standard“
http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html

[ramya1]. Performance optimization by interacting netlist transformations and placement Stenz, G.; Riess, B.M.; Rohfleisch, B.; Johannes, F.M.; Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on Volume 19, Issue 3, March 2000 Page(s):350 - 358

[ramya2]. A Gomory-Hu cut tree representation of a netlist partitioning problem Vannelli, A.; Hadley, S.W.; Circuits and Systems, IEEE Transactions on Volume 37, Issue 9, Sept. 1990 Page(s):1133 - 1139

[ramya3]. Fast batch incremental netlist compilation hierarchical schematics Jones, L.G.; Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on Volume 10, Issue 7, July 1991 Page(s):922 - 931

[ramya4]. Netlist Instances Definition Instance Design Netlists Folded.
http://www.economicexpert.com/2a/Netlist.html

[ramya5]. Netlist – Wikipedia. http://en.wikipedia.org/wiki/Netlist

[ramya6]. From C to netlists: hardware engineering for software engineers?
Alston, I. Madahar, B. Syst. Dept., BAE SYSTEMS Adv. Technol. Centre, Chelmsford;

[ramya7]. Zero-Change Netlist Transformations: A New Technique for Placement Benchmarking Kahng, A.B.; Reda, S.; Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on Volume 25, Issue 12, Dec. 2006 Page(s):2806 – 2819

[ramya8]. An efficient eigenvector approach for finding netlist partitions. Hadley, S.W.; Mark, B.L.; Vannelli, A.; Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on Volume 11, Issue 7, July 1992 Page(s):885 - 892

[ramya9]. An efficient eigenvector-node interchange approach for finding netlist partitions. Vannelli, A.; Hadley, S.W.; Mark, B.L.; Custom Integrated Circuits Conference, 1991., Proceedings of the IEEE 1991 12-15 May 1991 Page(s):28.2/1 - 28.2/4

9. Code

The following is code for generating the netlist, performing the simulated annealing algorithm and implementing and measuring the execution time in MPI.

```
#include<stdio.h>
#include<stdlib.h>
#define n 1000
#define s 4000


main()
{

        int i,j,p;
        int input[s];
        int output[n];
        int seed;
        double r;
        long int M;
        int y;
        int z;
        double x;

        FILE *fp;
        fp = fopen("c:\\nlist.txt","w");

        seed = 1000;
        M=999;

        printf("\n\n\n");

/* Initialize to 0 */
        for(i=1;i<=s;i++)
        {
                input[i] = 0;
        /*      printf("Initialize input[%d] = %d\n", i, input[i]);*/
        }
        for(i=1;i<=n;i++)
        {
                output[i] = 0;
/*              printf("Initialise output[%d] = %d\n", i, output[i]);*/
        }
        printf("Logic_in and Logic_out initialized\n");
```

```
srand(seed);
for(i=1;i<=n;++i)
{
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nOutput[%d]\nInput[%d]\t ", i, z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);

}
fprintf(fp,"\n");
for(i=1;i<=(n/2);i++)
{
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nOutput[%d]\nInput[%d]\t", i, z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
```

```
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);
}
fprintf(fp,"\n\n");
for(i=1;i<=(n/5);i++)
{
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nOutput[%d]\nInput[%d]\t", i, z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);
}
fprintf(fp,"\n\n");
for(i=1;i<=(n/10);i++)
{
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nOutput[%d]\nInput[%d]\t", i, z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
```

```
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);
}
fprintf(fp,"\n\n");
for(i=1;i<=(n/100);i++)
{

        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nOutput[%d]\nInput[%d]\t", i, z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);

}
fprintf(fp,"\n\n");
for(i=1;i<=(n/150);i++)
{
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nOutput[%d]\nInput[%d]\t", i, z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
```

```
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);

}
fprintf(fp,"\n\n");
for(i=1;i<=(n/500);i++)
{
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y =(int)x;
        z = y+1;
        fprintf(fp,"\nOutput[%d]\nInput[%d]\t", i, z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nInput[%d]\t", z);
}
fprintf(fp,"\n\n");
for(i=1;i<=(n/1000);i++)
{
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
        fprintf(fp,"\nOutput[%d]\nInput[%d]\t", i, z);
        r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
        x = (r * M);
        y = (int)x;
        z = y+1;
```

```
fprintf(fp,"\nInput[%d]\t", z);
r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
x = (r * M);
y = (int)x;
z = y+1;
fprintf(fp,"\nInput[%d]\t", z);
r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
x = (r * M);
y = (int)x;
z = y+1;
fprintf(fp,"\nInput[%d]\t", z);

r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
x = (r * M);
y = (int)x;
z = y+1;
fprintf(fp,"\nOutput[%d]\nInput[%d]\t", i, z);
r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
x = (r * M);
y = (int)x;
z = y+1;
fprintf(fp,"\nInput[%d]\t", z);
r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
x = (r * M);
y = (int)x;
z = y+1;
fprintf(fp,"\nInput[%d]\t", z);
r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
x = (r * M);
y = (int)x;
z = y+1;
fprintf(fp,"\nInput[%d]\t", z);

r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
x = (r * M);
y = (int)x;
z = y+1;
fprintf(fp,"\nOutput[%d]\nInput[%d]\t", i, z);
r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
x = (r * M);
y = (int)x;
z = y+1;
fprintf(fp,"\nInput[%d]\t", z);
r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
x = (r * M);
y = (int)x;
```

```
                z = y+1;
                fprintf(fp,"\nInput[%d]\t", z);
                r = ((double)rand()/((double)(RAND_MAX)+(double)(1)));
                x = (r * M);
                y = (int)x;
                z = y+1;
                fprintf(fp,"\nInput[%d]\t", z);

        }
fclose(fp);
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#define MASTER 0
MPI_Status status;

 main(int argc, char **argv)
{


  int i,j,k,l, num;

  int tot =500;
  int dim;
  dim=tot*tot;
  int min = 0;
  int mid = tot/2;
  int max = tot;

  int array[tot][tot];
  int array1[tot][tot];
  int array2[tot][tot];
  int array3[tot][tot];
  int array4[tot][tot];
  double start_time_T;   /*Total start time*/
  double finish_time_T;  /*Total finish time*/
  double start_time_B;   /*Broadcasting start time*/
  double finish_time_B;  /*Broadcasting finish time*/
  double B1,B2,U1,U2,U3,U4,U11,U22,U33,U44;
  double start_time_G;   /*Gathering start time*/
  double finish_time_G;  /*Gathering finish time*/
  double start_time_U2;   /*Updating start time */
  double finish_time_U2; /*Updating finish time*/
  double start_time_U1;
  double finish_time_U1;
  double start_time_U3;
  double finish_time_U3;
  double start_time_U4;
  double finish_time_U4;
  double start_time_U;
  double finish_time_U;

  int size,myid,dest,mtype,offset,p,source,nprocess;
```

```
/***************start MPI*************************/
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myid);
MPI_Comm_size(MPI_COMM_WORLD, &size);
nprocess=size-1;


/*****************************Master Task************************/
start_time_T=MPI_Wtime(); /*total start time*/

if (myid==MASTER)



  {


      //initializing the array with values

      num = 0;
      for (i=0; i<tot; i++)
      {
            for (j=0; j<tot; j++)
            {
                  array[i][j] = num;
                  num = num + 1;
                  //                printf("%d\t", array[i][j]);
            }
            //      printf("\n");
      }

      //      printf("\n\n");


      for (i=0; i<tot; i++)
      {
            for (j=0; j<tot; j++)
            {
                  array1[i][j] = array[i][j];
                  array2[i][j] = array[i][j];
                  array3[i][j] = array[i][j];
                  array4[i][j] = array[i][j];
                  //printf("%d\t", array[i][j]);
            }
            //printf("\n");
      }
```

```
        start_time_B=MPI_Wtime();     /*Broadcasting start time*/


        MPI_Bcast(&array[0][0],dim,MPI_INT,0,MPI_COMM_WORLD);
        finish_time_B=MPI_Wtime();



        printf("Sending   time=%f\n",finish_time_B-start_time_B);

        B1= finish_time_B-start_time_B;


MPI_Recv(&array2[0][0],dim,MPI_INT,1,80,MPI_COMM_WORLD,&status);


  }

   else //slaves
  {
   start_time_B=MPI_Wtime();

        MPI_Bcast(&array[0][0],dim,MPI_INT,0,MPI_COMM_WORLD);
        //          printf("Received Array\n");


          finish_time_B=MPI_Wtime();   /*boradcasting finish time*/
printf("Receiving time=%f\n", finish_time_B-start_time_B);




        for (i=0;i<tot;i++)
        {
         //for (j=0;j<tot;j++)
                    //        printf ("%d ",array[i][j]);
                    //        printf("\n");
        }



  }
start_time_U=MPI_Wtime();    /******Total Updating time
```

```
/************** computer 1 Send*****************/
//        printf("computer 1 sending");

          if (myid ==1)
            {


//computer1 broadcasts a part of the array to all the other computers


for (i=min; i<mid; i++)
   {
     for (j=min; j<mid; j++)
          {
            array1[i][j] = array[i][j];

            //          printf("%d\t", array1[i][j]);
          }
    //   printf("\n");

   }
// printf("\n\n");



 start_time_U1=MPI_Wtime();    /*start updating time*/


 MPI_Send(&array1[0][0],dim/4,MPI_INT,2,1,MPI_COMM_WORLD);
 MPI_Send(&array1[0][0],dim/4,MPI_INT,3,1,MPI_COMM_WORLD);
 MPI_Send(&array1[0][0],dim/4,MPI_INT,4,1,MPI_COMM_WORLD);

 MPI_Recv(&array2[0][0],dim/4,MPI_INT,2,2,MPI_COMM_WORLD,&status);
 MPI_Recv(&array3[0][0],dim/4,MPI_INT,3,3,MPI_COMM_WORLD,&status);
 MPI_Recv(&array4[0][0],dim/4,MPI_INT,4,4,MPI_COMM_WORLD,&status);

 //  printf("U1=%f\n",U1);

MPI_Send(&array1[0][0],dim,MPI_INT,0,80,MPI_COMM_WORLD);
 finish_time_U1=MPI_Wtime();   /*finish updating time*/
 U1=start_time_U1-finish_time_U1;

  printf("Updating time 1=%f\n",finish_time_U1 - start_time_U1);

          }
```

```
        else if (myid==2)
          {

  /************* computer 2 Send**************/
            //    printf("computer 2 sending\n");

 //computer2 broadcasts a part of the array to all the other computers
 for (i=min; i<mid; i++)
  {
    for (j=mid; j<max; j++)
        {
          array2[i][j] = array[i][j];

          // printf("%d\t", array2[i][j]);
        }
    // printf("\n");
  }
//  printf("\n\n");

start_time_U2=MPI_Wtime();    /*start updating time*/

MPI_Send(&array2[0][0],dim/4,MPI_INT,1,2,MPI_COMM_WORLD);
 MPI_Send(&array2[0][0],dim/4,MPI_INT,3,2,MPI_COMM_WORLD);
 MPI_Send(&array2[0][0],dim/4,MPI_INT,4,2,MPI_COMM_WORLD);

 MPI_Recv(&array1[0][0],dim/4,MPI_INT,1,1,MPI_COMM_WORLD,&status);

 MPI_Recv(&array3[0][0],dim/4,MPI_INT,3,3,MPI_COMM_WORLD,&status);
 MPI_Recv(&array4[0][0],dim/4,MPI_INT,4,4,MPI_COMM_WORLD,&status);
 finish_time_U2=MPI_Wtime();

 finish_time_U2=MPI_Wtime();   /*finish updating time*/

 //U2=start_time_U-finish_time_U;
 //printf("updating time2=%f\n",U2+U22);

 printf("Updating time 2=%f\n",finish_time_U2 - start_time_U2);
          }

        else if (myid==3)
          {

 /***************computer 3 Send*********************/
```

```
            // printf("computer 3 sending\n");
 //computer3 broadcasts a part of the array to all the other computers

 for (i=mid; i<max; i++)
  {
    for (j=min; j<mid; j++)
         {
           array3[i][j] = array[i][j];

           //        printf("%d\t", array[i][j]);
         }
  //  printf("\n");
  }
 // printf("\n\n");

 start_time_U3=MPI_Wtime();

MPI_Send(&array3[0][0],dim/4,MPI_INT,1,3,MPI_COMM_WORLD);
 MPI_Send(&array3[0][0],dim/4,MPI_INT,2,3,MPI_COMM_WORLD);
 MPI_Send(&array3[0][0],dim/4,MPI_INT,4,3,MPI_COMM_WORLD);
 MPI_Recv(&array1[0][0],dim/4,MPI_INT,1,1,MPI_COMM_WORLD,&status);
 MPI_Recv(&array2[0][0],dim/4,MPI_INT,2,2,MPI_COMM_WORLD,&status);
 MPI_Recv(&array4[0][0],dim/4,MPI_INT,4,4,MPI_COMM_WORLD,&status);

finish_time_U3=MPI_Wtime();
U3=finish_time_U3-start_time_U3;
printf("Updating time 3=%f\n",U3);

          }
         else if (myid==4)
          {

 /*************** computer 4 Send**********************/

            // printf("computer 4 sending\n");

//computer4 broadcasts a part of the array to all the other computer
 for (i=mid; i<max; i++)
  {
    for (j=mid; j<max; j++)
         {
          array4[i][j] = array[i][j];

          //  printf("%d\t", array4[i][j]);
         }
    // printf("\n");
```

```
  }
 // printf("\n\n");

 start_time_U4=MPI_Wtime();

MPI_Send(&array4[0][0],dim/4,MPI_INT,1,4,MPI_COMM_WORLD);
 MPI_Send(&array4[0][0],dim/4,MPI_INT,2,4,MPI_COMM_WORLD);
 MPI_Send(&array4[0][0],dim/4,MPI_INT,3,4,MPI_COMM_WORLD);

 MPI_Recv(&array1[0][0],dim/4,MPI_INT,1,1,MPI_COMM_WORLD,&status);
 MPI_Recv(&array2[0][0],dim/4,MPI_INT,2,2,MPI_COMM_WORLD,&status);
 MPI_Recv(&array3[0][0],dim/4,MPI_INT,3,3,MPI_COMM_WORLD,&status);
 finish_time_U4=MPI_Wtime();


 U4=finish_time_U4-start_time_U4;
 printf("Updating time 4=%f\n",U4);

          }
         finish_time_U=MPI_Wtime();


/*****************************************************/

         if (myid==1)
           {



/*****************computer 1 Recev ******************/

            //start_time_U=MPI_Wtime();

   //    MPI_Recv(&array2[0][0],2500,MPI_INT,2,2,MPI_COMM_WORLD,&status);
   //MPI_Recv(&array3[0][0],2500,MPI_INT,3,3,MPI_COMM_WORLD,&status);
   //MPI_Recv(&array4[0][0],2500,MPI_INT,4,4,MPI_COMM_WORLD,&status);

   //finish_time_U=MPI_Wtime();

   //U11=start_time_U-finish_time_U;
   //printf("U11=%f\n",U11);

          }

         else if (myid==2)
           {
```

```
 /***************computer 2 Recev ******************/
            //start_time_U=MPI_Wtime();

            //
MPI_Recv(&array1[0][0],2500,MPI_INT,1,1,MPI_COMM_WORLD,&status);

            //
MPI_Recv(&array3[0][0],2500,MPI_INT,3,3,MPI_COMM_WORLD,&status);

//MPI_Recv(&array4[0][0],2500,MPI_INT,4,4,MPI_COMM_WORLD,&status);
            //finish_time_U=MPI_Wtime();

            //U22=start_time_U-finish_time_U;
  //  printf("U22=%f\n",U22);

           }

          if (myid==3)
            {

  /***************computer 3 Recev ******************/
            //start_time_U=MPI_Wtime();

            //
MPI_Recv(&array1[0][0],2500,MPI_INT,1,1,MPI_COMM_WORLD,&status);

//MPI_Recv(&array2[0][0],2500,MPI_INT,2,2,MPI_COMM_WORLD,&status);
            //
MPI_Recv(&array4[0][0],2500,MPI_INT,4,4,MPI_COMM_WORLD,&status);
            //finish_time_U=MPI_Wtime();
            //U33=start_time_U-finish_time_U;
            //printf("U33=%f\n",U33);


           }

          else if (myid ==4)
            {

  /*************** computer 4 Recv ******************/
            //start_time_U=MPI_Wtime();

            //
MPI_Recv(&array1[0][0],2500,MPI_INT,1,1,MPI_COMM_WORLD,&status);
```

```
//MPI_Recv(&array2[0][0],2500,MPI_INT,2,2,MPI_COMM_WORLD,&status);

//MPI_Recv(&array3[0][0],2500,MPI_INT,3,3,MPI_COMM_WORLD,&status);
              //   finish_time_U=MPI_Wtime();
              // U44=start_time_U-finish_time_U;
              //printf("U44=%f\n",U44);



            }



 //MPI_Bcast(&offset,64,MPI_INT,0,MPI_COMM_WORLD);




 /*wait for result from all processors*/
 //   for (source=1; source=nprocess;source++);




 {

   //    MPI_Recv(&offset,64,MPI_INT,source,2,MPI_COMM_WORLD,&status);
 //  MPI_Recv(array,64,MPI_INT,source,2,MPI_COMM_WORLD,&status);
   //    printf("source=%d\n",p);

 }




 //now all the computers should have an updated version of the array
 for (i=0; i<tot; i++)
   {
     for (j=0; j<tot; j++)
          {

            printf("%d\t", array[i][j]);
          }
```

```
    printf("\n");

  }
*/
        /*printf("Updating time 1=%f\n",U1+U11);

        printf("Updating time 2=%f\n",U2+U22);

        printf("Updating time 3=%f\n",U3+U33);

        printf("Total Updating Time=%f\n",U1+U2+U3+U4+U11+U22+U33+U44);*/



 finish_time_T=MPI_Wtime();    /*Total finish time*/
 // printf("Total time=%f\n",finish_time_T- start_time_T);
 //  printf("source=%d\n",p);
 //printf("nprocess=%d\n",nprocess);
 //ends MPI

 //    printf("Total=%f\n",finish_time_U-start_time_U);

 if (myid==0)
   {
    printf("Total Time=%f\n", finish_time_T - start_time_T);
    //       printf("Total Updating Time=%f\n",finish_time_U- start_time_U);

   }


MPI_Finalize();
 //  printf("Updating time 1=%f\n",U1+U11);
 //printf("Updating time 2=%f\n",U2+U22);
 //printf("Updating time 3=%f\n",U3+U33);
 //printf("Updating time 4=%f\n",U4+U44);
 //printf("Total Updating Time=%f\n",U1+U2+U3+U4+U11+U22+U33+U44);


 // printf("Total Time=%f\n",finish_time_T - start_time_T);

//  printf("Broadcasting time=%f\n",finish_time_B - start_time_B);

}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <time.h>
#include <sys/time.h>

int print_initial_array = 0;  //do you want to print the input array to
the screen?

int ran_num = 251;

int tot_blocks = 10000;
int max = 10;
int mid = 5;

int mid_size = 5;

int computers = 10;
int comp_steps = 50;

int outputs  = 1;
int line_size = 1;


int connectivity = 50;

int swap_steps = 5;    //swap steps


int array[10000][8];
int array1[10000][8];
int array2[10000][8];
int array3[10000][8];
int array4[10000][8];


int old_array[10000][8];
int para_array[10000][8];
int old_array1[10000][8];

int temp_steps = 20;  //temperature steps
int init_temp = 100;
float temp_rate = 0.9;

float time_update = 0.4;



int calc_cost(void);
void printing();
void switch_blocks(int outputs);
```

```
main()
{

  srand((int)(time(NULL)));
  time_update = time_update + (float)(rand()%40) / (float)100;
  printf("%f time update\n", time_update);
  clock_t testtime1, testtime2;

  testtime1 = clock();




      time_t time1, time2;
  int tot_time;
  time1 = time(NULL);


  clock_t tpart1_start, tpart1_stop, tpart1_tot;
  clock_t tpart2_start, tpart2_stop, tpart2_tot;
  clock_t tpart_para_start, tpart_para_stop;
  float tpart_para_tot;
  float tpart_para_div;

      float tpart_para_tot_sec = 0;
  tpart1_start = clock();







  //setting temperature
  int p,ii;
  int temp[temp_steps];
  temp[0] = init_temp;

  for (ii=1; ii<temp_steps; ii++)
    {
      temp[ii] = (int) (temp[ii-1] * temp_rate);

      //printf("%d\n", temp[ii]);
    }




  int i,j,k;

  double n = 2;
  char line[10];
  char c, a, b, d;
```

```
 int number;


 outputs = 3;



 FILE *myfile;
 if ((myfile = fopen ( "placement.txt", "rt")) == NULL)
   {
     printf("ERROR: COULD NOT OPEN FILE: placement.txt\n");
   }
 else
   {
     //printf("opened file\n");
   }

 FILE *outputfile;
 if ((outputfile = fopen ( "outputfile.txt", "w")) == NULL)
   {
     printf("ERROR: COULD NOT OPEN FILE: outputfile.txt\n");
   }
 else
   {
     //printf("opened file\n");
   }


 FILE *updatefile;
 if ((updatefile = fopen ( "updatefile.txt", "w")) == NULL)
   {
     printf("ERROR: COULD NOT OPEN FILE: updatefile.txt\n");
   }
 else
   {
     //printf("opened file: updatefile\n");
   }



 FILE *analysisfile;
 if ((analysisfile = fopen ( "analysisfile.txt", "a")) == NULL)
   {
     printf("ERROR: COULD NOT OPEN FILE: analysisfile.txt\n");
   }
 else
   {
     //printf("opened file: analysis.txt\n");
   }


int x_size;
int y_size;
```

```
  j=0;


  //aquireing the FPGA size
  while ( (c=getc(myfile)) != '\n')
    {
      x_size = atoi(&c);
    }

  while ( (c=getc(myfile)) != '\n')
    {
      y_size = atoi(&c);
    }

  //printf("%d, %d\n", x_size, y_size);


  //tot_blocks = x_size * y_size;

  line_size = outputs*2 +2;// this needs to be read from the file!!

//  int array[tot_blocks][8];


  //reading the numbers in from file
  for (i = 0; i< tot_blocks; i++)
    {
      //printf("%d: ", i);

      j= 0;

      for (k = 0; k<8; k++)
      {
      fscanf(myfile, "%d", &array[i][k]);
      if (print_initial_array == 1)
      {
            printf("%d, ", array[i][k]);
      }

      }

      if (print_initial_array == 1)
      {
            printf("\n");
      }

    }

  //printf("\n");
```

```
  int w;

  //printing();


  fclose(myfile); // ******closing the file*******



  //************************MAIN ALGORITHM
 //PART II - placement ****************************** ONE ARRAY
  int cost;
  int t;
  int new_cost;
  int e,g;
  int tryme;
  int updates;
  int pp, ps, para_cost;


  int initial_cost;
  cost = calc_cost();
  initial_cost = cost;
  printf("initial cost: %d\n", cost);

  int new_cost1, new_cost2, new_cost3, new_cost4;


  tpart1_stop = clock();
  tpart1_tot = tpart1_start - tpart1_stop;


  for (g=0; g<temp_steps; g++)
    {
      updates = 0;

      for (e = 0; e < swap_steps; e++)
      {
        //switch_blocks(outputs);  //does the switch multiple times
            new_cost = calc_cost();
        old_array = array;

        tpart_para_start = clock();
            for (pp=0; pp< computers; pp++)
//COMPUTER PARALLELIZATION
            {

          for (ps=0; ps< comp_steps; ps++)
          {
            old_array1 = array;
                    switch_blocks(outputs);  //does the switch multiple
times
                    new_cost1 = calc_cost();
                    //printf("step:%d:: %d,%d::   %d, %d,
%d\n",e,pp,ps, cost, new_cost, new_cost1);
                    if (new_cost1 < new_cost)
                    {
```

```
                        new_cost = new_cost1;
                    }
                    else
                    {
                        array = old_array1;
                    }
            }
            //printf("\n");
        }
            tpart_para_stop = clock();
        //printf("start: %f, stop: %f  ---> %f\n",
(float)(tpart_para_start), (float)(tpart_para_stop), tpart_para_tot);
        tpart_para_tot = tpart_para_tot +  (float)(tpart_para_stop -
tpart_para_start);

            if (new_cost < cost)    //accept lower cost  ****
//DECISION*************************************************8
                {
                  cost = new_cost;
                  //printf("%d, p: %d: new_cost:%d\n",g, pp, cost);
                  updates++;
                }
            else
                {
                  tryme = rand()%100 +1;
              //printf("temp[%d] > %d\n", temp[g], tryme);

                  if (temp[g] < tryme)    // reject higher cost ****
                  {
                    array = old_array;
                    //printf("%d,p: %d: reject higher cost: %d\n",g,
pp, cost);
                  }
                  else        //accept hight cost  *****
                  {
                    cost = new_cost;
                  //printf("%d, p: %d: accept higher cost: %d\n",
g,pp,cost);
                    updates++;
              }
                }
                //printf("p: %d, cost: %d\n",pp, cost);
      }
      //printing();
      //printf("t: %d, cost: %d\n", g, cost);
      printf("***t: %d, updates: %d, cost: %d\n", g, updates, cost);
   }


  fprintf(updatefile, "%d\t%d\n", g, updates);
  fprintf(outputfile, "%d\t%d\t%d\t%d\n",g,e, cost, updates);



 //PART III - placement ****************************** FOUR ARRAYS
```

```
  //tpart_para_tot = tpart_para_start - tpart_para_stop;




  time2 = time(NULL);
  tot_time = time2 - time1;
  testtime2 = clock();

//  double time4;

//    time4 = time(NULL);
      printf("%d\n", time2);

  //printing to screen;
  printf("initial cost: %d\nfinal cost: %d   <-------------\ndiff:
%d\npercentage improvement:%d\n", initial_cost, cost, initial_cost -
cost, (initial_cost - cost)/initial_cost);
  printf("total time (time_t): %d\n", tot_time);



  //printing to file;
  fprintf(outputfile, "initial cost: %d\nfinal cost: %d\ndiff:
%d\npercentage improvement:%d\n", initial_cost, cost, initial_cost -
cost,( (double)((initial_cost - cost)/initial_cost))*100  );
  fprintf(outputfile, "total blocks: %d\n", tot_blocks);
  fprintf(outputfile, "swap steps: %d\n", swap_steps);
  fprintf(outputfile, "temp steps: %d\n", temp_steps);
  fprintf(outputfile, "total time: %d\n", tot_time);

  fprintf(outputfile, "tot_blocks: %d\n", tot_blocks);
  fprintf(outputfile, "connectivity: %d\n", connectivity);

  fprintf(outputfile, "ran_num: %d\n", ran_num);




//TIME CALCULATIONS

  float tot_time_sec, time_parallel_1, time_parallel_2,
time_update_tot;
```

```
  tot_time_sec = (float)(testtime2 - testtime1) /
(float)(CLOCKS_PER_SEC) ;

  tpart_para_tot_sec = (float)(tpart_para_tot) /
(float)(CLOCKS_PER_SEC);
  tpart_para_div = tpart_para_tot / computers / CLOCKS_PER_SEC;

  time_update_tot = time_update * temp_steps * swap_steps;

  time_parallel_1 = tot_time_sec - tpart_para_tot_sec;
  time_parallel_2 = time_parallel_1  + tpart_para_div +
time_update_tot;

  printf("total time (from clocks):  %f  <------\n", tot_time_sec);

  printf("tpart_para_tot: %f\n", tpart_para_tot);
  printf("tpart_para_tot_sec  : %f\n", tpart_para_tot_sec);
  printf("tpart_para_div  : %f\n", tpart_para_div   );
  printf("time_update_tot  : %f\n", time_update_tot);
  printf("time_parallel_1  : %f\n", time_parallel_1   );
  printf("time_parallel_2  : %f  <------\n", time_parallel_2   );



  fprintf(outputfile, "\n\ntot_blocks, computers, comp_steps,
temp_steps, temp_rate, swap_steps, connectivity, tot_time_sec,
time_parallel_2, cost\n%d\t%d\t%d\t%d\t%f%d\t%d\t%f\t%f\t%d\t%d\t%f\n",
tot_blocks, computers, comp_steps, temp_steps, temp_rate, swap_steps,
connectivity, tot_time_sec, time_parallel_2, initial_cost, cost,
time_update);


  fprintf(analysisfile,
"%d\t%d\t%d\t%d\t%f%d\t%d\t%f\t%f\t%d\t%d\t%f\t%f\n", tot_blocks,
computers, comp_steps, temp_steps, temp_rate, swap_steps, connectivity,
tot_time_sec, time_parallel_2, initial_cost, cost, time_update,
tpart_para_tot_sec);

  fclose(outputfile);
  fclose(analysisfile);

}


//************************END MAIN*******************************
```

```
int calc_cost()
{
 //calculating cost
  int cost = 0;

  int minx,maxx, miny, maxy,i, k, j;
  int tot_cost = 0;

  for (i = 0; i < tot_blocks; i++)
    {
      j = 0;

      minx = maxx = array[i][j];
      miny = maxy = array[i][j+1];

      // printf("%d, %d\n", array[i][j], array[i][j+1]);

      for(k=0; k < (outputs); k++)
      {
        j = j+2;

        if ((array[i][j] != 0) && (array[i][j+1] != 0))
          {

            if (array[i][j] < minx)
            {
              minx = array[i][j];

            }
            if (array[i][j] > maxx)
            {
              maxx = array[i][j];
              //printf("%d:%d\n",j, maxx);

            }
            if (array[i][j+1] < miny)
            {
              miny = array[i][j+1];
            }
            if (array[i][j+1] > maxy)
            {
              maxy = array[i][j+1];
            }
          }

      }

      //printf("%d: %d, %d, %d, %d --- ",i, minx, maxx, miny, maxy);
      cost = (maxx - minx) + (maxy - miny);
      tot_cost = tot_cost + cost;

      //printf("%d: line_cost=%d\n",i, cost);

    }
  //printf("\n");
```

```
  return(tot_cost);


}
```

```
void printing()
{
  int w,j;

  //printing the array to the screen
  for (w = 0; w < tot_blocks; w++)
    {
      printf("%d: ",w);

      for (j = 0; j < line_size; j++)
      {

        printf("%d ", array[w][j]);
      }
      printf("\n");

    }
}
```

```
//switching blocks function


void switch_blocks( int outputs)
{

  //switching two blocks

  int from, to;
  int tempx,tempy;
  int s, t;
  int i, j, k;
  int temp, f;


  //mod to use the range function!!!! *****
  from = (rand()%tot_blocks) ;
  to = (rand()%tot_blocks);
```

```
  //printf("from:%d, to:%d\n", from,to);

  for (f=2; f<8; f++)
    {
      //printf("%d, ",f);


      temp=array[from][f];
      //printf("%d\n", array[to][f]);

      array[from][f] = array[to][f];
      array[to][f]=temp;
    }



  //temp1 = array[from][0];
  //temp2 = array[from][1];
  //array[from][0] = array[to][0];
  //array[from][1] = array[to][1];
  //array[to][0] = tempx;
  //array[to][1] = tempy;

  for (i = 0; i < tot_blocks; i++)
    {
      j = 2;

      // printf("%d, %d\n", array[i][j], array[i][j+1]);

      for(k=0; k < (outputs); k++)
      {
        //printf("i:%d, j=%d, k=%d\n", i, j, k);

        if ((array[i][j] == array[from][0]) && (array[i][j+1] ==
array[from][1]))
          {
            array[i][j] = array[to][0];
            array[i][j+1] = array[to][1];
            //printf("i = %d, here: (%d, %d) = (%d, %d)\n",i,
array[i][j], array[i][j+1], array[from][0], array[from][1]);

          }

        else if ((array[i][j] == array[to][0]) && (array[i][j+1] ==
array[to][1]))
          {
            array[i][j] = array[from][0];
            array[i][j+1] = array[from][1];
          }

        j = j+2;

      }
    }
}
```