CHAPTER 1

INTRODUCTION

1.1 OVERVIEW:

Sequential algorithms and single core processors are not convenient these days because of their ineffectiveness in solving a problem within the allocated time and high resource usage. Because of the several limitations of predominant programming paradigm, a better programming construct is needed that would help solving today’s problems. There exist lots of parallel architecture for which serial code is inefficient. For e.g: Grid computing, Multicore processing, Multi processing, Multicore MPSoC, Super Computer, GPU, Cluster, Cloud Computing, etc… In nature everything is parallel, uncountable number of processes happen simultaneously at a time. Definitely implementing this methodology in computer architecture has a huge benefit. Using parallel computing technique, simultaneous implementation of many independent algorithms can be achieved. The speed and the amount of the operations that can be performed simultaneously are tremendously high. Effective usage of available resources and minimized time spent in execution are some of the advantages of parallelization.[1]

The only way of increasing the performance of a sequential code is to implement costlier hardware's such as high-speed processor. Even if we invest high performing hardware’s, most of the times these hardware’s remain idle because of the traditional top to bottom problem solving approach. The load, which a single core can bear, has reached its limit. And it is a bare necessity to use a Multicore processor when solving a complex problem. One way of conquering the ineffectiveness of sequential programming model is to change the fundamental problem solving technique by parallelism. Technically Sequential code needs to be converted into parallel code. New compilers, software’s and most importantly new ideas are required to convert sequential code to parallel code. A parallelizing software model which uses the available hardware resources very effectively, is necessary for new generation CPU’s.

1.2 PROBLEM STATEMENT

Automatic parallelization, also auto parallelization, autoparallelization, or parallelization refers to converting sequential code into multi-threaded or vectorized (or even both) code in order to utilize multiple processors simultaneously in a shared-memory multiprocessor (SMP) machine. The goal of automatic parallelization is to relieve programmers from the tedious and error-prone manual parallelization process. Though the quality of automatic parallelization has improved in the past several decades, fully automatic parallelization of sequential programs by compilers remains a grand challenge due to its need for complex program analysis and the unknown factors (such as input data range) during compilation.[23]

The programming control structures on which autoparallelization places the most focus are loops, because, in general, most of the execution time of a program takes place inside some form of loop. There are two main approaches to parallelization of loops: pipelined multi-threading and cyclic multi-threading.[24]

For example, consider a loop that on each iteration applies a hundred operations, runs for a thousand iterations. This can be thought of as a grid of 100 columns by 1000 rows, a total of 100,000 operations. Cyclic multi-threading assigns each row to a different thread. Pipelined multi-threading assigns each column to a different thread.

1.3 LITERATURE SURVEY

i) CETUS:

Cetus is a compiler infrastructure for the source-to-source transformation of software programs. This project is developed by Purdue University. Cetus is written in Java. It provides basic infrastructure for write automatic parallelization tools or compilers. The basic parallelizing techniques Cetus currently implements are privatization, reduction variables recognition and induction variable substitution.

ii) SUIF compiler

SUIF (Stanford University Intermediate Format) is a free infrastructure designed to support collaborative research in optimizing and parallelizing compilers. SUIF is a fully functional compiler that takes both Fortran and C as input languages. The parallelized code is output as an SPMD (Single Program Multiple Data) parallel C version of the program that can be compiled by native C compilers on a variety of architectures.

1.4 APPLICATION

1. Relieve programmers from the tedious and error-prone manual parallelization process.

2. Already Parallelized problems serve as the template for future encounters of similar type of problems, thus code reusability can be achieved.

3. Save time and/or money: In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel computers can be built from cheap, commodity components.

4. Solve larger problems: Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.

5. Provide concurrency: A single compute resource can only do one thing at a time. Multiple computing resources can be doing many things simultaneously.

CHAPTER 2

CHALLENGES IN PARALLEL-PROGRAMMING

2.1 OVERVIEW

When trying to convert a sequential code, many difficulties pop up from nowhere. When running sequential codes, one could accurately tell where the control was, at a given time. But once it is parallelized, the control simultaneously exists in many places. It is very difficult to trace the program or even to debug the program, when multiple instances run at the same time it is even harder to manage things. When converting a sequential code to parallel code, along with the speed, the complexity of the program also increases. This results in the increased overall human time and effort spent on a problem. In addition, since there is no standard methodology to write a parallel program yet, the development process will be cumbersome. With so many threading libraries floating around the net, one could be easily confused when selecting the suitable parallelizing technique.

There are many methods/libraries available, which can be used to parallelize the problem (parallel programming standards).[2]

1) Thread libraries like win32 API/POSIX.

POSIX threads or Pthreads is a POSIX standard for threads.

2) Compiler Directives like OpenMP, OpenCL

Which use shared memory programming technique, where all the cores of a CPU (or GPU) are effectively used.

3) Message passing libraries (MPI), Distributed memory programming

In this type, multiple PC's are connected together and a task is divided among them.

Each of these libraries are unique in their ways. However, none of them have automatic parallelizing techniques implemented nor they have a sophisticated development environment. Sometimes even initializing a thread with selected parameters may produce a starting delay. An ideal developing environment[1] would be the one which emphasize automatic parallelization, supports source-to-source transformation, is user oriented and easy to handle, and provides the most important parallelization passes as well as the underlying enabling techniques.

2.2 PROFILING SEQUENTIAL CODE TO IDENTIFY HOTSPOTS BOTTLENECKS

Not every single line of the sequential code can be executed in parallel. Depending on the hardware resource availability and the aim of the program, the scope of parallelization changes. The speed and the amount of the operations that can be performed simultaneously depend on the number of processors and its architecture. While optimizing a code, the extreme hotspots will be the blocks of code that are executed repeatedly [3]. These typically occur within loops, especially nested loops. In most of the practical situations, loops are the place where most of the program execution time is spent. Therefore, loops are considered as the optimizing hotspots, If loops are properly optimized, huge performance gain would be achieved. Further, if the loops are parallelized, the performance gain would be significantly huge. Therefore, it can be said that the loops are the good candidates for parallelization. However if a loop has an irresolvable dependency related to parallel execution, it must be left unmodified. If there are more number of lines of code which cannot be parallelized, then the overall performance would be less. No program can run more quickly than the longest chain of dependent calculations (known as the critical path) [4]. A popular approach to the decomposition problem is to require programmers to perform the decomposition analysis themselves, and to communicate that information to the compiler using language extensions [5]. In a large program, analyzing each and every section of the program and finding out parallelizable lines of code would be a tedious job. One solution to this problem would be Time segmenting the loops/functions. In this method, a part of the program is time segmented and the amount of time spent there is calculated, by analyzing the complete overview of time segmented regions, one can get the bottlenecks of the program and resolve it suitably.

```
time_t start,end

…

time(&start)

…
```

A loop or section

…

time(&end)

Print End-Start time

Time segmentation method helps to find the hotspots, but this results in the addition of some lines of code to the original source code, managing the results and analyzing each and every result would be again time consuming. One more way is to use the call graph profiling functions such as gprof and gcov.[3] These built in linux tools provide an immense statistics of a program such as total calls to a function , the amount of time spent in each routine.etc... Analyzing the call graph, a programmer can easily find out the bottlenecks of a program without any additional effort. Once the time consuming hotspots are found out, first they are optimized and if possible parallelized. Optimization is very different from parallelizing, trimming down or simplifying a line(s) of code is called optimization.

E.g. 1) the expression (x*4)/(24*y) can be optimized as x/(6*y).

2) a=b ; c=a; can be replaced by a single statement, c=b;

3) Function inlining also increases the effectiveness of the code.

Moreover, if optimizations like these are done inside a loop, the performance gain would be large. gcc provides different levels(-O1,-O2,-O3) of automatic optimizations[3], this can be enabled by passing -O as a flag.

2.3 PARALLELIZING

One approach of implementing parallelization would be thinking of parallel solution to every problem we come across. In this type of approach, the tasks get parallelized in the step of developing an algorithm itself. This type of algorithm development is considered difficult, because this does not follow the traditional top to bottom problem solving approach. And also the new solutions would be more oriented towards programming instead of simple mathematical steps. One more approach would be parallelizing already written sequential code. This can be accomplished by manually rewriting the code or letting a compiler to do the job (A specific kind of hypothetical compiler, which converts sequential code to parallel code). Algorithm design is a crucial part of the development, in many cases, the best serial algorithm can be easily parallelized. While in other cases, a fundamentally different algorithm will be needed. When manually converting the code, the success of writing thorough transformation would require complete understanding of the previously written code with different combinations that exist.

CHAPTER 3

PARLLELIZATION

3.1 OVERVIEW:

Parallel computing is a form of computation in which many calculations are carried out simultaneously[7], operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). There are several different forms of parallel computing: bit-level, instruction level, data, and task parallelism. Parallelism has been employed for many years, mainly in high-performance computing, but interest in it has grown lately due to the physical constraints preventing frequency scaling. As power consumption (and consequently heat generation) by computers has become a concern in recent years [8], parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multicore processors.[10]

Parallel computers can be roughly classified according to the level at which the hardware supports parallelism, with multi-core and multi-processor computers having multiple processing elements within a single machine, while clusters, MPPs, and grids use multiple computers to work on the same task. Specialized parallel computer architectures are sometimes used alongside traditional processors, for accelerating specific tasks.

Parallel computer programs are more difficult to write than sequential ones [11], because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronization between the different subtasks are typically some of the greatest obstacles to getting good parallel program performance.

The maximum possible speed-up of a program as a result of parallelization is known as Amdahl's law.

Traditionally, computer software has been written for serial computation. To solve a problem, an algorithm is constructed and implemented as a serial stream of instructions.

These instructions are executed on a central processing unit on one computer. Only one instruction may execute at a time—after that instruction is finished, the next is executed.

Parallel computing, on the other hand, uses multiple processing elements simultaneously to solve a problem. This is accomplished by breaking the problem into independent parts so that each processing element can execute its part of the algorithm simultaneously with the others. The processing elements can be diverse and include resources such as a single computer with multiple processors, several networked computers, specialized hardware, or any combination of the above.[12]

Frequency scaling was the dominant reason for improvements in computer performance from the mid-1980s until 2004. The runtime of a program is equal to the number of instructions multiplied by the average time per instruction. Maintaining everything else constant, increasing the clock frequency decreases the average time it takes to execute an instruction. An increase in frequency thus decreases runtime for all compute-bound programs.[13]

However, power consumption by a chip is given by the equation $P = C \times V^2 \times F$, where P is power, C is the capacitance being switched per clock cycle (proportional to the number of transistors whose inputs change), V is voltage, and F is the processor frequency (cycles per second).[14] Increases in frequency increase the amount of power used in a processor. Increasing processor power consumption led ultimately to Intel's May 2004 cancellation of its Tejas and Jayhawk processors, which is generally cited as the end of frequency scaling as the dominant computer architecture paradigm.

Moore's Law is the empirical observation that transistor density in a microprocessor doubles every 18 to 24 months. Despite power consumption issues, and repeated predictions of its end, Moore's law is still in effect. With the end of frequency scaling, these additional transistors (which are no longer used for frequency scaling) can be used to add extra hardware for parallel computing.

3.2 DEPENDENCIES

Understanding data dependencies is fundamental in implementing parallel algorithms. No program can run more quickly than the longest chain of dependent calculations (known as the critical path), since calculations that depend upon prior calculations in the chain must be executed in order. However, most algorithms do not consist of just a long chain of dependent calculations; there are usually opportunities to execute independent calculations in parallel.

Let Pi and Pj be two program segments. Bernstein's conditions describe when the two are independent and can be executed in parallel. For Pi, let Ii be all of the input variables and Oi the output variables, and likewise for Pj. P i and Pj are independent if they satisfy

- $I_j \cap O_i = \varnothing,$
- $I_i \cap O_j = \varnothing,$
- $O_i \cap O_j = \varnothing.$

Violation of the first condition introduces a flow dependency, corresponding to the first segment producing a result used by the second segment. The second condition represents an anti-dependency, when the second segment (Pj) produces a variable needed by the first segment (Pi). The third and final condition represents an output dependency: When two segments write to the same location, the result comes from the logically last executed segment.

3.3 RACE CONDITIONS,MUTUAL EXCULSION,SYNCHRONIZATION and PARALLEL SLOWDOWN

Subtasks in a parallel program are often called threads. Some parallel computer architectures use smaller, lightweight versions of threads known as fibers, while others use bigger versions known as processes. However, "threads" is generally accepted as a generic term for subtasks. Threads will often need to update some variable that is shared between them. The instructions between the two programs may be interleaved in any order. For example, consider the following program:

Thread A	Thread B
1A: Read variable V	1B: Read variable V
2A: Add 1 to variable V	2B: Add 1 to variable V
3A Write back to variable V	3B: Write back to variable V

If instruction 1B is executed between 1A and 3A, or if instruction 1A is executed between 1B and 3B, the program will produce incorrect data. This is known as a race condition. The programmer must use a lock to provide mutual exclusion. A lock is a programming language construct that allows one thread to take control of a variable and prevent other threads from reading or writing it, until that variable is unlocked. The thread holding the lock is free to execute its critical section (the section of a program that requires exclusive access to some variable), and to unlock the data when it is finished. Therefore, to guarantee correct program execution, the above program can be rewritten to use locks:

Thread A	Thread B
1A: Lock variable V	1B: Lock variable V
2A: Read variable V	2B: Read variable V
3A: Add 1 to variable V	3B: Add 1 to variable V
4A Write back to variable V	4B: Write back to variable V
5A: Unlock variable V	5B: Unlock variable V

One thread will successfully lock variable V, while the other thread will be locked out—unable to proceed until V is unlocked again. This guarantees correct execution of the program. Locks, while necessary to ensure correct program execution, can greatly slow a program.

Locking multiple variables using non-atomic locks introduces the possibility of program deadlock. An atomic lock locks multiple variables all at once. If it cannot lock all of them, it does not lock any of them. If two threads each need to lock the same two variables using non-atomic locks, it is possible that one thread will lock one of them and the second thread will lock the second variable. In such a case, neither thread can complete, and deadlock results.

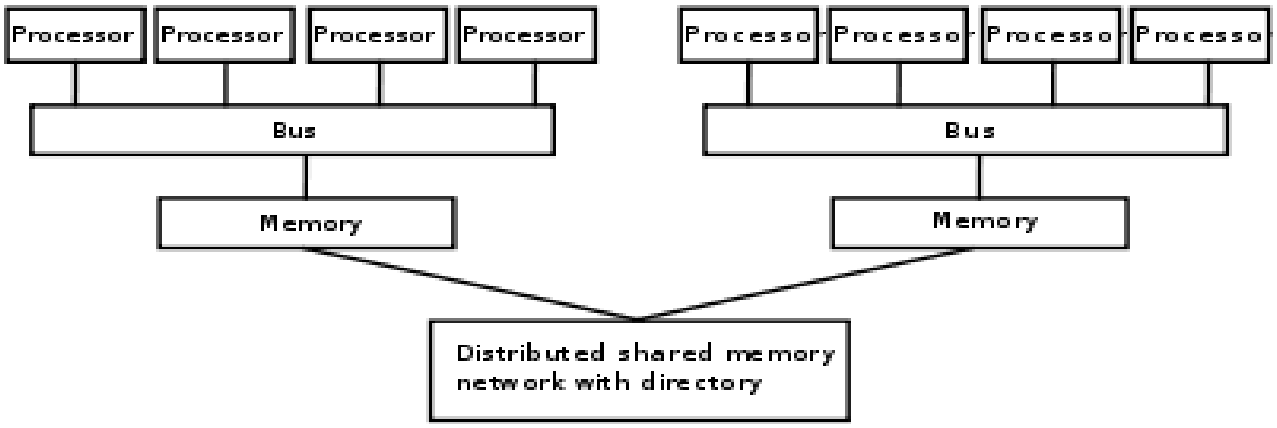
Many parallel programs require that their subtasks act in synchrony. This requires the use of a barrier Barriers are typically implemented using a software lock. One class of algorithms, known as lock-free and wait-free algorithms, altogether avoids the use of locks and barriers. However, this approach is generally difficult to implement and requires correctly designed data structures.

Not all parallelization results in speed-up. Generally, as a task is split up into more and more threads, those threads spend an ever-increasing portion of their time communicating with each other. Eventually, the overhead from communication dominates the time spent solving the problem, and further parallelization (that is, splitting the workload over even more threads) increases rather than decreases the amount of time required to finish. This is known as parallel slowdown.

3.4 MEMORY AND COMMUNICATION

Main memory in a parallel computer is either shared memory (shared between all processing elements in a single address space), or distributed memory (in which each processing element has its own local address space) Distributed memory refers to the fact that the memory is logically distributed, but often implies that it is physically distributed as well. Distributed shared memory and memory virtualization combine the two approaches, where the processing element has its own local memory and access to the memory on non-local processors. Accesses to local memory are typically faster than accesses to non-local memory. A logical view of a Non-Uniform Memory Access (NUMA) architecture. Processors in one directory can access that directory's memory with less latency than they can access memory in the other directory's memory.

Computer architectures in which each element of main memory can be accessed with equal latency and bandwidth are known as Uniform Memory Access (UMA) systems. Typically, that can be achieved only by a shared memory system, in which the memory is not physically distributed. A system that does not have this property is known as a Non-Uniform Memory Access (NUMA) architecture. Distributed memory systems have non-uniform memory access.



Computer systems make use of caches—small, fast memories located close to the processor which store temporary copies of memory values (nearby in both the physical and logical sense). Parallel computer systems have difficulties with caches that may store the same value in more than one location, with the possibility of incorrect program execution. These computers require a cache coherency system, which keeps track of cached values and strategically purges them, thus ensuring correct program execution. Bus snooping is one of the most common methods for keeping track of which values are being accessed (and thus should be purged). Designing large, high-performance cache coherence systems is a very difficult problem in computer architecture. As a result, shared-memory computer architectures do not scale as well as distributed memory systems do.

Processor–processor and processor–memory communication can be implemented in hardware in several ways, including via shared (either multiported or multiplexed) memory, a crossbar switch, a shared bus or an interconnect network of a myriad of topologies including star, ring, tree, hypercube, fat hypercube (a hypercube with more than one processor at a node), or n-dimensional mesh.

Parallel computers based on interconnect networks need to have some kind of routing to enable the passing of messages between nodes that are not directly connected. The medium used for communication between the processors is likely to be hierarchical in large multiprocessor machines.

3.5 PARALLEL PROGRAMMING LANGUAGES

Concurrent programming languages, libraries, APIs, and parallel programming models (such as Algorithmic Skeletons) have been created for programming parallel computers. These can generally be divided into classes based on the assumptions they make about the underlying memory architecture—shared memory, distributed memory, or shared distributed memory. Shared memory programming languages communicate by manipulating shared memory variables. Distributed memory uses message passing. POSIX Threads and OpenMP are two of most widely used shared memory APIs, whereas Message Passing Interface (MPI) is the most widely used message-passing system API. One concept used in programming parallel programs is the future concept, where one part of a program promises to deliver a required datum to another part of a program at some future time.

CAPS entreprise and Pathscale are also coordinating their effort to make HMPP (Hybrid Multicore Parallel Programming) directives an Open Standard denoted OpenHMPP. The OpenHMPP directive-based programming model offers a syntax to efficiently offload computations on hardware accelerators and to optimize data movement to/from the hardware memory. OpenHMPP directives describe remote procedure call (RPC) on an accelerator device (e.g. GPU) or more generally a set of cores. The directives annotate C or Fortran codes to describe two sets of functionalities: the offloading of procedures (denoted codelets) onto a remote device and the optimization of data transfers between the CPU main memory and the accelerator memory.

CHAPTER 4

PROPOSED MODEL

4.1 OVERVIEW

The gcc compiler source code is modified and included a separate flag (compilation option) called parallelize (invoked using gcc -parallelize *.c) that reads in a plain C file and converts it into an executable optimized for a multi-core processor.

We propose an automatic parallelization tool plugged into gcc which has the following functionalities.

4.2 PRE-PARALLELIZATION STEPS

1) Code Optimization:

The previously written codes will follow traditional Sequential approach, The Code has to be optimized either manually or automatically to achieve maximum performance before going to parallelization phase

2) Profiling:

The input code is analyzed for hotspots where the processor spends most of the time. Multiple hotspots are detected and profile is created based on the call graph and other vital parameters.

3) Analyzing dependencies:

The following dependencies are analyzed

i) Flow dependence

ii) Anti-dependence

iii) Output dependence

iv) I/O dependence

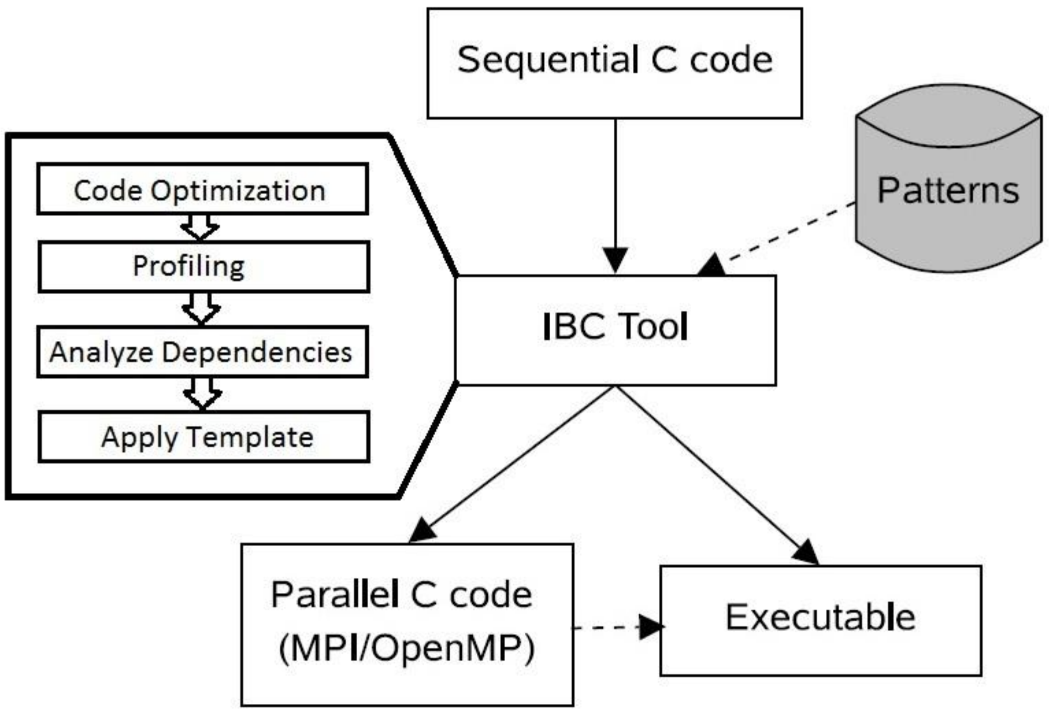
4) Identifying suitable predefined templates to apply:

The identified pattern is replaced with pre-defined templates. The pre-defined templates serve as the solution to the currently identified pattern of potentially parallelizable code.

5) Error Checking:

Finally the Error checking is done to identify the errors which might have been introduced to the parallel code during the process of automation.

Block diagram of Intent-Based Compilation Tool



IBC - Intent Based Compiler

4.3 STEPS INVOLVED

1. The process starts with identifying code sections that the programmer feels have parallelism possibilities. Often this task is difficult since the programmer who wants to parallelize the code has not originally written the code under consideration. Another possibility is that the programmer is new to the application domain. Thus, though this first stage in the parallelization process and seems easy at first it may not be so.

2. The next stage is to shortlist code sections out of the identified ones that are actually parallelization. This stage is again most important and difficult since it involves lot of analysis. Generally for codes in C/C++ where pointers are involved are difficult to analyze. Many special techniques such as pointer alias analysis, functions side effects analysis are required to conclude whether a section of code is dependent on any other code. If the dependencies in the identified code sections are more the possibilities of parallelization decreases.

3. Sometimes the dependencies are removed by changing the code and this is the next stage in parallelization. Code is transformed such that the functionality and hence the output is not changed but the dependency, if any, on other code section or other instruction is removed.

4. The last stage in parallelization is generating the parallel code. This code is always functionally similar to the original sequential code but has additional constructs or code sections which when executed create multiple threads or processes.

4.4 APPLICATIONS

1. Relieve programmers from the tedious and error-prone manual parallelization process.

2. Already Parallelized problems serve as the template for future encounters of similar type of problems, thus code reusability can be achieved.

3. Save time and/or money: In theory, throwing more resources at a task will shorten its time to completion, with potential cost savings. Parallel computers can be built from cheap, commodity components.

4. Solve larger problems: Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory.

5. Provide concurrency: A single compute resource can only do one thing at a time. Multiple computing resources can be doing many things simultaneously.

CHAPTER 5

IMPLEMENTATION

The proposed model can be implemented with following minimum system requirements.

Hardware Requirements

Processor: Any processor architecture supported by latest GCC, preferably with multicore architecture.

Memory: Atleast 1GB RAM

Disk Storage: Minimum 5 GB free Disk space

Software requirements

Operating system: MatrixOS v1.01

Packages: Prebuilt binaries of gcc, g++, flex, bison and OpenJDK

5.1 BUILDING A LOCAL COPY OF GCC

A copy of gcc is locally built to experiment and implement the proposed system.

Steps Involved:

1) Download gcc source code:

The GNU Compiler Collection is an open source project where thousands of developers contribute to the project. Since the project is open source we are free to do any modification to the project and redistribute it.

The source code of latest gcc will be hosted at many places including github and multiple alternate servers. The source code is downloaded from *http://gcc.gnu.org/*

2) Choice of Operating system:

Our choice of operating system is MatrixOS 1.01, which is based on Ubuntu which is again based on debian. MatrixOS provides most of the development required packages out of the box.

3) Satisfy pre-requisites of gcc package:

The gcc package requires quite couple of packages to be present in the building environment. Gcc is dependent on many other packages. All its dependencies must be satisfied before we start building our own gcc local copy. The following are the pre requisites and the guide to resolve them.

3.1) Assumptions and shorthand's:

All commands mentioned hereafter can be executed by assuming the following points:

i) ~ is **/home/neo**

ii) All the required packages are downloaded into the path: **~/build**

iii) All the downloaded packages are compressed in tarball format

iv) Most of the basic packages have been already installed in the system: eg. Gcc,tar,zip etc.

3.2) GMP:

GMP is a free library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating point numbers. There is no practical limit to the precision except the ones implied by the available memory in the machine GMP runs on. GMP has a rich set of functions, and the functions have a regular interface. It is required for MFPR and thus for GCC. Go to *http://gmplib.org/* and download the source (e.g., gmp-5.0.2.tar.bz2).

Building GMP (the path assumptions are mentioned previously):

```
$mkdir ~/gmp-5.0.2 ~/build
```

```
$cd ~/build
$tar xjf gmp-5.0.2.tar.bz2
$cd gmp-5.0.2
$./configure --prefix=/home/neo/gmp-5.0.2 --enable-
cxx
$nice -n 19 time make -j8
$make install
## Optionally, test our build of gmp:
$make check
$echo $?
```

Assuming make check ran successfully and $? returned 0 , we proceed to the next step

3.3) MPFR:

The MPFR library is a C library for multiple-precision floating-point computations with correct rounding. MPFR is a library for arbitrary precision floating-number arithmetic which produces exactly same results for any CPU or operating systems. It is required for GCC. Go to *http://mpfr.org/* and download the source (e.g., mpfr-3.0.1.tar.bz2).

Building MPFR:

```
$mkdir ~/mpfr-3.1.0
$cd ~/build
$tar xjf mpfr-3.1.0.tar.bz2
$cd mpfr-3.1.0
$./configure --prefix=/home/neo/mpfr-3.1.0 --with-
$gmp=/home/neo/gmp-5.0.2
$nice -n 19 time make -j8
$make install
```

3.4) MPC:

Gnu Mpc is a C library for the arithmetic of complex numbers with arbitrarily high precision and correct rounding of the result. It extends the principles of the IEEE-754 standard for fixed precision real floating point numbers to complex numbers, providing well-defined semantics for every operation. At the same time, speed of operation at high precision is a major design goal. Go to *http://www.multiprecision.org/* and download the source (e.g., mpc-0.9.tar.gz)

Building MPC:

```
$mkdir ~/mpc-0.9
$cd ~/build
$tar xzf mpc-0.9.tar.gz
$cd mpc-0.9
$LD_LIBRARY_PATH=/home/neo/gmp-
5.0.2/lib:/home/neo/mpfr-3.1.0/lib ./configure --
prefix=/home/neo/mpc-0.9 --with-gmp=/home/neo/gmp-
5.0.2 --with-mpfr=/home/neo/mpfr-3.1.0
$LD_LIBRARY_PATH=/home/neo/gmp-
5.0.2/lib:/home/neo/mpfr-3.1.0/lib nice -n 19 time
make -j8
$make install
```

3.5) ERRORS WHILE INSTALLING PRE-REQUISITES

We encountered the following two errors while installing the above mentioned packages.

Error1:

```
/usr/include/features.h:323:26: fatal error:
bits/predefs.h: No such file or directory.
```

Resolution:

```
$sudo apt-get install libc6-dev-i386
```

Error2:

```
To solve errors about crti.o not being found.
```

Resolution:

```
$cd /usr/lib
$ln -s x86_64-linux-gnu/crt*.o .
```

Error3:

```
To avoid gmp.h not found error.
```

Resolution:

```
sudo apt-get install  libgmp3-dev
```

4) INSTALL GCC

After installing all the pre-requisites, gcc is built. As mentioned above the downloaded source tarball is assumed to be present in *~/build*.

```
$mkdir ~/gcc-4.4.0
$cd ~/build
$tar xjf gcc-4.4.0.tar
$cd gcc-4.4.0

$LD_LIBRARY_PATH=/home/neo/gmp-
5.0.2/lib:/home/neo/mpfr-3.1.0/lib:/home/neo/mpc-
0.9/lib

$./configure --prefix=/home/neo/gcc-4.4.0 --with-
gmp=/home/neo/gmp-5.0.2 --with-mpfr=/home/neo/mpfr-
```

```
3.1.0 --with-mpc=/home/neo/mpc-0.9 --disable-
multilib --disable-libgcj

$LD_LIBRARY_PATH=/home/neo/gmp-
5.0.2/lib:/home/neo/mpfr-3.1.0/lib:/home/neo/mpc-
0.9/lib

$nice -n 19 time make -j8
$make install
```

The binaries can be found at *~/gcc-4.4.0* which is our *local copy* of *gcc*.

Each time when we change the sourcecode of gcc, gcc has to be build and the binaries has to be installed at the appropriate location from where we wish to run the program.

5.2 GOAL OF MODIFYING GCC

The modified gcc should take the following as arguments and initiate corresponding processing functions

i) **-fparallel:**

This flag when given as an argument, initiates the analyzation of non-nested loops in source code and produces corresponding parallel source code and binaries under the current working directory.

ii) **-fnested:**

This flag when used along with *fparallel* flag, the compiler initiates a nested loop analyzation and produces corresponding parallel source code and binaries under the current working directory.

iii) **-finteractive:**

This flag when used along with *fparallel* flag, a GUI is presented to the user with which the user can dynamically interact with the source code and can accept the suggestions given by the parallel compiler engine for individual for loops.

5.3 MODIFYING GCC

The following modification are done to the original source code of gcc4.4.0

5.3.1 Invoking gcc

Three possible ways of invoking modified gcc are:

i) gcc -fparallel input.c

ii) gcc -fparallel -fnested input.c

iii) gcc -fparallel -finteractive input.c

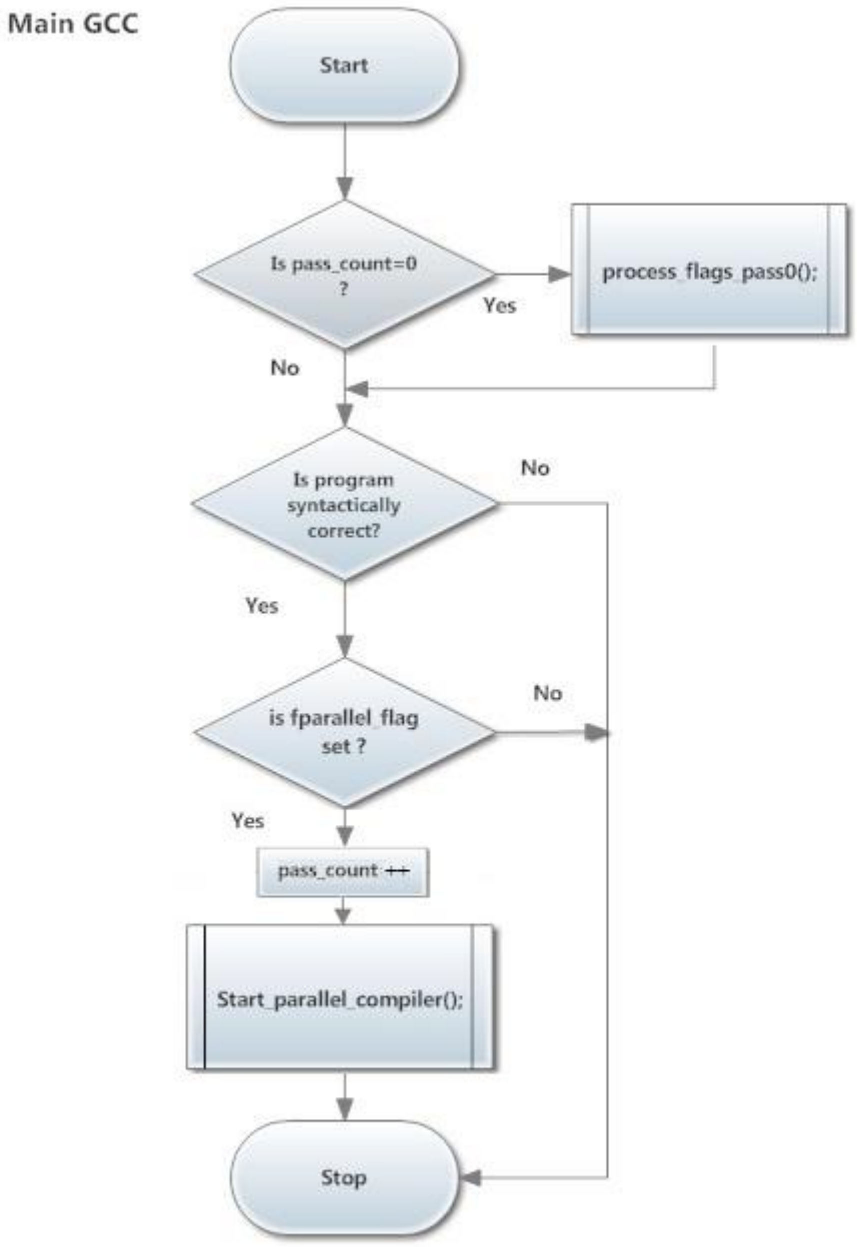


Fig 5.3.1.1 Flowchart of The Main program in gcc

Even though the program is externally called once, internally the the program is invoked twice, this is termed as *pass0* and pass1

pass0:

The variable pass_count is initialized to zero which indicates it's the first pass. The flags passsed are processed and syntax checking is done. The parallel conversion of sequential code is done in this step. And finally the pass_count variable is incremented.

pass1:

The suitable header files and compilation flags are appended and compilation is done in this step.

5.3.2) Processing flags:

The parallel compiler engine called only if any of the pre-defined three flags is found. Upon the not detection of parallel flags, normal execution of gcc is continued.

5.3.3) Syntax Checking:

In pass0, the compilation is stopped in between by stopping the compilation process just before running the linker. The compile time argument "-c" allows us to achieve this. By invoking gcc as "*gcc -c input.c*", the compilation is stopped just before the linker process. This allows us to detect any syntactical error before commencing parallel processing engine. gcc on successfully generating the object files after compilation sets the variable *"return_status"* to *"0"* . In case of any syntactical or any kind of other errors, the "*return_status"* will be non-zero value.

The *"return_status"* value is taken as the deciding factor for the calling of parallel processing engine. Even if the parallel processing flags are set the parallel processing is not initiated if the *"return_status"* is a non-zero value.

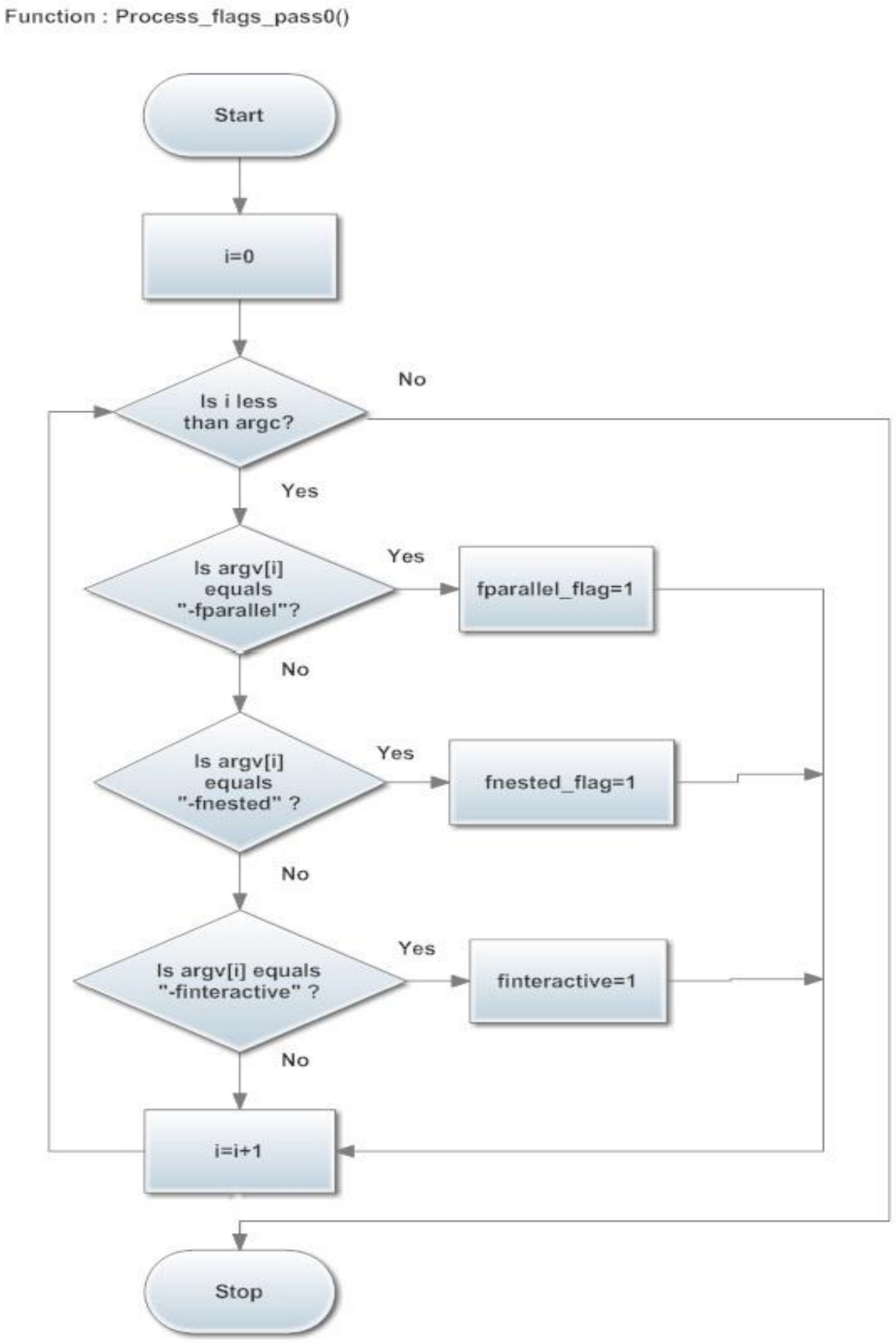

Function : Process_flags_pass0()
Start
i=0
Is i less than argc?
No
Yes
Is argv[i] equals "-fparallel"?
Yes
fparallel_flag=1
No
Is argv[i] equals "-fnested" ?
Yes
fnested_flag=1
No
Is argv[i] equals "-finteractive" ?
Yes
finteractive=1
No
i=i+1
Stop

Fig 5.3.3.1 Flag processing

The command line arguments will be present in *argv*. This is iterated against known parallel flags. Whenever a parallel flag is detected, corresponding flag is set to 1. If none of the parallel flags are found, normal execution of gcc is resumed.

5.3.4) Parallel processing:

The input source files can be referenced through the datastructure called *infiles*. The input sequential source files which has to be converted to parallel code are read from this datastrure and are sent for parallel processing engine.

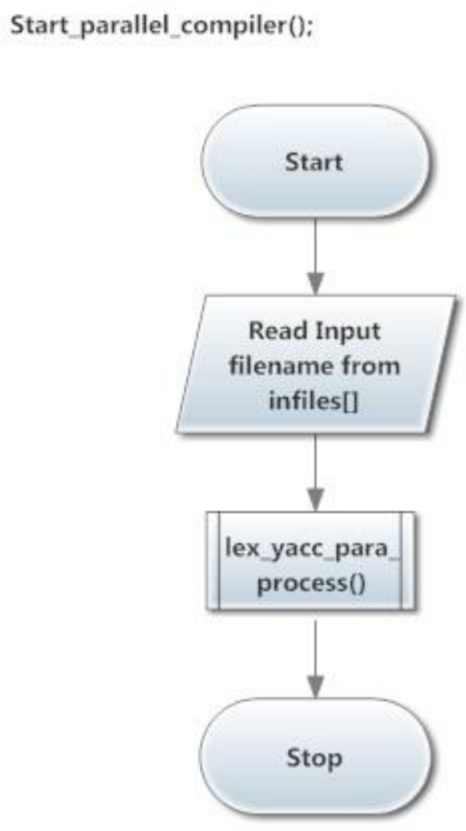

Start_parallel_compiler();
Start
Read Input filename from infiles[]
lex_yacc_para_process()
Stop

Fig 5.3.4.1 Sequential to parallel transformer

5.3.5) Processing of for loops using yacc grammar

The input file in analyzed and for loops are identified. Each identified for loops are uniquely labelled and their nested levels are stored in a datastructure. The individual for loop are written to a temporary buffer.

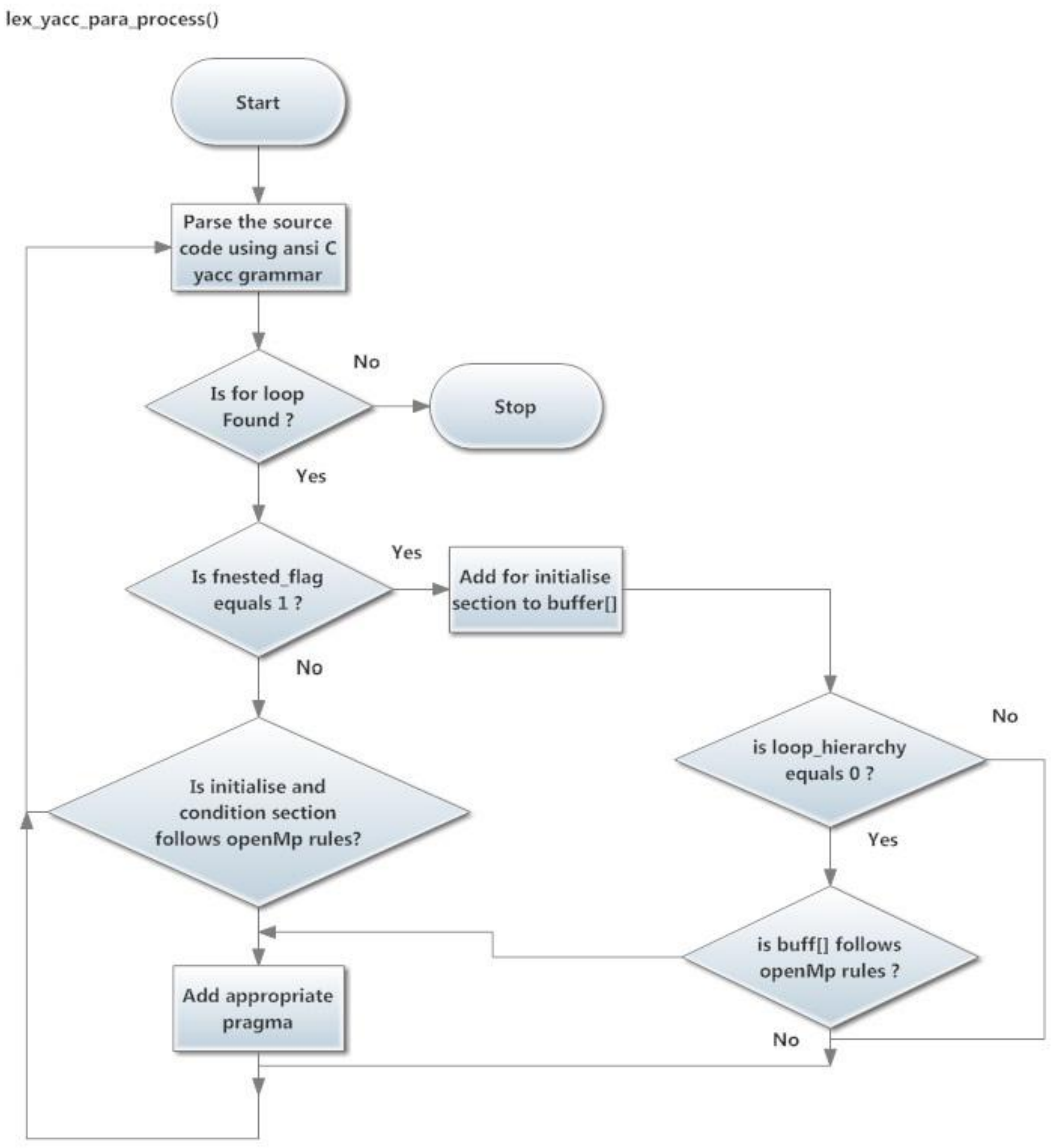

lex_yacc_para_process()
Start
Parse the source code using ansi C yacc grammar
Is for loop Found ?
No
Stop
Yes
Is fnested_flag equals 1 ?
Yes
Add for initialise section to buffer[]
No
Is initialise and condition section follows openMp rules?
is loop_hierarchy equals 0 ?
No
Yes
is buff[] follows openMp rules ?
No
Add appropriate pragma

Fig 5.3.5.1 for loop processing using yacc grammar

The buffer file is written with following properties:

1. Unique ID:

Each for loop is labelled with an identifier with which it can be uniquely dereferenced. This id is also used to identify the individual nodes which are constructed later.

2) Nested *for loop* Level ID:

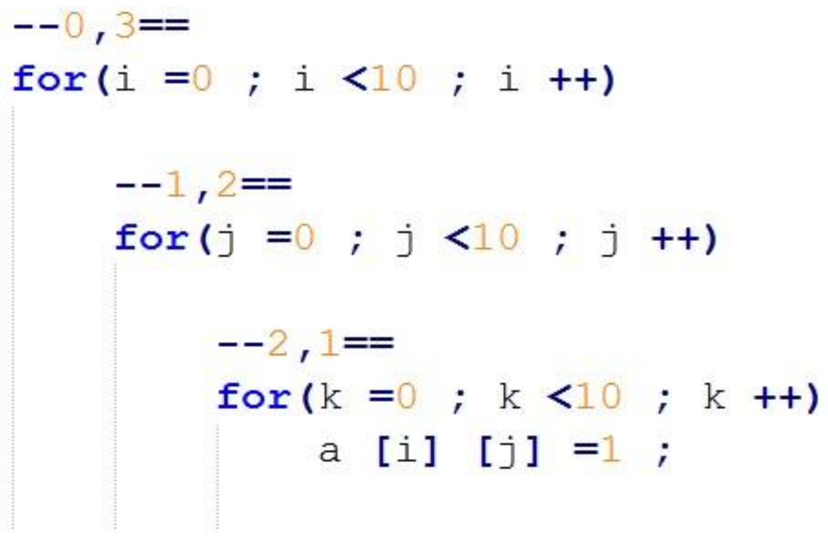
This is a successive for loop ID with which the nested loop levels are identified.

3) Non-nested *for loop* predictive pragmas:

After the analyzation of for loops, the parallel processing engine predicts the suitable pragmas to apply and its corresponding private and shared variables. This is with effect to the topmost for loop identified. This predictive pragmas are written to a separate file. And then is used for further processing.

4) Nested for loop predictive pragmas:

This contains the predicted pragmas for the nested for loops and is written to a separate file.



```
--0,3==
for(i =0 ; i <10 ; i ++)

    --1,2==
    for(j =0 ; j <10 ; j ++)

        --2,1==
        for(k =0 ; k <10 ; k ++)
            a [i] [j] =1 ;
```

Fig5.3.5.1 Labelling UID and forID

5.3.6) Interactive *for loop* tree :

The for loop tree node is constructed from the buffer files. The parameters which are required to construct the individual node are:

i) UID:

This is the unique for loop identifier which is used to identify the individual nodes.

ii) forID:

This is used to reveal the nested level of for loops identified. Each successive nested for loops are labelled according to the nested levelness.

iii) pragma:

This contains the non-nested predicted openmp pragma which is supposed to be applied to the source code if nested flag is turned off.

iv) pragmaNested:

This contains the nested predicted openmp pragma which is supposed to be applied to the source code if nested flag is turned on.

v) body:

The string inside this variable contains the entire for loop body of the for loop under inspection. The body contains even nested for loops.

vi) loopHead:

This is the short description of the individual for loop which is recognized and represented as a node. Usually loopHead contains the initialization variable, condition expression and incremental operation.

vii) userPragma:

If the user modifies the suggested pragma, then the modified strings are stored in this datastructure

viii) isParallelizable:

This is a Boolean value which is set depending on the parallelizability of a particular for loop based on the constraints specified by OpenMP specification

The GUI is designed in such a way that the *fir loop* tree is easily modifiable by the user. Using the hints provided by the parallel processor engine, user converts the sequential code to parallel code.

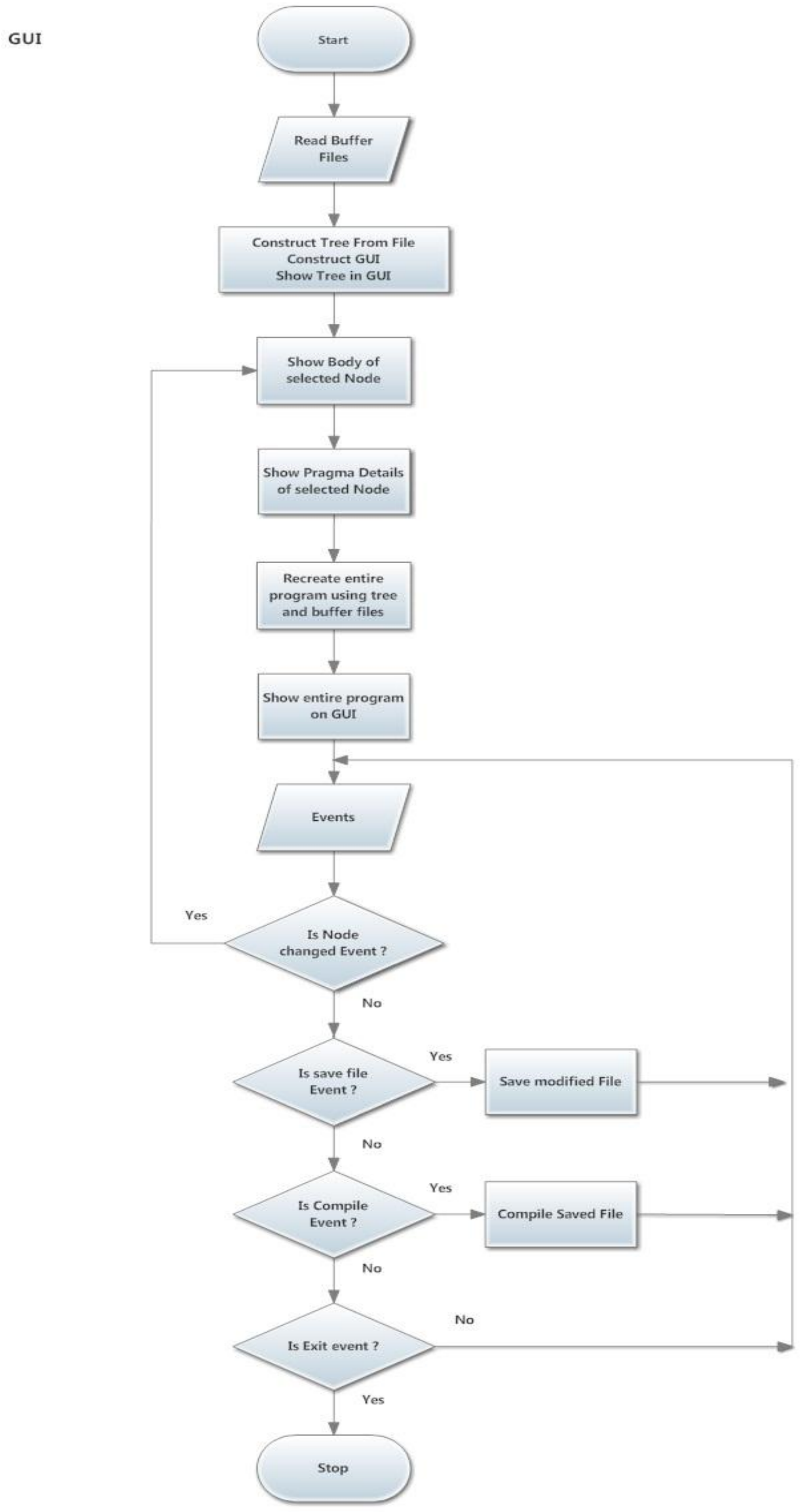


Fig 5.3.6.1 GUI of user interactive session

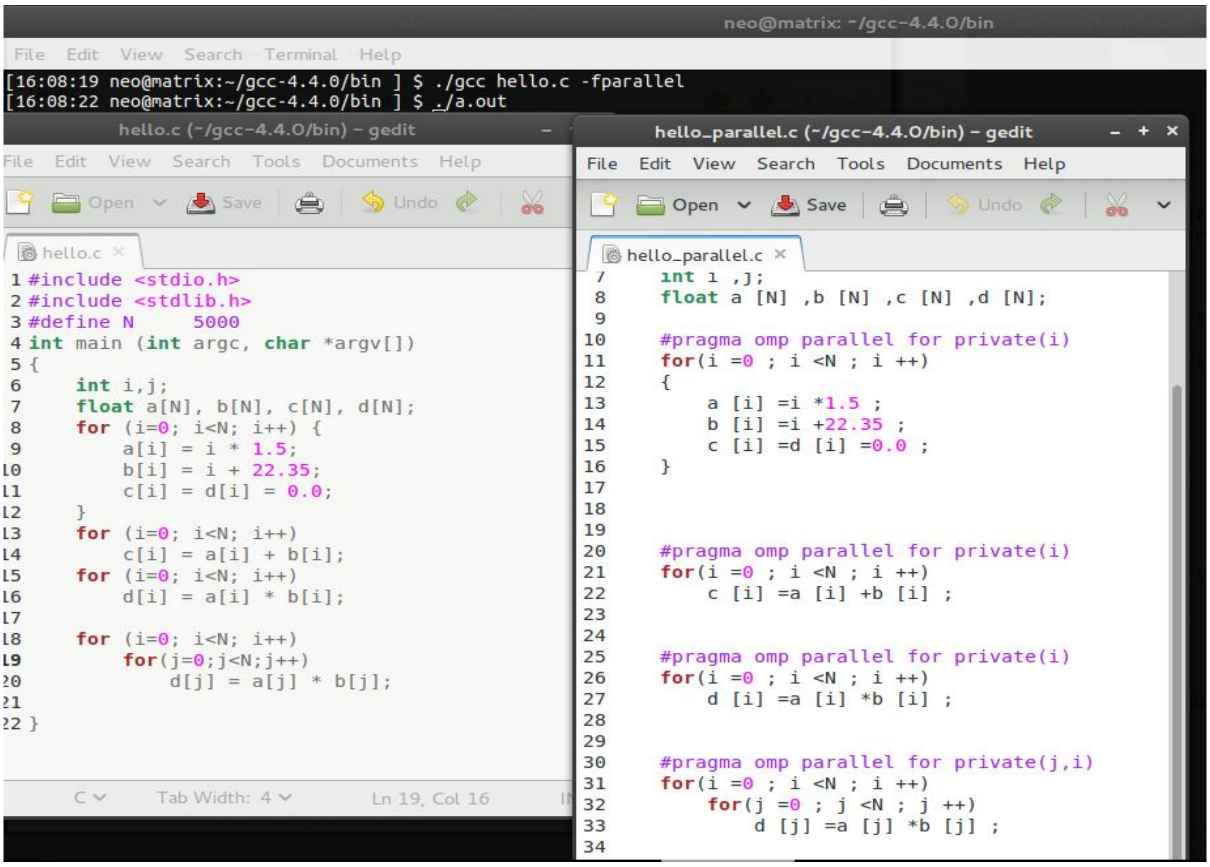
CHAPTER 6

Working with Extended GCC

This section explains the working with extended gcc with descriptive screenshots.

6.1 fparallel FLAG

Initially the *extended gcc* is invoked with *–fparallel* flag. This initiates the parallel processing engine with non-nested *for loop* analyzation. After the processing of input source code which is presented on the left side of the following screenshot is converted to OpenMP compatible parallelized source code which is presented on the right side of the following screenshot.


neo@matrix: ~/gcc-4.4.0/bin
File Edit View Search Terminal Help
hello.c (~/gcc-4.4.0/bin) - gedit
hello_parallel.c (~/gcc-4.4.0/bin) - gedit
File Edit View Search Tools Documents Help
Open Save Undo
hello.c
hello_parallel.c
C Tab Width: 4 Ln 19, Col 16

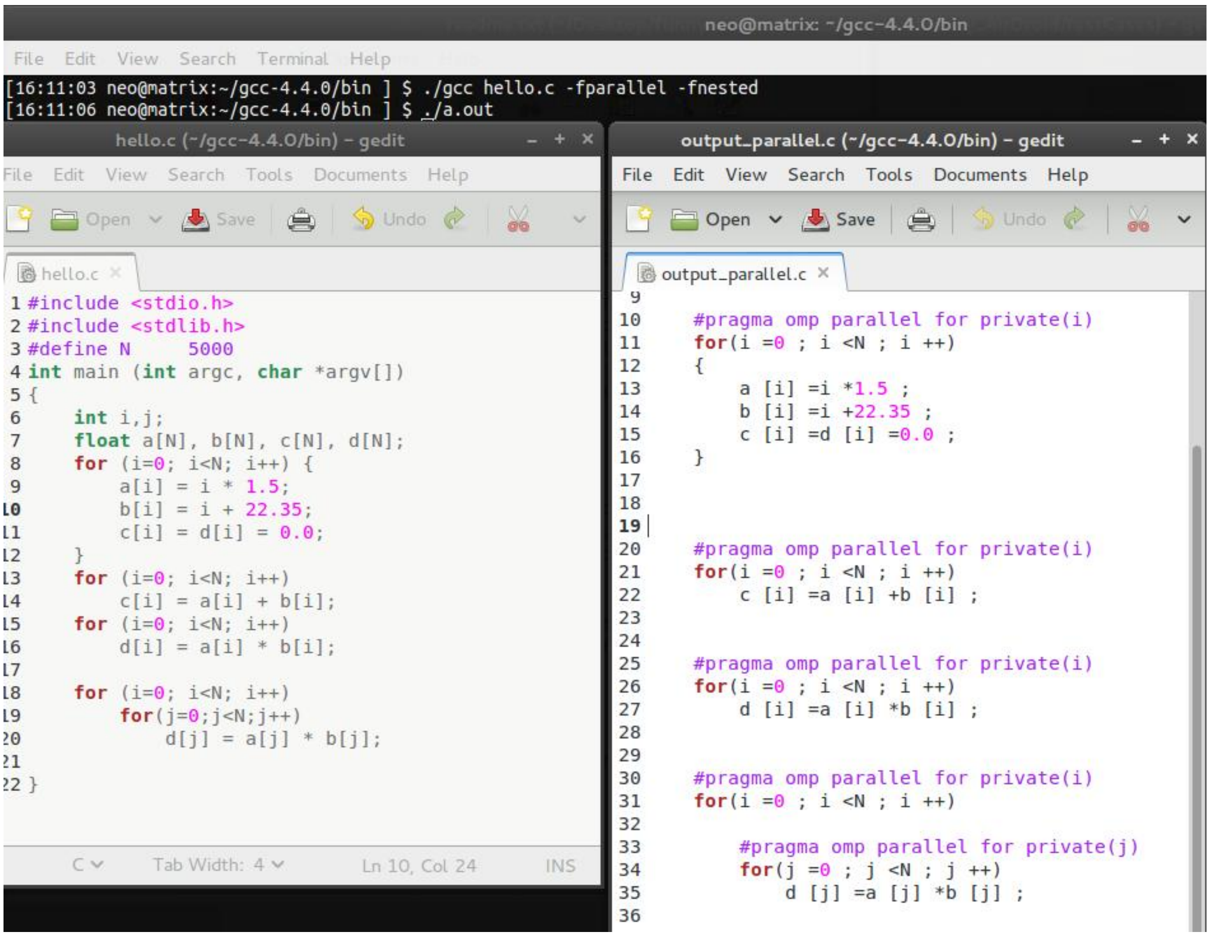
```
[16:08:19 neo@matrix:~/gcc-4.4.0/bin ] $ ./gcc hello.c -fparallel
[16:08:22 neo@matrix:~/gcc-4.4.0/bin ] $ ./a.out
```

```
#include <stdio.h>
#include <stdlib.h>
#define N      5000
int main (int argc, char *argv[])
{
    int i,j;
    float a[N], b[N], c[N], d[N];
    for (i=0; i<N; i++) {
        a[i] = i * 1.5;
        b[i] = i + 22.35;
        c[i] = d[i] = 0.0;
    }
    for (i=0; i<N; i++)
        c[i] = a[i] + b[i];
    for (i=0; i<N; i++)
        d[i] = a[i] * b[i];

    for (i=0; i<N; i++)
        for(j=0;j<N;j++)
            d[j] = a[j] * b[j];

}
```

```
    int i ,j;
    float a [N] ,b [N] ,c [N] ,d [N];

    #pragma omp parallel for private(i)
    for(i =0 ; i <N ; i ++)
    {
        a [i] =i *1.5 ;
        b [i] =i +22.35 ;
        c [i] =d [i] =0.0 ;
    }



    #pragma omp parallel for private(i)
    for(i =0 ; i <N ; i ++)
        c [i] =a [i] +b [i] ;


    #pragma omp parallel for private(i)
    for(i =0 ; i <N ; i ++)
        d [i] =a [i] *b [i] ;


    #pragma omp parallel for private(j,i)
    for(i =0 ; i <N ; i ++)
        for(j =0 ; j <N ; j ++)
            d [j] =a [j] *b [j] ;

```

6.2 –fnested FLAG

When *extended gcc* is invoked with *–fnested* flag along with *–fparallel* flag, the parallel processing engine initiates the nested for loop analyzation and predicts the OpenMP pragmas even for nested fo loops of any level.



```
[16:11:03 neo@matrix:~/gcc-4.4.0/bin ] $ ./gcc hello.c -fparallel -fnested
[16:11:06 neo@matrix:~/gcc-4.4.0/bin ] $ ./a.out
```

hello.c (~/gcc-4.4.0/bin) - gedit

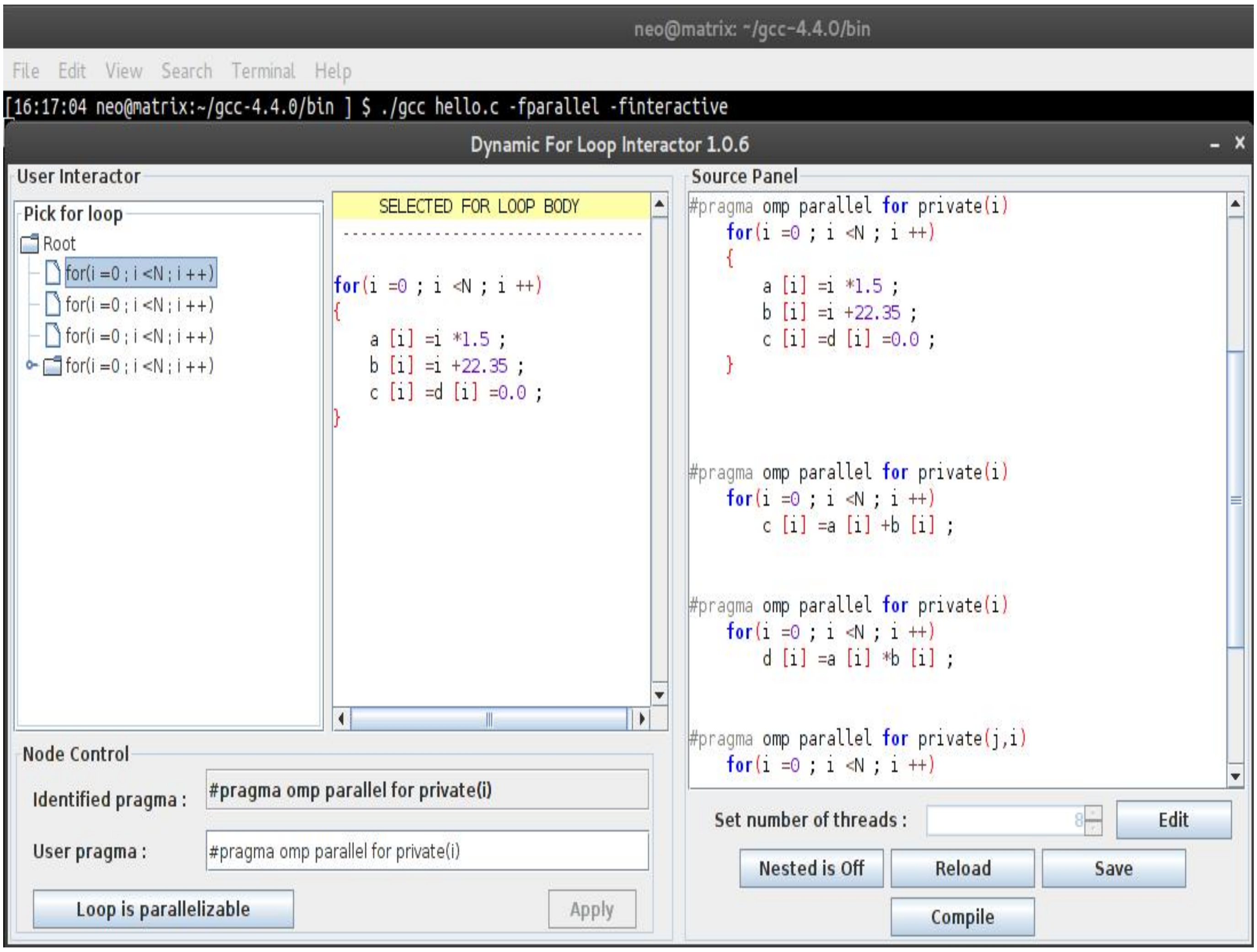
```
#include <stdio.h>
#include <stdlib.h>
#define N     5000
int main (int argc, char *argv[])
{
    int i,j;
    float a[N], b[N], c[N], d[N];
    for (i=0; i<N; i++) {
        a[i] = i * 1.5;
        b[i] = i + 22.35;
        c[i] = d[i] = 0.0;
    }
    for (i=0; i<N; i++)
        c[i] = a[i] + b[i];
    for (i=0; i<N; i++)
        d[i] = a[i] * b[i];

    for (i=0; i<N; i++)
        for(j=0;j<N;j++)
            d[j] = a[j] * b[j];

}
```

output_parallel.c (~/gcc-4.4.0/bin) - gedit

```
    #pragma omp parallel for private(i)
    for(i =0 ; i <N ; i ++)
    {
        a [i] =i *1.5 ;
        b [i] =i +22.35 ;
        c [i] =d [i] =0.0 ;
    }



    #pragma omp parallel for private(i)
    for(i =0 ; i <N ; i ++)
        c [i] =a [i] +b [i] ;


    #pragma omp parallel for private(i)
    for(i =0 ; i <N ; i ++)
        d [i] =a [i] *b [i] ;


    #pragma omp parallel for private(i)
    for(i =0 ; i <N ; i ++)

        #pragma omp parallel for private(j)
        for(j =0 ; j <N ; j ++)
            d [j] =a [j] *b [j] ;

```

6.3 GRAPHICAL USER INTERFACE WITH –*finteractive* FLAG

The GUI is launched when the –*finteractive* flag is specified along with –*fparallel* flag.



```
[16:17:04 neo@matrix:~/gcc-4.4.0/bin ] $ ./gcc hello.c -fparallel -finteractive
```

```
for(i =0 ; i <N ; i ++)
for(i =0 ; i <N ; i ++)
for(i =0 ; i <N ; i ++)
for(i =0 ; i <N ; i ++)
```

```
for(i =0 ; i <N ; i ++)
{
    a [i] =i *1.5 ;
    b [i] =i +22.35 ;
    c [i] =d [i] =0.0 ;
}
```

```
#pragma omp parallel for private(i)
    for(i =0 ; i <N ; i ++)
    {
        a [i] =i *1.5 ;
        b [i] =i +22.35 ;
        c [i] =d [i] =0.0 ;
    }


#pragma omp parallel for private(i)
    for(i =0 ; i <N ; i ++)
        c [i] =a [i] +b [i] ;


#pragma omp parallel for private(i)
    for(i =0 ; i <N ; i ++)
        d [i] =a [i] *b [i] ;


#pragma omp parallel for private(j,i)
    for(i =0 ; i <N ; i ++)
```

Identified pragma : #pragma omp parallel for private(i)

User pragma : #pragma omp parallel for private(i)

Set number of threads : 8

6.3.1 UNDERSTANDING THE "*Dynamic For Loop Interactor 1.0.6*"

Dynamic For Loop Interactor is a GUI written to compliment the development of parallel code along with extended gcc.

The GUI had 5 main components:

1) *for loop* picker
2) Selected *for loop* display
3) Source panel
4) Node Control
5) Control panel

i) *for loop* picker :

The entire input source code is analyzed for *for loops* and identified for loops are treated as individual nodes. These nodes contain necessary information required for the conversion of serial to parallel code. The nodes are represented here as interactive nodes and certain events are triggered when the nodes are clicked upon. These events include dynamic refresh of other panels with relative date corresponding to the clicked node.

ii) Selected *for loop* display :

This panel displays the entire for loop body for the current selected for loop node. This panel is refreshed whenever the node is clicked.

iii) Source panel :

This panel displays the entire source code input. Whenever any node is clicked, the cursor is moved to the corresponding position in the source code for easy analyzation.

iv) Node control

This panel contains the identified pragma which is suggested by the parallel processor engine. The user can edit this pragma to suit his need.

The other entity in this panel is a "*Loop is parallelizable*" toggle button. The user can specify whether a particular identified loop parallelizable or not.

v) Control Panel:

Control panel has the following entities:

1) **Set number of threads:**
   Our program automatically identifies the number of cores in the current executing system and set it as default value. But user can manually change this value.

2) **Nested toggle button:**
   This toggle button switches between nested for loop analyzation *on* and *off*.

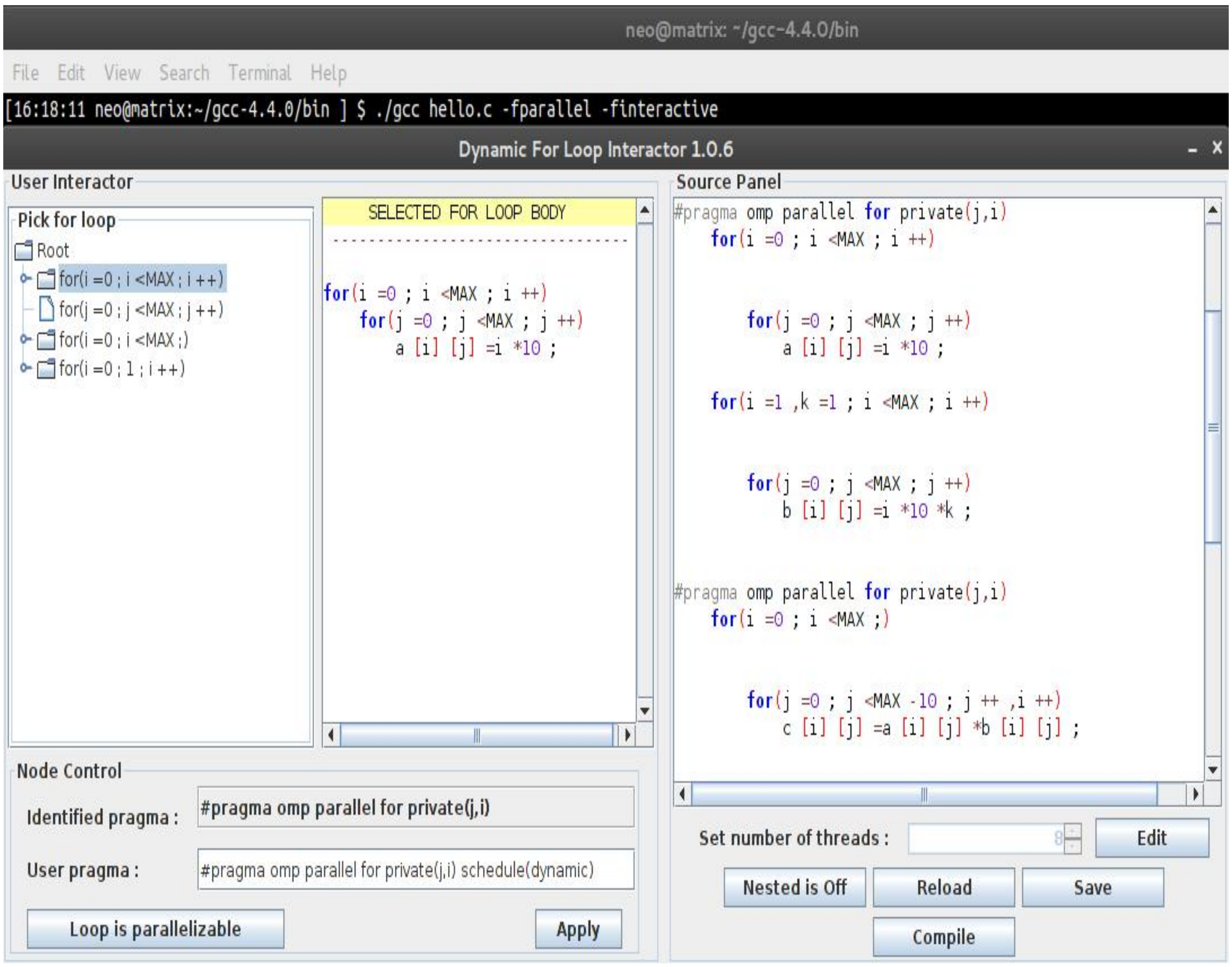
3) **Reload button:**
   Discards the changes made after last save. Provides a confirmation dialogue to prevent accidental changes.

4) **Save Button:**
   Saves the modified code to file. File path can be specified on clicking of this button.

5) **Compile:**
   The saved file is compiled with required OpenMP flags and headers added

6.4 MODIFYING SUGGESTED OpenMP PRAGMAS

The suggested OpenMP pragma can be modified dynamically using the node control panel. This is particularly helpful in specifying *schedule* and *reduction* clause.



```
[16:18:11 neo@matrix:~/gcc-4.4.0/bin ] $ ./gcc hello.c -fparallel -finteractive
```

```
Root
  for(i =0 ; i <MAX ; i ++)
    for(j =0 ; j <MAX ; j ++)
  for(i =0 ; i <MAX ;)
  for(i =0 ; 1 ; i ++)
```

```
for(i =0 ; i <MAX ; i ++)
    for(j =0 ; j <MAX ; j ++)
        a [i] [j] =i *10 ;
```

```
#pragma omp parallel for private(j,i)
    for(i =0 ; i <MAX ; i ++)


        for(j =0 ; j <MAX ; j ++)
            a [i] [j] =i *10 ;

    for(i =1 ,k =1 ; i <MAX ; i ++)


        for(j =0 ; j <MAX ; j ++)
            b [i] [j] =i *10 *k ;


#pragma omp parallel for private(j,i)
    for(i =0 ; i <MAX ;)


        for(j =0 ; j <MAX -10 ; j ++ ,i ++)
            c [i] [j] =a [i] [j] *b [i] [j] ;
```

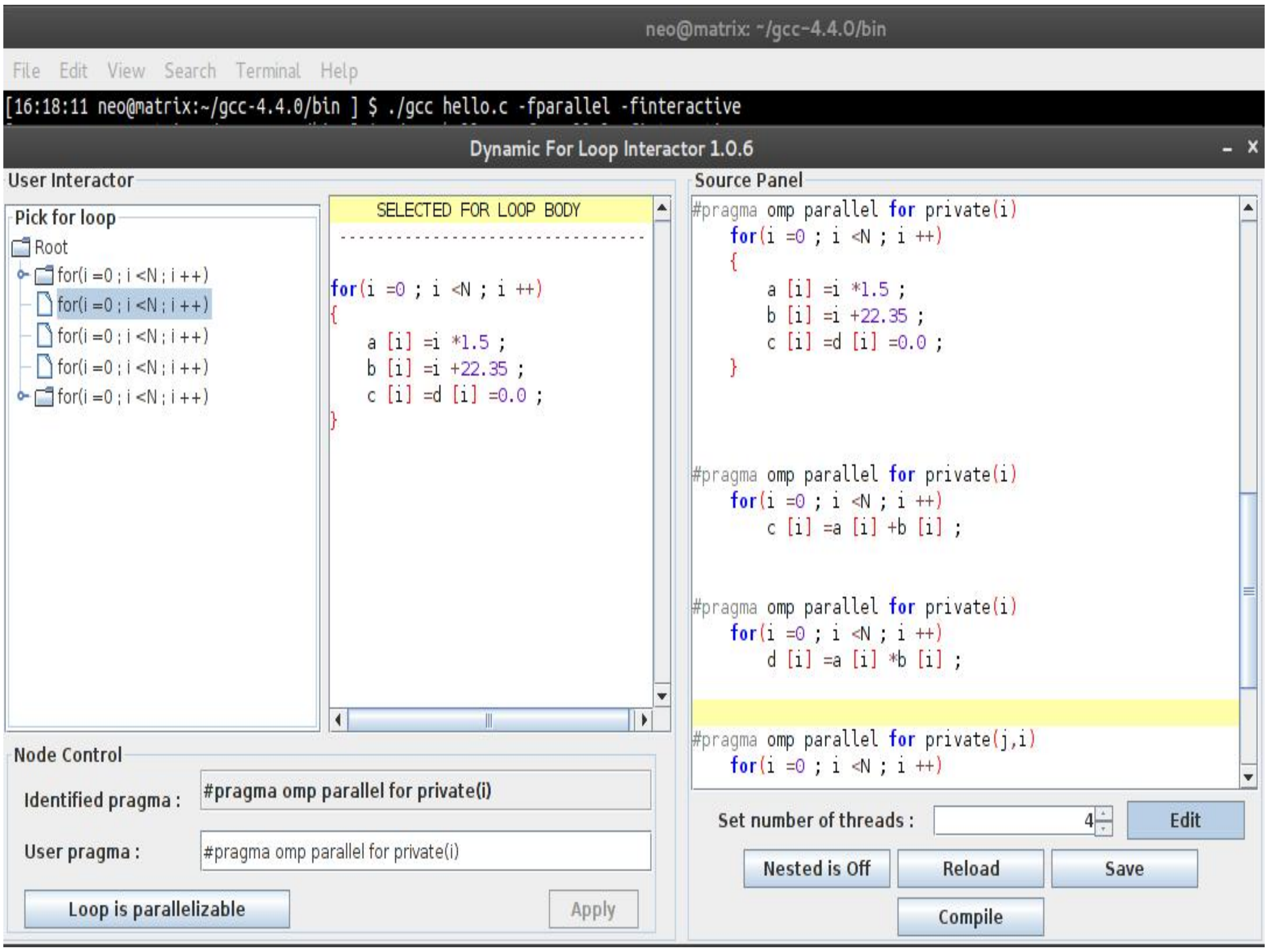
Identified pragma : #pragma omp parallel for private(j,i)

User pragma : #pragma omp parallel for private(j,i) schedule(dynamic)

Set number of threads : 8

6.5 SET NUMBER OF THREADS

The Control panel can be used to specify the number of threads to spawn for the current program. The number of cores in current executing system is automatically identified and initially it is set as number of threads to spawn.



```
[16:18:11 neo@matrix:~/gcc-4.4.0/bin ] $ ./gcc hello.c -fparallel -finteractive
```

```
for(i =0 ; i <N ; i ++)
for(i =0 ; i <N ; i ++)
for(i =0 ; i <N ; i ++)
for(i =0 ; i <N ; i ++)
for(i =0 ; i <N ; i ++)
```

```
SELECTED FOR LOOP BODY
--------------------------------

for(i =0 ; i <N ; i ++)
{
    a [i] =i *1.5 ;
    b [i] =i +22.35 ;
    c [i] =d [i] =0.0 ;
}
```

```
#pragma omp parallel for private(i)
    for(i =0 ; i <N ; i ++)
    {
        a [i] =i *1.5 ;
        b [i] =i +22.35 ;
        c [i] =d [i] =0.0 ;
    }


#pragma omp parallel for private(i)
    for(i =0 ; i <N ; i ++)
        c [i] =a [i] +b [i] ;


#pragma omp parallel for private(i)
    for(i =0 ; i <N ; i ++)
        d [i] =a [i] *b [i] ;


#pragma omp parallel for private(j,i)
    for(i =0 ; i <N ; i ++)
```

Identified pragma : #pragma omp parallel for private(i)

User pragma : #pragma omp parallel for private(i)

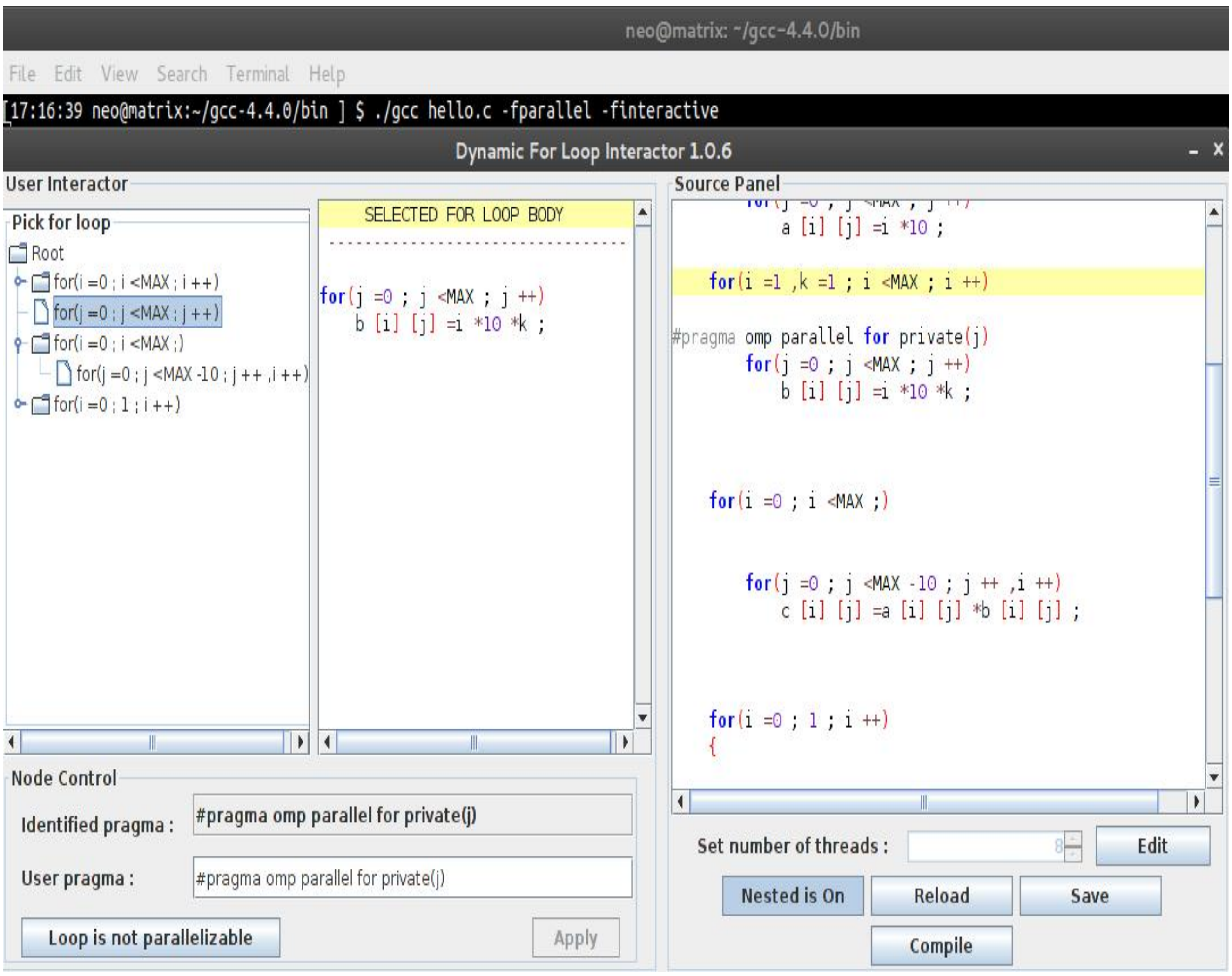
Set number of threads : 4

6.6 IDENTIFICATION OF NON-PARALLEIZABLE *FOR* LOOPS

OpenMP specification does not allow parallelization of *for loops* in certain format.

6.6.1 Case1: more than one initialization statements

The loops with more than one initialization statements are marked as not parallelizable and are ignored from further processing automatically.



```
[17:16:39 neo@matrix:~/gcc-4.4.0/bin ] $ ./gcc hello.c -fparallel -finteractive
```

Pick for loop tree:

```
Root
  for(i =0 ; i <MAX ; i ++)
    for(j =0 ; j <MAX ; j ++)
  for(i =0 ; i <MAX ;)
    for(j =0 ; j <MAX -10 ; j ++ ,i ++)
  for(i =0 ; 1 ; i ++)
```

Selected for loop body:

```
for(j =0 ; j <MAX ; j ++)
    b [i] [j] =i *10 *k ;
```

Source panel:

```
        a [i] [j] =i *10 ;

    for(i =1 ,k =1 ; i <MAX ; i ++)

#pragma omp parallel for private(j)
        for(j =0 ; j <MAX ; j ++)
            b [i] [j] =i *10 *k ;



    for(i =0 ; i <MAX ;)


        for(j =0 ; j <MAX -10 ; j ++ ,i ++)
            c [i] [j] =a [i] [j] *b [i] [j] ;



    for(i =0 ; 1 ; i ++)
    {
```

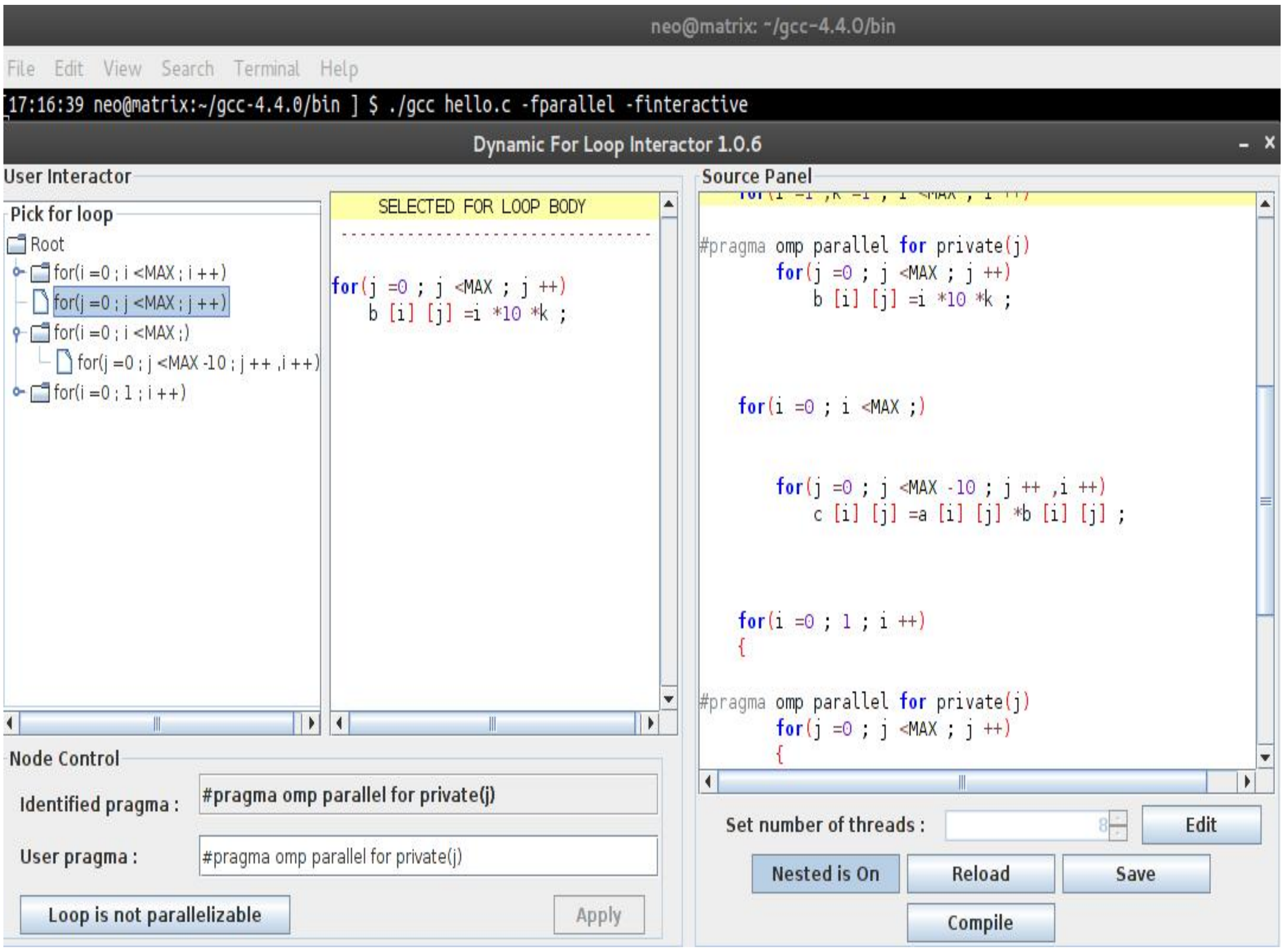
Identified pragma : #pragma omp parallel for private(j)

User pragma : #pragma omp parallel for private(j)

Set number of threads : 8

6.6.2 Case2: No incremental expression/ More than one incremental expression

The number of incremental operation must be exactly one for a parallelizable for loop.



```
[17:16:39 neo@matrix:~/gcc-4.4.0/bin ] $ ./gcc hello.c -fparallel -finteractive
```

Pick for loop:

```
Root
  for(i =0 ; i <MAX ; i ++)
    for(j =0 ; j <MAX ; j ++)
  for(i =0 ; i <MAX ;)
    for(j =0 ; j <MAX -10 ; j ++ ,i ++)
  for(i =0 ; 1 ; i ++)
```

Selected for loop body:

```
for(j =0 ; j <MAX ; j ++)
    b [i] [j] =i *10 *k ;
```

Source Panel:

```
#pragma omp parallel for private(j)
      for(j =0 ; j <MAX ; j ++)
          b [i] [j] =i *10 *k ;



   for(i =0 ; i <MAX ;)


      for(j =0 ; j <MAX -10 ; j ++ ,i ++)
          c [i] [j] =a [i] [j] *b [i] [j] ;



   for(i =0 ; 1 ; i ++)
   {

#pragma omp parallel for private(j)
      for(j =0 ; j <MAX ; j ++)
      {
```

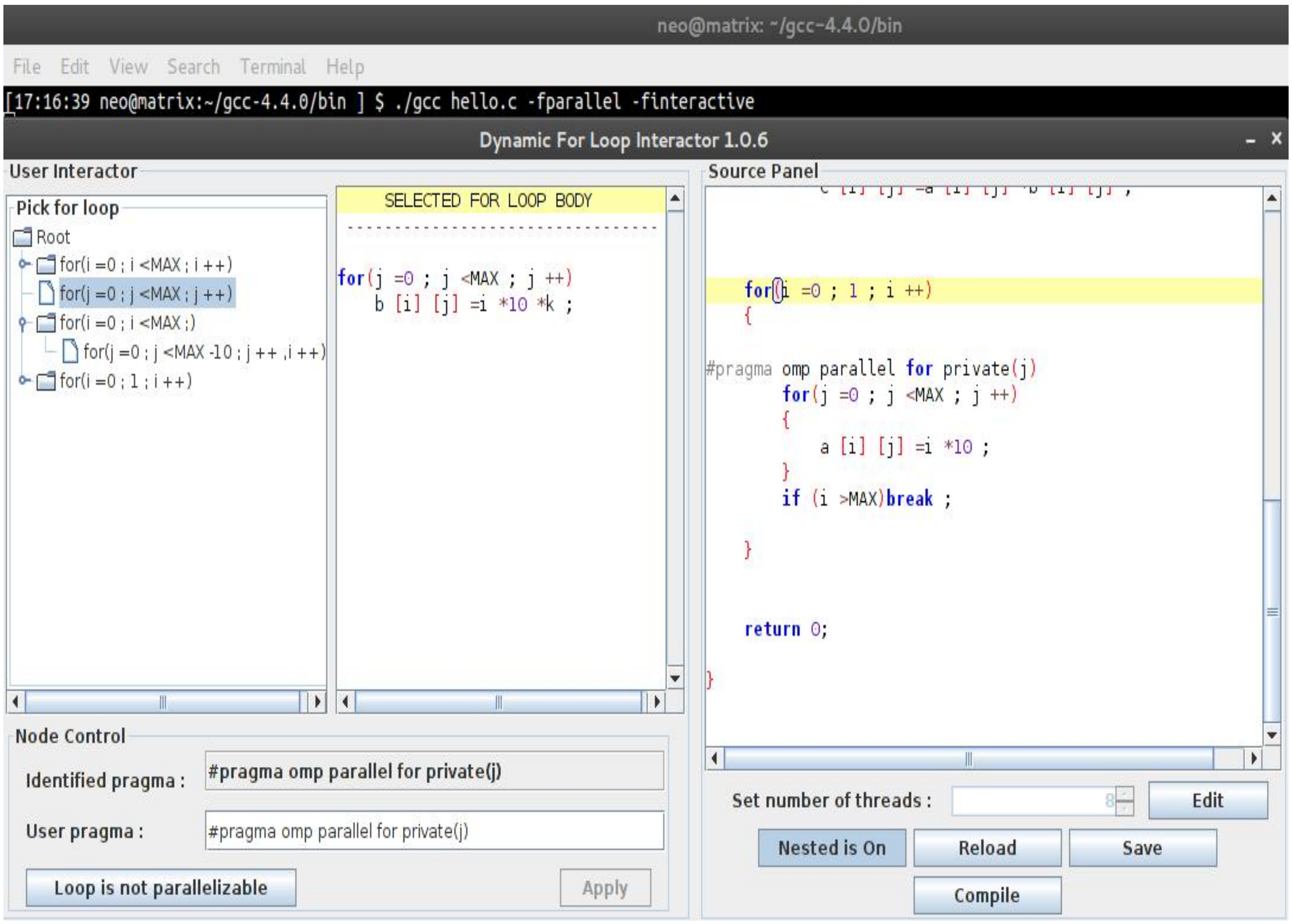
Identified pragma : #pragma omp parallel for private(j)

User pragma : #pragma omp parallel for private(j)

Set number of threads : 8

6.6.3 Case3: No conditional Expressions

Conditional Expression must be present to be able to parallelize a for loop.


neo@matrix: ~/gcc-4.4.0/bin
File Edit View Search Terminal Help

```
[17:16:39 neo@matrix:~/gcc-4.4.0/bin ] $ ./gcc hello.c -fparallel -finteractive
```

Dynamic For Loop Interactor 1.0.6
User Interactor
Pick for loop
Root

```
for(i =0 ; i <MAX ; i ++)
    for(j =0 ; j <MAX ; j ++)
for(i =0 ; i <MAX ;)
    for(j =0 ; j <MAX -10 ; j ++ ,i ++)
for(i =0 ; 1 ; i ++)
```

```
SELECTED FOR LOOP BODY
-----------------------------------

for(j =0 ; j <MAX ; j ++)
    b [i] [j] =i *10 *k ;
```

```
    for(i =0 ; 1 ; i ++)
    {

#pragma omp parallel for private(j)
        for(j =0 ; j <MAX ; j ++)
        {
            a [i] [j] =i *10 ;
        }
        if (i >MAX)break ;

    }


    return 0;

}
```

Identified pragma : #pragma omp parallel for private(j)

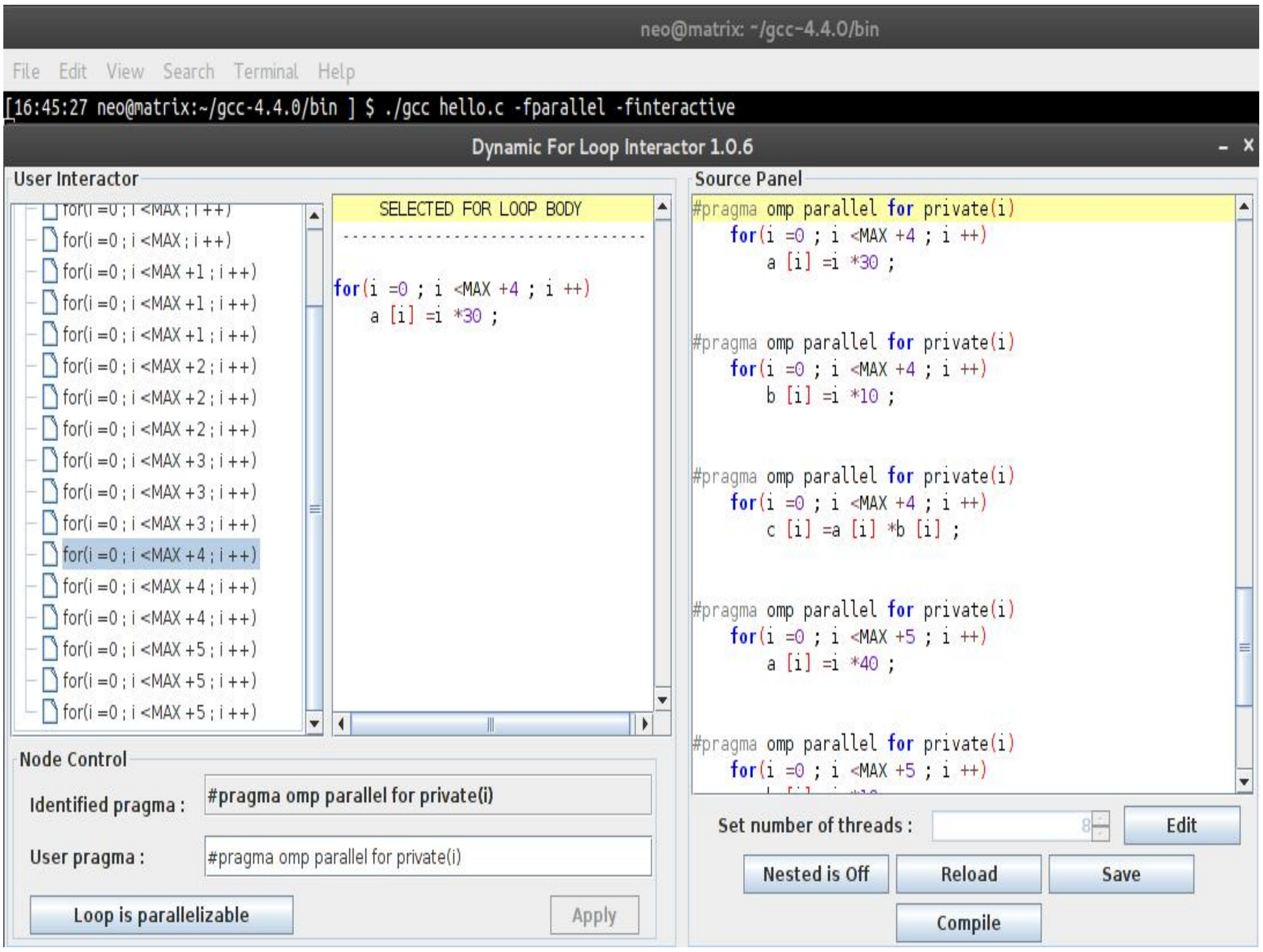
User pragma : #pragma omp parallel for private(j)

Loop is not parallelizable
Apply

Set number of threads : 8

6.7 USER ERGONOMICS

When there are lot of for loops to be analyzed, a convenient scrollbar is provided to scroll through the list of *for loop* nodes. When an individual node is clicked, in the source code panel, the cursor is moved to the corresponding position.



```
[16:45:27 neo@matrix:~/gcc-4.4.0/bin ] $ ./gcc hello.c -fparallel -finteractive
```

```
for(i =0 ; i <MAX ; i ++)
for(i =0 ; i <MAX ; i ++)
for(i =0 ; i <MAX +1 ; i ++)
for(i =0 ; i <MAX +1 ; i ++)
for(i =0 ; i <MAX +1 ; i ++)
for(i =0 ; i <MAX +2 ; i ++)
for(i =0 ; i <MAX +2 ; i ++)
for(i =0 ; i <MAX +2 ; i ++)
for(i =0 ; i <MAX +3 ; i ++)
for(i =0 ; i <MAX +3 ; i ++)
for(i =0 ; i <MAX +3 ; i ++)
for(i =0 ; i <MAX +4 ; i ++)
for(i =0 ; i <MAX +4 ; i ++)
for(i =0 ; i <MAX +4 ; i ++)
for(i =0 ; i <MAX +5 ; i ++)
for(i =0 ; i <MAX +5 ; i ++)
for(i =0 ; i <MAX +5 ; i ++)
```

```
SELECTED FOR LOOP BODY
--------------------------------

for(i =0 ; i <MAX +4 ; i ++)
    a [i] =i *30 ;
```

```
#pragma omp parallel for private(i)
    for(i =0 ; i <MAX +4 ; i ++)
        a [i] =i *30 ;


#pragma omp parallel for private(i)
    for(i =0 ; i <MAX +4 ; i ++)
        b [i] =i *10 ;


#pragma omp parallel for private(i)
    for(i =0 ; i <MAX +4 ; i ++)
        c [i] =a [i] *b [i] ;


#pragma omp parallel for private(i)
    for(i =0 ; i <MAX +5 ; i ++)
        a [i] =i *40 ;


#pragma omp parallel for private(i)
    for(i =0 ; i <MAX +5 ; i ++)
```

Identified pragma : #pragma omp parallel for private(i)

User pragma : #pragma omp parallel for private(i)

Loop is parallelizable

Set number of threads : 8

CHAPTER 7

RESULTS and ANALYSYS

Several real life examples have been taken as input and parallelized output performance is recorded.

7.1 GENERATING SUBSETS:

In mathematics, especially in set theory, a set A is a subset of a set B, or equivalently B is a superset of A, if A is "contained" inside B, that is, all elements of A are also elements of B. Generating subsets of a large set takes up a lot of time when it is run sequentially. Parallelizing this programs yields the following results.

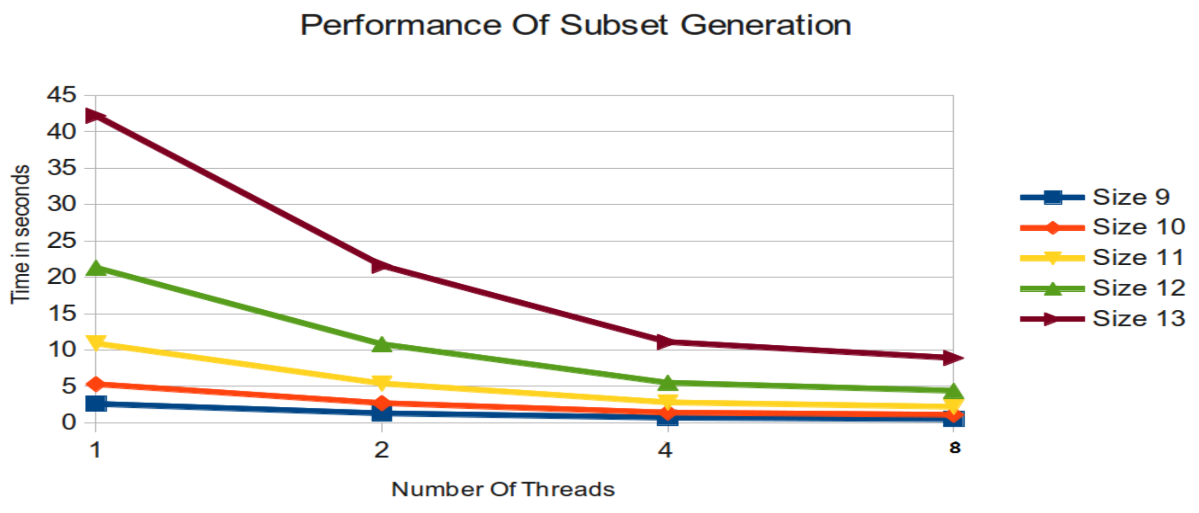


Fig 7.1.1 Performance of subset generation

We can observe that as we increase the number of threads for the processing, there is a significant amount of decrease in time taken.

Threads \ Subset size	9	10	11	12	13
1	2.6	5.3	10.9	21.3	42.2
2	1.3	2.7	5.4	10.8	21.6
4	0.7	1.4	2.8	5.5	11.1
8	0.5	1.1	2.2	4.4	8.9

Fig 7.1.2 Table of subset generation statistics

7.2 ARRAY SUM CALCULATION

Finding the sum of the array is another embarrassingly parallel problem, where the sum of all elements in an array has to be calculated. The performance graph of this has been plotted below

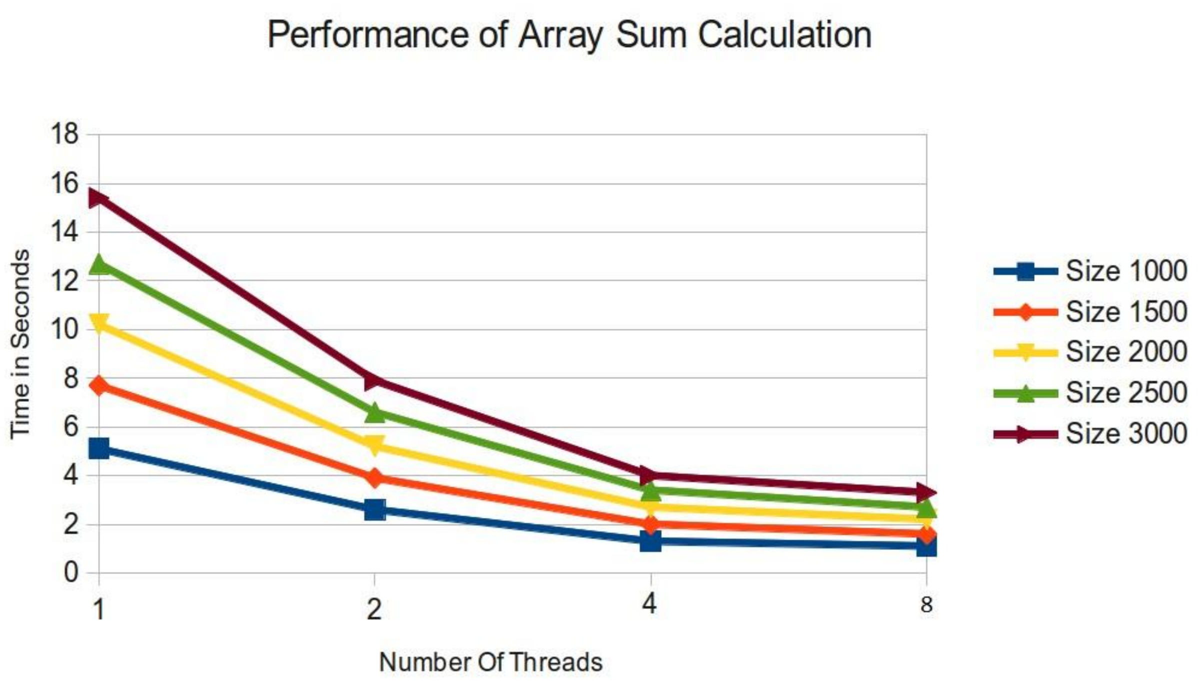


Fig 7.2.1 Performance of Array sum calculation

The table of statistic values have been represented below. The array size versus thread number table values are described.

Array size / Threads	1000	1500	2000	2500	3000
1	5.1	7.7	10.2	12.7	15.4
2	2.6	3.9	5.2	6.6	7.9
4	1.3	2	2.7	3.4	4
8	1.1	1.6	2.2	2.7	3.3

Fig 7.2.2 Table of array sum statistics

7.3 PRIME NUMBER GENERATION

A prime number (or a prime) is a natural number greater than 1 that has no positive divisors other than 1 and itself. In this example the total number of prime numbers under the specified upper bound is calculated. And suitable performance graph is drawn.

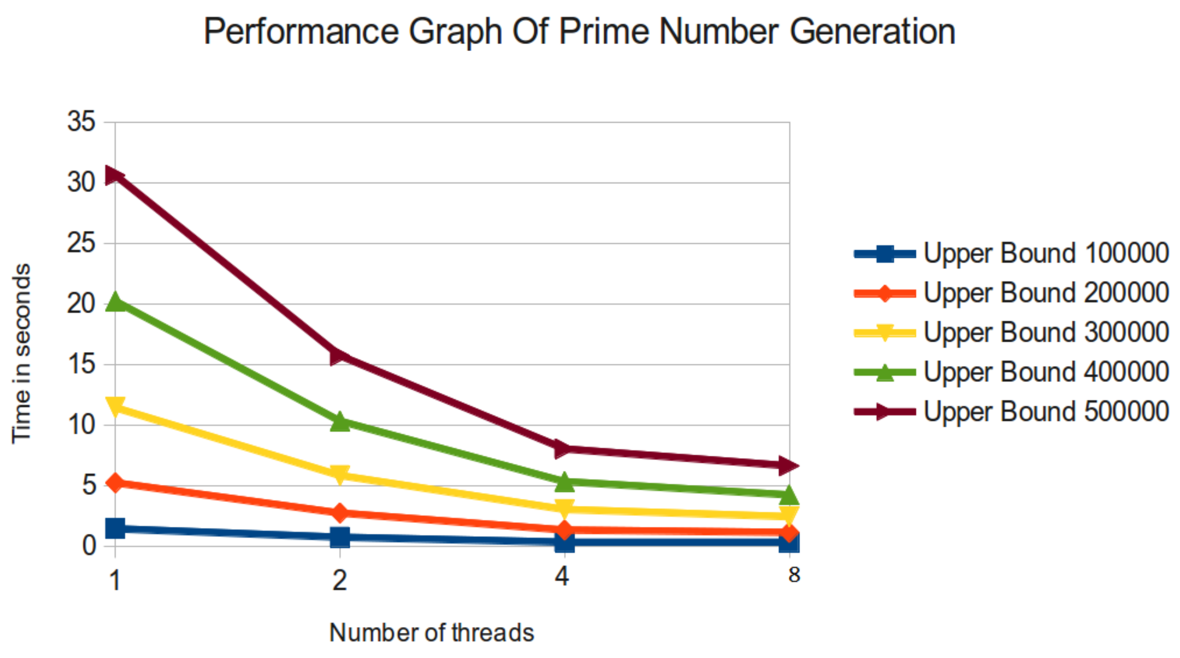


Fig 7.3.1 Performance of prime number generation

The table shows the amount of time taken to calculate the number of prime numbers under the specified upper bound

UpperBound / Threads	100000	200000	300000	400000	500000
1	1.4	5.2	11.4	20.2	30.6
2	0.7	2.7	5.8	10.3	15.7
4	0.3	1.3	3	5.3	8
8	0.3	1.1	2.4	4.2	6.6

Fig 7.3.2 Table of prime number performance

7.4 MATRIX MULTIPLICATION

Matrix multiplication is a classic example for embarrassingly parallel problems. In mathematics, matrix multiplication is a binary operation that takes a pair of matrices, and produces another matrix.

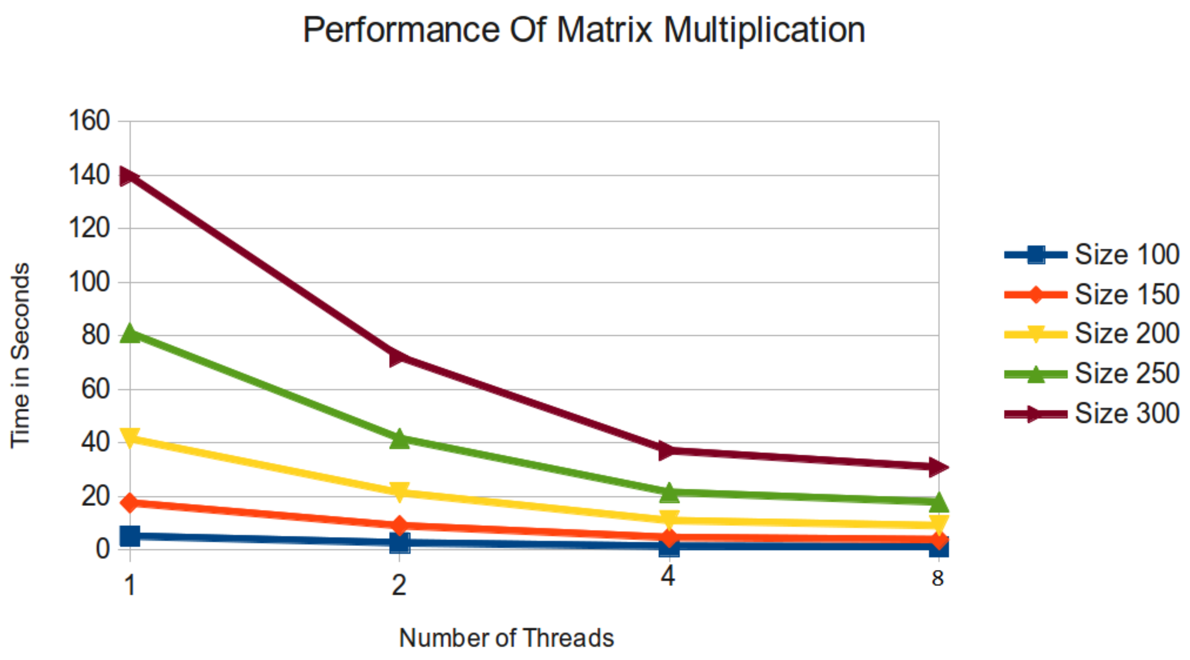


Fig 7.4.1 Performance Matrix Multiplication

The following table shows the statistics of matrix multiplication time taken.

Dimension / Threads	100	150	200	250	300
1	5.1	17.5	41.5	81	139.5
2	2.6	9	21.3	41.6	72
4	1.3	4.6	10.9	21.5	37
8	1.1	3.8	9	17.8	30.8

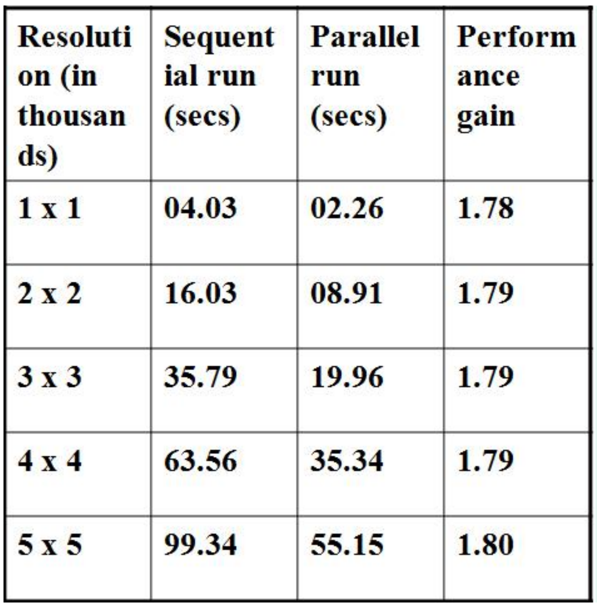
Fig 7.3.2 Table of Matrix Multiplication

7.5 FRACTAL GENERATION

Fractals are beautiful figures drawn using complex number formulae. Fractal generation problem falls under the category of embarrassingly parallel problem, for which little or no effort is required to separate the problem into a number of parallel tasks. There exists no dependency (or communication) between those parallel tasks.

Individual pixel values for a fractal image which has to be rendered have no dependencies with other pixel values. So the pixel values can be calculated independently without disturbing the overall calculation. Traditional Fractal generation algorithms can be effectively used to benchmark the performance because they can be easily parallelized and it requires lot of CPU time to generate fractal images of large dimensions.

A simple mandelbrot set is drawn with different resolutions and the time taken to generate the image is documented for both sequential and parallel run with two cores.



Resolution (in thousands)	Sequential run (secs)	Parallel run (secs)	Performance gain
1 x 1	04.03	02.26	1.78
2 x 2	16.03	08.91	1.79
3 x 3	35.79	19.96	1.79
4 x 4	63.56	35.34	1.79
5 x 5	99.34	55.15	1.80

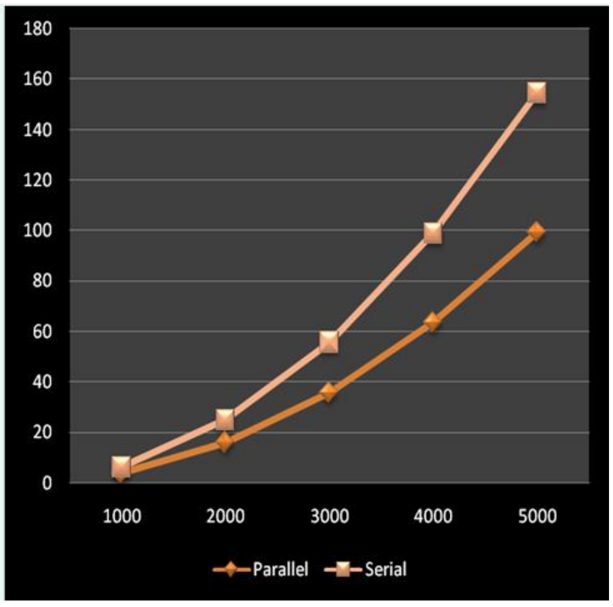
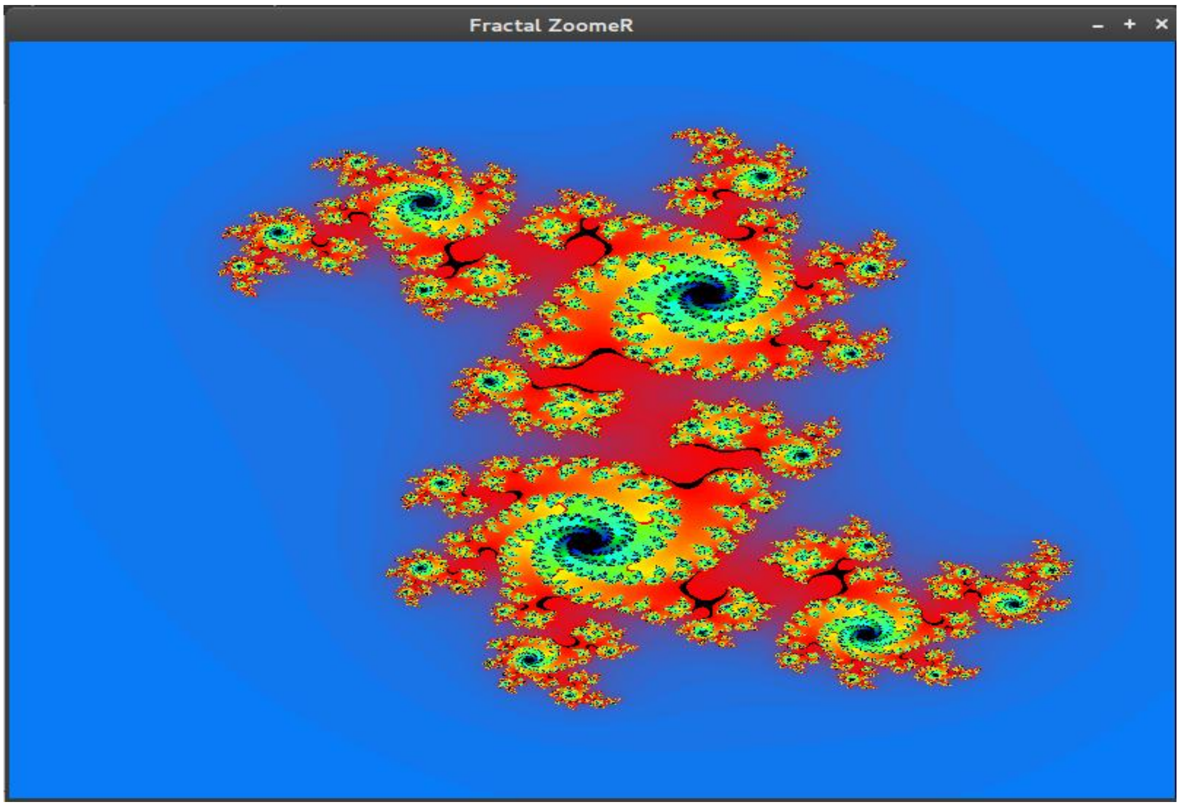
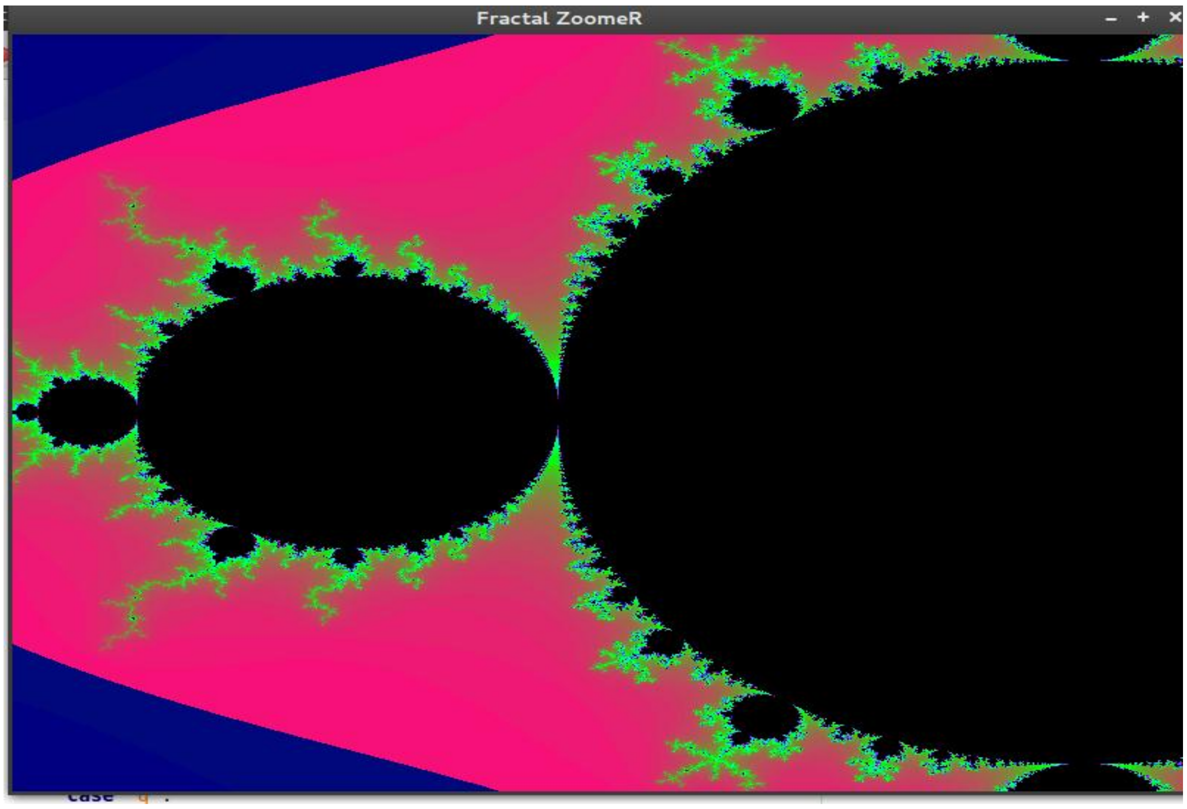


Fig 7.5.1 Fractal generation benchmark

Fractal Application screenshots:





7.6 INTER-FLOCK CROSSOVER

In problems where the search space is very large and heuristics fail drastically, genetic algorithm offers an advantage over other methods. Genetic Algorithms (GA), which replicates some of the nature’s laws to solve a problem, almost converges to the optimum solution within relatively less time. An approach of solving TSP using GA by applying different cross over methods is presented in the paper “Applying Inter-flock Crossover to Improve Convergence in Genetic Algorithm on Shared-Memory Multicore Computers”. In this paper, a unique multi-processor inter-flock crossover operator is used to improve the convergence. The comparisons of methods taken from standard TSPLIB (TSP library) have been benchmarked. The results obtained are very promising with 0% deviation from the optimum solution for nodes less than 100 and less than 0.3% for cases larger than 100 nodes. A new operator known as interflock crossover was presented that deployed multiple threads on multiple cores to further bring down the number of generations needed to arrive at the most optimum solution based on known solutions. The interflock operator was limited to two flocks as the number of cores in the available machines were just two.

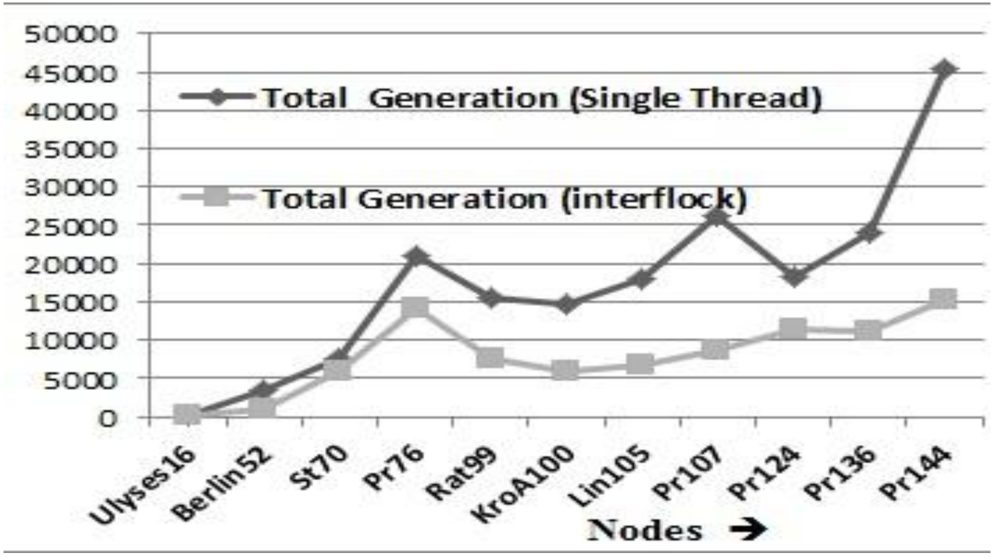


Fig 6.6.1 Bench marking of TSP Symmetric instances on 2 cores

CHAPTER 8

CONCLUSION

The goal of automatic parallelization is to relieve programmers from the tedious and error-prone manual parallelization process. Though the quality of automatic parallelization has improved in the past several decades, fully automatic parallelization of sequential programs by compilers remains a grand challenge due to its need for complex program analysis and the unknown factors (such as input data range) during compilation.

Automatic parallelization relieve programmers from the tedious and error-prone manual parallelization process. Already Parallelized problems serve as the template for future encounters of similar type of problems, thus code reusability can be achieved. In theory, throwing more resources at a task will shorten it’s time to completion, with potential cost savings. Parallel computers can be built from cheap, commodity components. Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer, especially given limited computer memory. A single compute resource can only do one thing at a time. Multiple computing resources can be doing many things simultaneously

REFERENCES

[1] Wu-chun Feng , Pavan Balaji
IEEE computer magazine December 2009 (page 26,36)

[2] Seung-Jai Min, “OpenMP tutorial” (page 2)

[3] Joseph D Sloan, “High Performance Linux Clusters” O’Relliy Publications.

[4] Parallel computing , www.wikipedia.org

[5] Jennifer-Ann Monique Anderson , “Automatic Computation and data decomposition for multiprocessors”

[6] en.wikipedia.org/wiki/Parallel_computing

[7] Gottlieb, Allan; Almasi, George S. (1989). Highly parallel computing. Redwood City, Calif.: Benjamin/Cummings. ISBN 0-8053-0177-1.

[8] Asanovic et al. Old [conventional wisdom]: Power is free, but transistors are expensive. New [conventional wisdom] is [that] power is expensive, but transistors are "free"

[9] Yajnesh T , Jickson P and Waseem Ahmed ,
“Applying Inter-flock Crossover to Improve Convergence in Genetic Algorithm on Shared-Memory Multicore Computers”, Proceedings of the international conference on “On Demand Computing”

[10] Asanovic, Krste et al. (December 18, 2006). "The Landscape of Parallel Computing Research: A View from Berkeley" (PDF). University of California, Berkeley. Technical Report No. UCB/EECS-2006-183.

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

[12] Barney, Blaise. "Introduction to Parallel Computing". Lawrence Livermore National Laboratory. Retrieved 2007-11-09.

[13] Hennessy, John L.; Patterson, David A. (2002). Computer architecture / a quantitative approach. (3rd ed.). San Francisco, Calif.: International Thomson. p. 43. ISBN 1-55860-724-2.

[14] Barney, Blaise. "Introduction to Parallel Computing". Lawrence Livermore National Laboratory. Retrieved 2007-11-09.

[15] Parallel Computing: Principles and Practice T. J. Fountain

[16] Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms
Ananth Grama,Anshul Gupta,George Karypis,Vipin Kumar

[17] http://www.springer.com/computer/swe/book/978-3-642-04817-3

[18] http://charm.cs.uiuc.edu/

[19] http://en.wikipedia.org/wiki/Automatic_parallelization

[20] http://gcc.gnu.org

[21] Programming on Parallel Machines; GPU, Multicore, Clusters and More
Professor Norm Matloff , University of California, Davis

[22] Patterns for Parallel Programming
Timothy G. Mattson , Beverly A. Sanders , Berna L. Massingill

[23] Fox, Geoffrey; Roy Williams, Paul Messina (1994). Parallel Computing Works!. Morgan Kaufmann. pp. 575, 593. ISBN 978-1-55860-253-3.

[24] Simone Campanoni, Timothy Jones, Glenn Holloway, Gu-Yeon Wei, David Brooks. "The HELIX Project: Overview and Directions". 2012.