Solutions by Optimization of the 2-Dimensional Heat Conductivity Problem on Grid Machines using C++ and OpenMP

Aliyu Sani Ahmad
Dept. of Comp. Science
Federal Univ, Wukari

Mu'awuya Dalhatu
Dept. of Comp. Science
Federal Univ. Wukari,

Emmanuel Masa-Ibi
Dept. of Comp. Science
Federal Univ. Wukari

Abdulrashid Aliyu Babando
ICT Unit, Advanced Manufacturing Tech

ABSTRACT
Optimization is a method of transforming a piece of code into another functionally equivalent piece of code to improve code quality and efficiency. The two most important characteristics are the speed and size of the code. Other characteristics include: energy required and the time it takes to compile the code. This paper uses source code (jacobi2d.c) to provide solutions by optimization of the 2-Dimentional heat conductivity problem on grid machine using C++ compiler and OpenMP (Open Multi-Processing). The paper discuss the benefits of parallel grid computing. The grid server on which this paper used has 10 computers, grid-01 to grid-08 have 2 cores and 2 Gigabytes memory while 09 and 10 have 8 cores and 8 Gigabytes memory. The process of solving rectangular 2-dimensional heat conductivity problem to optimized performance is done in number of steps from step 1 to step 4, optimization starts in step 3 and ends in step 4. In step 4, performance speedup of step 3 is compared with performance speedup of step 4 and optimal (result) performance is achieved in step 4. It is concluded that performance is optimized with parallel processing using grid machine with performance level -3 and large problem size.

General Terms
Performance Optimization, Serial Processing, Parallel Processing

Keywords
High performance computing, Optimization, Parallel Processing, thread, Grid, Heat Conductivity Problem

1. INTRODUCTION
Compilers are constantly improving in terms of the techniques they use to optimize the code. However, they're not perfect. Instead of spending time manually tweaking a program, it's usually more fruitful to use specific features provided by the compiler and let the compiler tweak the code. The four ways to help the compiler optimize your code more effectively include: (1) Write understandable and maintainable code. (2) Use compiler directives (3) Use compiler-intrinsic functions. And (4) Use Profile-Guided Optimization (PGO) [1] [15].

Parallel processing is the simultaneous use of multiple computer resources to solve a computational problem and state: A problem is broken into discrete parts that can be solved concurrently, each part is further broken down into a series of instructions, instructions from each part execute simultaneously on different processors, and an overall control mechanism is employed. Computational problem should: (1) Be broken into discrete pieces of work that can be solved simultaneously (2) Execute multiple program instructions at any moment and (3) Be solved in less time with multiple compute resources than with a single compute resource. The compute resources are typically: A single computer with multiple processors/cores and or an arbitrary number of such computers connected by a network. [1]

1.1 Parallel Computers
Virtually all stand-alone computers today are parallel from a hardware perspective, [1] they have multiple functional units (L1 cache, L2 cache, branch, prefetch, decode, floating-point, graphics processing (GPU), integer, etc.), they have Multiple execution units/cores and are typically multiple hardware threads [1]**.** The real world in its physical and logical, is massively parallel: In the natural world, many complex, interrelated events are happening at the same time, yet within a temporal sequence. Compared to serial computing, parallel computing is much better suited for modelling, simulating and understanding complex real world phenomena. For instance, imagine modelling Galaxy formation, planetary movement, climate change, rush hour traffic, plate tectonic, weather etc. serially, it is barely impossible or if possible it will be going to be time consuming of back breaking task. Parallel computing has many benefits such as: (1) Save Time and Money: Parallel cluster can be built from cheap commodity components because in theory, throwing more resources at a task will shorten its time to completion, with potential cost savings [2]. (2) Solve Larger / More Complex Problems: Many problems are so large and/or complex that it is impractical or impossible to solve them on a single computer with limited memory. (3) Provide Concurrency: A single compute resource can only do one thing at a time. Multiple compute resources can do many things simultaneously. (4) Make better use of Underlying Parallel Hardware: Modern computers are parallel in architecture with multiple processors/cores. Parallel software is specifically intended for parallel hardware with multiple cores, threads, etc. In most cases, serial programs run on modern computers "waste" potential computing power.

1.2 Two Dimensional Heat Conduction
Two dimensional heat conduction refers to two processes of heat conduction that is taking take place: the generation of heat inside the body and the heat transfer between the body and its environment. Therefore, the total amount of heat dQ in a specific dV (Volume differential) corresponds to the sum of both terms [21].

1.3 Standard Formulation of Heat Conduction
Heat conduction is the transfer of internal thermal energy by the collisions of microscopic particles and movement of

electrons within a body. The microscopic particles in the heat conduction can be molecules, atoms, and electrons. Internal energy includes kinematic and potential energy of microscopic particles. Heat conduction is governed by the law of heat conduction, which is also called the Fourier's law: the time rate of heat transfer through a material is proportional to the negative gradient in the temperature and to the area. Under a Cartesian coordinate system, the Fourier's law for the rate of heat transfer by conduction is expressed as, (9.1).

Where 9.1 are the rate of heat transfer, 9.1 are the heat flux, and kx, ky, and kz are the conductivity coefficients along x-, y-, and z-axis, and are the cross-section areas, and are the temperature gradients along three axes of the Cartesian coordinate system [12] [17].

1.4 Grid and Cloud Computing

Cloud computing refers to a variety of services available over the Internet that deliver compute functionality on the service provider's infrastructure such as: Google Apps, Amazon EC2, or Salesforce etc. A cloud environment may actually be hosted on either a grid or utility computing environment. On the other hand, a computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities [2]. Grids enable access to **shared** computing power and storage capacity from desktop computers. [2] Virtualization is a key enabler of cloud computing. The most prominent feature is the ability to install multiple Operating System on different virtual machines on the same physical machine, this provides the additional benefits of overall cost reduction owing to the use of less hardware and consequently less power and increases machine utilization [16]. Grid and cloud computing are networks which abstract processing tasks [3]. Abstraction masks the actual complex processes taking place within a system, and presents a user with a simplified interface with which they can interact easily. The idea is to be able to make the system more user-friendly whilst retaining all the benefits of more complicated processes [3] [16]. Although there is a difference in the fundamental concepts of grid and cloud computing that does not necessarily mean they are mutually exclusive; it is quite feasible to have a cloud within a computational grid, as it is possible to have a computational grid as a part of a cloud. Grids promised to deliver computing power on demand. However, despite a decade of active research, no viable commercial grid computing provider has emerged. On the other hand, it is widely believed especially in the Business World that HPC will eventually become a commodity [14].

1.5 Open Multi-Processing

OpenMP is a Short for Open Multi-Processing it is used in various application such as multi-threaded parallel processing, use on shared-memory multi-processor (core) computers. It is also used where part of program is a single thread and part is multi-threaded [5]. OpenMP has three components namely: directives, runtime library and environment variables [5].

1.5.1 Compiler Directives

OpenMP standard is a compiler directive driven parallel programming system. It uses a fork-join model, so relies on a (logically) shared memory system and a global data model [1]. Compiler directives appear as comments in the source code and are ignored by compilers unless otherwise stated usually by specifying the appropriate compiler flag. The OpenMP compiler directives was used for spawning a parallel region, dividing blocks of code among threads, distributing loop iterations between threads, serializing sections of code and synchronization of work among threads. The syntax for compiler directives used (see appendix II).

1.5.2 Run-time Library Routines

Runtime library is a collection of software programs used at program runtime to provide one or more native program functions or services [17]. The runtime library enables a software program to be executed with its complete functionality and scope by providing add-on program resources that are essential for the primary program. OpenMP API includes number of run-time library routines. These routines are used for a variety of purposes such as: Setting and querying the number of threads, Querying a thread's unique identifier (thread ID), Querying a thread's ancestor's identifier, initializing and terminating locks and nested locks and Querying wall clock time and resolution etc.

The run-time library routines used such as header file **(<omp.h>), omp_get_wtime** to provides a portable wall clock timing routine, and **omp_set_num_threads:** to sets the number of threads that will be used in the next parallel region (see Appendix II)

1.5.3 Environment Variables

OpenMP environment variables were used for controlling the execution of parallel code at run-time. The environment variables were used to set the number of threads, specified how loop iterations are divided, bind threads to processors, enabled/disabled nested parallelism; setting the maximum levels of nested parallelism, enabled/disabled dynamic threads, set thread stack size, and set thread wait policy

1.5.4 Critical Directive

This directive specifies a region of code that must be executed by only one thread at a time. If a thread is currently executing inside a critical region and another thread reaches that critical region and attempts to execute it, it will block until the first thread exits the critical region. For the critical command used (see Appendix II)

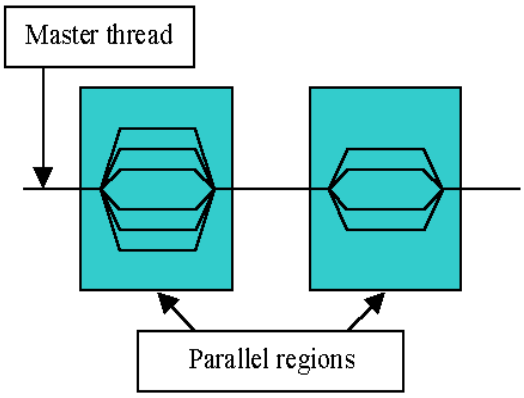

Fig. 1: shows OpenMP Master Thread and Parallel region Source: [www.dartmouth.edu/classes/intro_openmp]

1.6 Memory Architectures and Parallel Programming

OpenMP memory architecture and parallel programming. Memory can be distributed or shared. In distributed memory: each processor has its own memory and parallel programming by message passing (MPI)

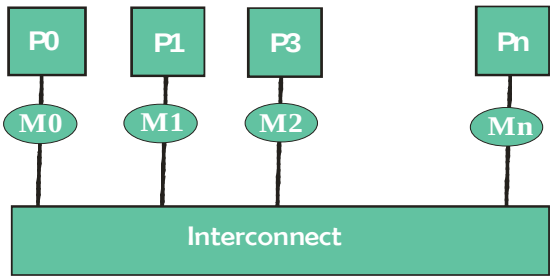

Fig 2: Distributed Memory Architecture Source: [www.dartmouth.edu/classes/intro_openmp]

In shared memory: processors share memory in two parallel programming approaches known as: message passing (MPI) and directives-based interface.

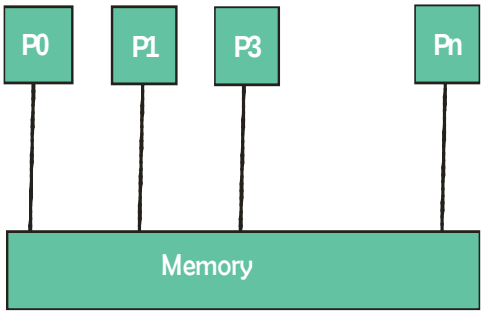

Fig 3: Shared Memory ArchitecturevSource: [www.dartmouth.edu/classes/intro_openmp]

OpenMP programs accomplish parallelism exclusively through the use of threads. A thread of execution is the smallest unit of processing that can be scheduled by an operating system. This exist within the resources of a single process. Without the process, they stop to exist. However, the number of threads match the number of machine processors/cores [5] [19].

1.7 Optimization Level

It is argue that the performance of a code must be measured on a dedicated system [13] as used in this paper**,** no other user can start a process and the user measuring the performance only runs the minimum amount of processes [13]. All modern compilers perform some automatic optimization when generating code. Most compilers provide several levels of optimization namely: **-O0**: No optimization, **-O1:** Optimization Level 1, **-O2:** Optimization level 2 etc. The higher the optimization level the higher the probability that a debugger may have trouble dealing with the code depending on the compiler. Some compiler documentation may indicate that higher levels of (aggressive) optimization should be used with caution. Therefore this paper uses –O1 to –O3 (Optimization level 1 to 3) [20].

2. LITERATURE REVIEW

2.1 Optimization

The desire for optimization is inherent to humans. The search for extremes inspires scientists, mathematicians and the human being in general. Methods of optimization explore suppositions about the nature of responses to the target function, by varying parameters and suggesting the best way to change them. The variety of a priori suppositions corresponds to the variety of optimization methods [2]. Optimization or mathematical programming is the selection of a best element from some set of available alternatives. An optimization problem consists of maximizing or minimizing a real function by systematically choosing input values from within an allowed set and computing the value of the function [3]. There are many ways to improve performance: (1) Buy Faster Hardware: the amount of hardware to buy would be staggering but better to achieve the same effect by modifying the Code. (2) Modify the algorithm: For example: Searching for an element in a sorted array, the implementation is based on a linear search which is easy to write but when performance becomes an issue, is to replace the linear search implemented with a binary search that demands more complex code (3) Modify the implementation: in doing this, you don't change the spirit of the algorithm but rather you shuffle lines of code around by modifying the instruction in a different order, also modify the code organization. (4) Use concurrency: Multi-threaded code on a single-CPU machine to utilize hardware resources more effectively and multi-threaded code on a multi-CPU/multi-core machine. Time as a performance measurement, show the time between the start and the end of an operation which is called run time, wall-clock time, execution time etc. example a *program takes 12.5s on a Pentium 3.5GHz.* Execution time must be measured on a dedicated machine because it is used often so that performance can be independent of the (memory) "size" of the application [12]. For example: compressing a 1MB file takes 1 minute and compressing a 2MB file takes 2 minutes. "The performance is the same". Performance Rates: is measured by Timing a section of code, count how many operations are in that section of the code and compute the rate as the number of items divided by the measured time. Example:

start_stopwatch();

for (i=0; i<1000000; i++)

x = y + z * a

stop_stopwatch()

- Number of Flop: 2million (1M additions, 1M multiplications)
- Number of MFlops = 2 / time.

2.2 High Performance Computing (HPC)

High Performance Computing (HPC) has become an essential tool in every researcher's arsenal. Most research problems nowadays can be simulated, clarified or experimentally tested by using computational simulations. Since most researchers have little-to-no knowledge in low-level computer science, they tend to look at computer programs as extensions of their minds and bodies instead of completely autonomous systems. Since computers don't work the same way as humans, the result is usually Low Performance Computing where HPC would be expected [4]. High performance means how fast a machine and/or a code will run, Performance conflicts with other concerns: (1) Correctness: aggressive optimisation can cause problems. (2) Readability: more code is harder to follow and modularity can hurt performance with its overhead. (3) Portability: code that is fast on machine A can be slow on machine B at the extreme, highly optimised code is not portable at all, and in fact is done in hardware.

Inverse heat conduction problem has a very wide application background. It has been applied in almost all fields of scientific engineering, including power engineering, aerospace engineering, metallurgical engineering, biomedical engineering, chemical engineering, nuclear physics, geometry optimization of equipment, and non-destructive testing. In this paper, the authors adopts Finite Difference Method (FDM)

and Model Predictive Control Method (MPCM) to study the inverse problem in the third-type boundary heat-transfer coefficient involved in the two-dimensional unsteady heat conduction system. They introduced the residual principle to estimate the optimized regularization parameter in the model prediction control method, thereby obtaining a more precise inversion result. They uses Finite Difference Method (FDM) for direct problem to calculate the temperature value in various time quanta of needed discrete point as well as the temperature field verification by time quantum, they uses inverse problem to discuss the impact of different measurement errors and measurement point positions on the inverse result. As demonstrated in their empirical analysis, they states that the proposed method remains highly precise despite the presence of measurement errors or the close distance of measurement point position from the boundary angular point angle [6].

Inverse Heat Transfer Problems (IHTP) are a significant class of inverse problems. In a heat transfer problem, if the boundary conditions, the thermo-physical properties, the geometrical configuration of the heated body, and the applied heat flux are all known but the temperature distribution is unknown, the heat transfer problem is referred to as the direct heat transfer problem. Contrary to the direct heat transfer problem, the inverse heat transfer problem is concerned with the determination of the boundary conditions, the thermo physical properties, the geometrical configuration of the heated body, and the applied heat flux by knowing the temperature distribution on some part of the heat conducting body boundary. Such problems are ill-posed. The difficulty in solving ill-posed problems results from the fact that they are inherently unstable and very sensitive to noise and perturbation. In this paper, the authors states that, the development of new numerical schemes, their numerical implementation, and their efficiency improvement is of crucial importance. Their aim is to estimate the thermal conductivity, the heat transfer coefficient, and the heat flux in irregular bodies (both separately and simultaneously) using a two dimensional inverse analysis. Their numerical procedure consists of an elliptic grid generation technique to generate a mesh over the irregular body and solve for the heat conduction equation. Their paper describes a novel sensitivity analysis scheme to compute the sensitivity of the temperatures to variation of the thermal conductivity, the heat transfer coefficient, and the heat flux. The sensitivity analysis scheme allows for the solution of inverse problem without requiring solution of adjoin equation even for a large number of unknown variables. They use conjugate gradient method (CGM) to minimize the difference between the computed temperature on part of the boundary and the simulated measured temperature distribution. Their results reveal that the proposed algorithm is very accurate and efficient [7].

In a paper reviewed topology optimization of conductive heat transfer problems using parametric L-systems states that, to prevent overheating, of electronic devices that are packed in increasingly compact space which increases the heat density generated by their component, their architecture must be designed with an effective cooling system. They assert that the first task of the cooling system is to conduct the heat from the electronic components to a heat sink, using highly conductive material, e.g. copper or aluminium. They shows that the availability of conductive material is limited by space constraints and because the manufacturers always wish to reduce the cost of such components. Consequently, properly distributing the high conductivity material through a finite volume becomes an important topology optimization problem. It is stated that, generative encodings have the potential of improving the performance of evolutionary algorithms. They apply parametric L-systems, which can be described as developmental recipes, to evolutionary topology optimization of widely studied two-dimensional steady-state heat conduction problems. They translate L- systems into geometries using the turtle interpretation, and evaluate their objective functions, i.e. average and maximum temperatures, using the Finite Volume Method (FVM). The method requires two orders of magnitude fewer function evaluations, and yields better results in 10 out of 12 tested optimization problems (the result is statistically significant), than a reference method using direct encoding. Their results indicate that the method yields designs with lower average temperatures than the widely used and well established SIMP (Solid Isotropic Material with Penalization) method in optimization problems where the product of volume fraction and the ratio of high and low conductive material is less or equal to 1. They demonstrate the ability of the methodology to tackle multi-objective optimization problems with relevant temperature and manufacturing related objectives [8].

In a paper that solve problem of optimal design of cooling elements in modern battery systems based on mathematical model and optimization problem, they consider a simplified model of the problem based on some set of assumptions. They consider a simple model of two-dimensional steady-state heat conduction described by elliptic partial differential equations and involving a one dimensional cooling element represented by a contour on which interface boundary conditions are specified. They states, the problem consists in finding an optimal shape of the cooling element which will ensure that the solution in a given region is close (in the least squares sense) to some prescribed target distribution. They formulate the problem as PDE-constrained optimization and the locally optimal contour shapes are found using a gradient-based descent algorithm in which the Sobolev shape gradients are obtained using methods of the shaped differential calculus. The main novelty of their work is an accurate and efficient approach to the evaluation of the shape gradients based on a boundary-integral formulation which exploits certain analytical properties of the solution and does not require grids adapted to the contour. Their approach is thoroughly validated and optimization results obtained in different test problems exhibit nontrivial shapes of the computed optimal contours [9].

In a paper "The basic Heat Transfer Search (HTS) Algorithm", the authors considers only one of the modes of heat transfer (conduction, convection, and radiation) for each generation. In their proposed algorithms, the system molecules are considered as the search agents that interact with each other as well as with the surrounding to a state of the thermal equilibrium. Another improvement is the integration of a population regenerator to reduce the probability of local optima stagnation. The population regenerator is applied to the solutions without improvements for a pre-defined number of iterations. The feasibility and effectiveness of their proposed algorithms are investigated in 23 classical benchmark functions and 30 functions extracted from the CEC2014 test suite. They also, solve two truss design problems to demonstrate the applicability of the proposed algorithms. Their results show that the IHTS algorithm is more effective as compared to the HTS algorithm. Moreover, the Improved Heat Transfer Search Algorithm (IHTS algorithm) provides very competitive results as compared to the existing meta-heuristics [10].

A paper aims at improving the performance of a smart antenna by optimizing the radiation pattern using various approaches. It is discover that high side lobe levels in a radiation pattern often lead to unwanted patterns of radiation, energy wastage and reduction in the overall performance of the antenna. Optimizing the radiation pattern was done by obtaining the optimum weights that give a radiation pattern with reduced side lobe level (SLL). They used Least Mean Squares (LMS) Algorithm and Genetic Algorithm (GA) to determine the optimal antenna parameters that would minimize side lobe level. They carried out simulations to determine the effect of increase in inter-element spacing on array factor and beam-width using the optimal antenna parameters. From the results obtained for the same number of elements, Least Mean Square gave the better result in form of a more reduced beam-width while Genetic Algorithm performed better for the reduction of the side lobe level. This translates to the reduction of radiated power wasted in side lobes for linear arrays in antenna systems [11].

3. AIM AND OBJECTIVES

The aim of this paper is to provide solutions by optimization using C++ compiler, OpenMP and serial version of jacobi2d source code that solves 2-dimensional heat conductivity problem on grid machines to optimize in order to speed up the performance of the code. The objectives are as follows:

a) To modify the source code to reflect the boundary conditions for a rectangular 2D problem set at top 20oC, bottom 60oC, left 10oC and right 80oC and compute the temperature distribution with a range of problem sizes, report the execution time and record the run-time of the code using different levels of compiler optimization.

b) To transform the serial version to parallel by including timers in order to report the parallel run-time and test the OpenMP version to establish correct operation using 1, 2, 3 and 4 threads/processors, regardless of performance.

c) To use grid machines to run performance tests with the OpenMP implemented and increase the problem size to provide sufficient work that demonstrate useful speedup.

d) To provide speedup results for a range of problem sizes using up to 8 threads using optimizations level.

e) To further modify the OpenMP application to improve the parallel performance that would provide results that permit comparison with the one previously obtained.

4. PROBLEM STATEMENT

The generalization of optimization theory and techniques to other formulations comprises a large area of applied mathematics. Optimization or mathematical programming is the selection of a best element from some set of available alternatives. An optimization problem consists of maximizing or minimizing a real function by systematically choosing input values from within an allowed set and computing the value of the function. Most of the related literature reviewed uses mathematical modelling and techniques to provide optimization solutions for the 2-dimesnsional heat state transfer problem where by optimizing a code to improve its efficiency require experiment and measuring the performance of the code must be done on a dedicated system. Execution time must be measured on a dedicated machine because it is used often so that performance can be independent of the (memory) "size" of the application. No other user can start a process and the user measuring the performance only runs the minimum amount of processes and should present measurement results as averages over a few experiments.

5. METHODOLOGY

The methodology adopted for this paper is using a dedicated grid server that has 10 computers, 01-08 of the computers has 2 cores and 2 GB memory and 09 and 10 computers has 8 cores and 8 GB memory with C++ Compiler along with source code (jacobi2d.c) that solve a rectangular 2 dimensional heat conductivity problem to optimized the code performance in number of steps.

6. SOLUSIONS

6.1 Experimental Setting

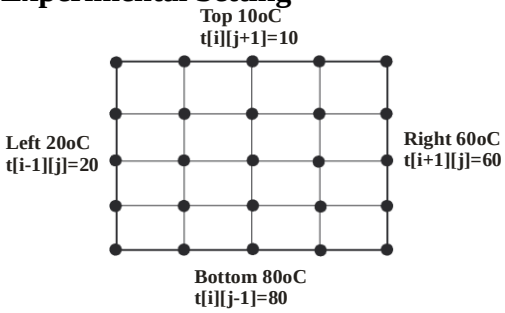

Fig 4: Rectangular 2D with top 10oC, bottom 80oC, left 20oC and right 60oC.

Figure 4 above show rectangular 2D problem with boundary conditions set at top 10oC, bottom 80oC, left 20oC and right 60oC.

Section 1.0 (Step 1)

In this step, jacobi2d.c source code (see appendix I) that solves 2 dimensional heat conductivity problem is modified to reflect the boundary conditions as shown in figure 4 above. Start timer and stop timer (tstart and tstop) was included in order to get serial (single processor) execution time.

1.1 Execution Time of grid-01 machine

The execution time of a given task is the time spent by the system executing that task, including the time spent executing system services on its behalf.

Fig. 5: runtime of a single processor

Figure 5 above show the execution time of the single processor as **183 usec.**

1.2 Recoding Runtime under a Range of Problem Size Using Different Levels of Compiler Optimization

To execute and record the runtime under a range of problem sizes using different level of optimization level in order to determine what time each level would produce, optimization level 1 to 3 (–O1 to –O3) and up to 4 No. of thread were used. Also, grid-09, Maximum of 100,000 problem size and 0.001

maximum difference tolerance (see appendix I) were used as indicated in the table below.

Table 1.0 Problem Size 100000

No of Thread	Optimization level	Time (USEC)
1	-O1	34
2	-O1	43
4	-O1	69

Table 1.0 above show the runtime of each thread when optimization level 1 (-O1) was applied. The table indicate that the higher the number of threads, the higher the amount of time require to run.

Table 1.1 Problem Size 100000

No of Thread	Optimization level	Time (USEC)
1	-O2	30
2	-O2	39
4	-O2	54

Table 1.1 above show the runtime of each thread as optimization level 2 (-O2) was applied. The table indicate that as optimization level is change using same problem size there is probability of reduction in running time.

Table 1.2: Problem Size 100000

No of Thread	Optimization level	Time (USEC)
1	-O3	21
2	-O3	33
4	-O3	28

Table 1.2 above show the runtime of each thread as optimization level 3 (-O3) was used while compiling. The table indicates that optimization level 3 runtime is less than the runtime yield by level 1 (-O1) and level 2 (-O2) respectively.

Section 2.0 (Step 2)

2.1 Modifying the Serial Version of Step 1 to Produce Parallel Version Using OpenMP.

In this step, the serial version (see Appendix I) of step 1 was modified, (see Appendix II) compile and tested on a platform that has OpenMP installed to establish correct operation using 1, 2, 3 and 4 threads/processors regardless of performance because it is not an issue.

2.2 Compiling OpenMP (Parallel Version)

OpenMP require to use the appropriate compiler flag to "turn on" OpenMP compilations. The following command was used to compile the parallel version. Gcc –fopenmp omp.c –o silver on grid-09 machine using –O3 (performance level 3) and up to 1, 2, 3 and 4 number of threads was used. For the number of thread define, the starttime and stoptime, problem size and maximum difference tolerance (see Appendix II). Table 2.0 below shows the runtime.

Table 2.0: Runtime of threads

NO OF THREAD	RUNTIME (USEC)
1	147.62
2	106.33
4	87.09

Table 2.0 above shows the runtime of each number of thread as compiled with performance level 3 (-O3). Performance in this step is not an issue.

Section 3.0 (Step 3)

3.1 Performance Test of Parallel version Implemented in step 2.

This step uses grid machines to run performance tests with the OpenMP created in step 2, it is require to remove most of the print output from the code and increase the problem size to provide sufficient work to demonstrate useful speedup. Therefore problem size was increase to 400000 maximum. To provide speedup results for a range of problem sizes, up to 8 number of threads were used. In calculating the speedup, the runtime of the single processor version produced in step 1 were used and optimizations level (-O3) was applied.

3.2 Speedup

The parallel speedup of a code tells how much performance gain is achieved by running a program in parallel on multiple processors. Speedup means the length of time it takes a program to run on a single processor (t1), divided by the time it takes to run on a multiple processors (tP) or serial time (Ts) divided by parallel time (Tp) Speedup generally ranges between 0 and P, where P is the number of processors [20]. Therefore,

Speedup SP = t1 / tP. In calculating speedup, SP = t1 / tP formula was applied where t1 is single processor and tP is multiple processor. The following table shows the performance test/speedup as compiled using the following command.

gcc -fopenmp -O3 omp.c -o silver.c

Table 3.0 Problem Size = 400000,

No of Thread	Serial time = 183 usec Time (USEC)	SP = t1/tp Speedup
1	106.73	-
2	127.04	1.440
4	85.78	2.133
8	43.36	4.220

Convergence Tolerance = 0.001

Compile: -fopenmp –O3

Grid Machine = 09

Table 3.0 above presents the runtime and speedup of each number of thread using 400,000 maximum problem size and optimization level 3 (-O3). As table 3.0 above indicated, 1 number of thread speedup is not calculated because is not parallel, 2 number of thread speedup is calculated because it is parallel and the speedup = 1.440, 4 number of threads is calculated because it is parallel and the speedup = 2.133 also 8 number of threads is calculated because it is parallel and its speedup = 4.220. It is interpreted as the table indicated that code performance gain is achieved as the problem size increases from 100,000 of step 1 to 400,000 and the code was optimized.

Section 4.0 (Step 4)

4.1 Improving Parallel Performance of Step 3.

In this step, the modified parallel code (OpenMP version) of step 3 was further modified to improve the parallel performance of step 3 using grid-09 machine. This step provides results that permit comparison with results obtained in step 3. For the critical command used (see Appendix II). To improve the parallel performance the code (file) were modified by including **Priv_difmax** and rename to step4.c, optimization level -O3 was applied and result of this step is compared with step 3 (see table 4.0 and fig. 6) below. For the **Priv_difmax** (see Appendix II). The following table shows the improved parallel performance of step 3 as compiled using the following command.

gcc -fopenmp -O3 omp.c -o abaho4.c

Table 4.0 Problem Size = 400000

NO OF THREAD	Serial time = 183 usec TIME (USEC)	SP = t1/tp SPEEDUP
1	147.8	-
2	82.27	2.224
4	41.82	4.375
8	24.11	7.590

Convergence Tolerance = 0.001

Compile: -fopenmp –O3

Grid Machine = 09

Table 4.0 above presents the improved runtime and speedup of parallel performance of each number of thread using 400,000 maximum problem size and optimization level 3 (-O3). As table 4.0 above indicated, 1 number of thread speedup is not calculated because is not parallel, 2 number of thread speedup is calculated because it is parallel, the speedup = 2.224, 4 number of threads is calculated because it is parallel, the speedup = 4.375 also 8 number of threads is calculated because it is parallel and its speedup = 7.590. It is interpreted as the table 4.0 above shows that code improved parallel performance gain is achieved as the problem size increases from 100, 000 to 400,000 and the code was further modified and optimized.

7. RESULT (OPTIMAL PERFORMANCE)

The following table and the corresponding graph shows the comparisons of the speed up of step 3 and step 4.

Table 4.1 Step 3 and Step 4 Speedup Comparison

No of Threads	Speedup Step 3	Speedup Step 4
1	-	-
2	1.440	2.224
4	2.133	4.375
8	4.220	7.590

Table 4.1 above presents the comparison of speedup of step 3 and 4. Step3 speedup is the modified version of step 2 while speedup of step 4 is the improved parallel performance of step 3. As table 4.1 above indicated there is remarkable increase in performance in step 4 because the code was further modified.

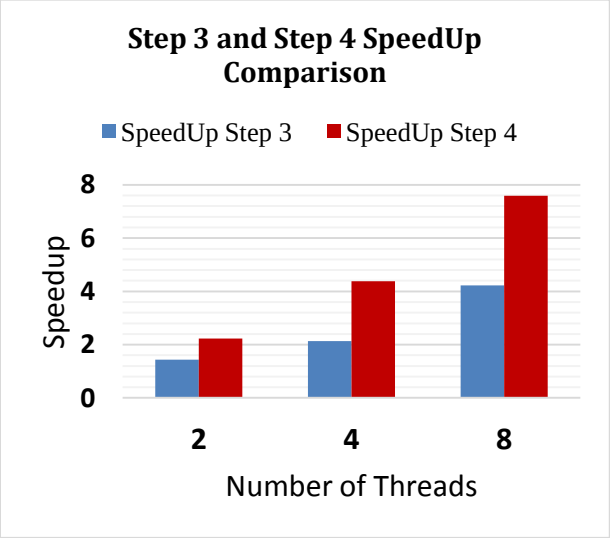

Fig 6: Step 3 and Step 4 Speedup Comparison

Figure 6 above is the pictorial representation of the comparison between the speedup of step 3 and step 4 as indicated in table 4.1 above.

8. DISCUSSION

It is observed that step 3 speedup (performance gain) is promising because the problem size was already increased but can we get any more improvement from the parallelism itself? Yes when the parallel code is improved more further the parallel performance would improve as in step 4. It is suggested Optimization level 3 (-O3) should be applied and maintain in determining speedup because higher levels of optimization should be used with caution because the higher the optimization level the higher the probability that a debugger may have trouble dealing with the code depending on the compiler. It is also observed that increase in number of thread does not minimizes the execution time in step 2 after the serial source code version is modified to parallel OpenMP version.

9. CONCLUSION

Essential benefit of parallel processing cannot be over emphasized. The generalization of optimization theory and techniques to other formulations comprises a large area of applied mathematics. An optimization problem consists of

maximizing or minimizing a real function by systematically choosing input values from within an allowed set and computing the value of the function. Most of the related literature that contribute to knowledge uses mathematical approach to provide optimization solutions for the 2-dimesnsional heat conduction problem. This paper uses computer science approach to provide solutions by optimizing the source code that solve the 2-dimensional heat conduction problem because optimizing a code to improve its efficiency require experiment and measuring the performance of the code must be done on a dedicated system. Execution time must be measured on a dedicated machine because it is used often so that performance can be independent of the (memory) "size" of the application. In this paper, the process started with serial processing in step 1 to get the execution time of the single processor, in step 2, application produce in step 1 was modified to produce the parallel version using OpenMP regardless of performance because it is not an issue, in step 3, OpenMP version of step 2 was modified to test the performance of the parallel version and to determine the speedup, in step 4, the OpenMP version of step 3 code was further modified to improve the parallel performance which provides results that permit comparison with result obtained in step 3. Optimal (Result) performance was achieved in step 4. This concludes that parallel processing method is best and fastest than serial, and when the code is optimized, it will yield best performance also parallel processing is more suitable in solving complex problem as when the problem size increase.

10. FUTURE WORK

Speedup Extremes is suggested as future work. Extremes of speedup happen when speedup (SP) is greater than P or SP less than 1, this is called super-linear speedup. How can speedup be greater than the number of processors used? The answer usually lies with the program's memory use, with multiple processors each processor only gets part of the problem. How does this help? It is possible that the smaller problem can make better use of the memory hierarchy, the caches and the registers. The smaller partitioned problem may fit in cache when the entire problem would not. When super-linear speedup is achieved, it is often an indication that the sequential code, run on one processor, had serious cache miss problems. The most common programs that achieve super-linear speedup are those that solve dense linear algebra problems (often using very large matrices and data sets).

Parallel Code is slower than Sequential Code: When speedup is less than one, it means that the parallel code runs slower than the sequential code sometimes called slowdown. This happens when there isn't enough computation to be done by each processor. The overhead of creating and controlling the parallel tasks outweighs the benefits of parallel computation causing the code to run slower. To eliminate this problem you can try to increase the problem size or run with fewer processors.

11. REFERENCES

[1] B. Blaise and L. Lawrence, "Introduction to Parallel Computing National Laboratory". Available from: https://computing.llnl.gov/tutorials/parallel_comp/ [Last Accessed 14 September, 2019].

[2] L. J. Villegas, E. Castro & J. Gutiérrez 2009 "Representations in problem solving: A case study with optimization problems", Electronic Journal of Research in Educational Psychology, April 2009.

[3] IGI Disseminator of Knowledge [available from: https://www.igi-global.com/dictionary/cuckoo-search-for-optimization-and-computational-intelligence/21383 [Last Accessed 3 January, 2020]

[4] S. Almeida, 2013 "An Introduction to High Performance Computing", International Journal of Modern Physics, 2013.

[5] P. Krastev, "Introduction to Parallel Computing", FAS Research Computing, Faculty of Arts and Sciences, Harvard Univ.

S. Wang and R. Ni, 2019 "Solving of Two-Dimensional Unsteady-State Heat-Transfer Inverse Problem using Finite Difference Method and Model Prediction Control Method", Volume 2019, Article ID 7432138, 12 pages, Hindawi Publishing, 2019.

[6] F. Mohebbi and M. Sellier, "Parameter Estimation in Heat Conduction using a Two Dimensional Inverse Analysis", International Journal for Computational Methods in Engineering Science and Mechanics, 2016; http://dx.doi.org/10.1080/15502287.2016.1204034

[7] T. J. Ikonen, G. Marck, A. Sobester and A. J. Keane, 2018 "Topology Optimization of Conductive Heat Transfer Problems using Parametric L-Systems", Structural and Multidisciplinary Optimization, Volume 58, Issue 5, pp. 1899-1916, 2018.

[8] X. Peng, K. Niakhai and B. Protas, "A Method for Geometry Optimization in a Simple Model of Two-Dimensional Heat Transfer", 2013, available at: https://arxiv.org/pdf/1307.1248.pdf [Last Accessed 15 January, 2020]

[9] G. G. Tejani, V. J. Savsani, V. K. Patel and S. Mirjalili, 2019"An Improved Heat Transfer Search Algorithm for Unconstrained Optimization Problems", Journal of Computational Design and Engineering, Volume 6, Issue 1, pp. 13-32, 2019.

[10] J. F. Opadiji, T. O. Fajemilehin and S. A. Olatunji, 2019 "Performance Evaluation of Equally Spaced Linear Array Antenna", © 2019 Afr. J. Comp. & ICT – All Rights Reserved https://afrjcict.net Vol. 12, No. 2, June 2019, pp. 19 – 29 2019.

[11] ScienceDirect: "Heat conduction" From: Finite Element Analysis Applications, 2018. Available at:http?//African%20Journal/Heat%20Conduction%20-%20an%20overview%20_%20ScienceDirect%20Topics.html [Last Accessed 5 January, 2020]

[12] C. Massimo and A. Giovani, "Grid, Cloud and Virtualization, Computer Communication and Network", available at: http://www.asecib.ase.ro/cc/carti/Grids,%20Clouds%20and%20Virtualization%20%5B2010%5D.pdf [Last Accessed 1 January, 2020]

[13] S. Karishma and M. Michele, "Cloud Computing vs. Grid Computing", 2011. Available at: http://www.brighthub.com/environment/green-computing/articles/68785.aspx [Last Accessed 1 January, 2020]

[14] B. Hadi "Compilers What Every Programmer Should Know about Compiler Optimizations", .2015 available from:https://msdn.microsoft.com/en-

gb/magazine/dn904673.aspx [Last Accessed: 14 December, 2016]

[15] Clouds, Grids and Virtualisation Introduction to OpenMP © 2012 Greenwich Univ., available from (http://openmp.org) [Last Accessed 2 January, 2020].

[16] Clouds, Grids and Virtualisation Introduction to OpenMP © 2012 Greenwich Univ., available from (http://openmp.org) [Last Accessed 2 January, 2020].

[17] Techopedi Runtime Library, available from https://www.techopedia.com/definition/17261/runtime-library [Last Accessed 2 January, 2020]

[18] HEATED_PLATE_OPENMP "*2D Steady State Heat Equation Using OpenMP",* available from: https://people.sc.fsu.edu/~jburkardt/c_src/heated_plate_openmp/heated_plate_openmp.html, [Last Accessed 13 January, 2020]

[19] RWTH, "Compute Cluster Parallel Programming OpenMP The Jacobi Solver Revisited", available from https://doc.itc.rwth-aachen.de/display/CCP/The+Jacobi+Solver+revisited [Last Accessed 15 January, 2020]

[20] Clouds, Grids and Virtualisation Performance Measures University of Greenwich.

[21] L. G. Blanch "Two-Dimensional Modelling of Steady State Heat Transfer in Solids with use of Spreadsheet (MS EXCEL)", Thermo-energetical Master Thesis, Dept. Mech. Eng. and Com. Sci., Bielsko Univ., Bielsko-Biała, Poland, Spring 2011

12. APPENDIX I
Serial Version of Jacobi2d Source Code

```
#include <omp.h>
#include <stdio.h>
#include <math.h>

int max_size=100000;
int main(int argc, char *argv[])
{

//omp_set_num_threads(1);       // This number 1 is going to determine the number of threads, 1, 2, 3, etc
//# pragma omp parallel
{

int m = 20;
int n = 5;
double tol = 0.001;
double tstart, tstop;
double t[m+2][n+2], tnew[m+1][n+1], diff, difmax;
int i, j, iter;

// initialise temperature array
for (i=0; i <= m+1; i++) {
    for (j=0; j <= n+1; j++) {
        t[i][j] = 20.0;
    }
}

 // fix boundary conditions
 for (i=1; i <= m; i++) {
     t[i][0] = 10.0;
     t[i][n+1] = 80.0;
 }
 for (j=1; j <= n; j++) {
     t[0][j] = 30.0;
     t[m+1][j] = 60.0;
 }

 // main loop
 iter = 0;
 difmax = 1000000.0;
 while (difmax > tol) {
     iter++;

     // update temperature for next iteration
     for (i=1; i <= m; i++) {
         for (j=1; j <= n; j++) {
             tnew[i][j] = (t[i-1][j]+t[i+1][j]+t[i][j-1]+t[i][j+1])/4.0;
         }
     }

     // work out maximum difference between old and new temperatures
     difmax = 0.0;
     for (i=1; i <= m; i++) {
         for (j=1; j <= n; j++) {
             diff = fabs(tnew[i][j]-t[i][j]);
             if (diff > difmax) {
                 difmax = diff;
             }
```

```
            // copy new to old temperatures
            t[i][j] = tnew[i][j];
        }
    }

}

// print results
printf("iter = %d  difmax = %9.11lf", iter, difmax);
for (i=0; i <= m+1; i++) {
    printf("\n");
    for (j=0; j <= n+1; j++) {
        printf("%3.5lf ", t[i][j]);
    }
}
printf("\n");
printf("time taken is %4.3lf\n", (tstop-tstart));
}
}
```

13. APPENDIX II
Parallel Version (OpenMP) of Jacobi2d Source Code

```
#include <omp.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

int max_size=400000;

int main(int argc, char *argv[]) {
    double t[max_size], tnew[max_size], tol, diff, difmax, priv_difmax;
    double tstart, tstop;
    int i, j, iter, n, m, nthreads;
    printf("enter the problem size (n) and the convergence tolerance\n");
    scanf("%d %lf", &n, &tol);
    printf("Enter the number of threads (max 10) ");
    scanf("%d",&nthreads);

    /* define the number of threads to be used */
    omp_set_num_threads(nthreads);
tstart = omp_get_wtime ();
    // initialise temperature array
    #pragma omp parallel for schedule(static) \
        default(shared) private(i)
    for (i=1; i <= n; i++) {
        t[i] = 20.0;
    }

// fix boundary conditions
for (i=1; i <= m; i++) {
    t[0] = 10.0;
    t[n+1] = 80.0;
}
for (j=1; j <= n; j++) {
    t[0] = 30.0;
    t[m+1] = 60.0;
}
```

```
  // main loop
    iter = 0;
    difmax = 1000000.0;
    while (difmax > tol) {
        iter=iter+1;
        // update temperature for next iteration
        #pragma omp parallel for schedule(static) \
            default(shared) private(i)
        for (i=1; i <= n; i++) {
            tnew[i] = (t[i-1]+t[i+1])/2.0;
        }
// work out maximum difference between old and new temperatures
        difmax = 0.0;
       #pragma omp parallel default(shared) private(i, diff, priv_difmax)
        {
            priv_difmax = 0.0;
            #pragma omp for schedule(static)
            for (i=1; i <= n; i++) {
                diff = fabs(tnew[i]-t[i]);
                if (diff > priv_difmax) {
                    priv_difmax = diff;
                }
                t[i] = tnew[i];
            }
            #pragma omp critical
            if (priv_difmax > difmax) {
                difmax = priv_difmax;
            }
        }
      }//while (difmax>tol)
   tstop = omp_get_wtime ();
```

```
    for (i=1; i <= n; i++) {
        printf("t[%d] = %-5.7lf  \n", i, tnew[i]);
    }
    printf("iterations = %d  maximum difference = %-5.7lf  \n", iter, difmax);


    printf("time taken is %4.3lf\n", (tstop-tstart));
}
```