# Applications for Multi-core System

Mamta Kumari Prasad

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Rourkela, Odisha 769008

# Applications for Multi-core System

Thesis submitted

By

**Mamta Kumari Prasad**

(109cs0589)

In partial fulfilment for the award of the degree of

**Bachelor of Technology**

In

**Computer Science and Engineering**

Under the Guidance of

**Prof. Ashok Kumar Turuk**

Department of Computer Science & Engineering

National Institute Of Technology Rourkela-769008, Orissa, India

National Institute Of Technology

Rourkela-769008, Orissa, India

# Certificate

This is to certify that the thesis entitled “**Applications for Multi-core System**” is submitted by Mamta Kumari Prasad, (Roll NO 109cs0589) to this Institute in partial fulfilment of the requirement for the award of the degree of Bachelor of Technology in Department of Computer Science & Engineering, is a bonafide record of the work carried out under my supervision and guidance. It is further certified that no part of this thesis is submitted for the award of any degree.

Date:

Prof. Ashok Kumar Turuk

Associate Professor

Head of the Department

Computer Science & Engineering

NIT Rourkela

# Acknowledgement

It is my great pleasure to express my deep sense of gratitude and sincere thanks to my Research supervisor, Prof Ashok Kumar Turuk, Associate Professor, Department of Computer Science & Engineering, NIT Rourkela, for his excellent guidance, constant encouragement, critical reviews and valuable suggestion throughout this research work. It would not have been possible for me to bring out this project report without his help and constant encouragement. His profound experience, incomparable expertise along with kind supportive nature has been a substantial asset for me throughout my learning experience.

My heartiest thanks to Mr Ram Prasad Mohanty for consistently showing me innovative research directions for the entire period of carrying out the research. I am indebted to all the professors, batch mates and friends at National Institute of Technology Rourkela for their cooperation.

And at last but not the least, I am thankful to the almighty and my family members who have constantly encouraged and supported me. Without their support this project could not have been a success.

Mamta Kumari Prasad

# Abstract

A multi-core processor is a single computing unit with two or more processors (“cores”). These cores are integrated into a single IC for enhanced performance, reduced power consumption and more efficient simultaneous processing of multiple tasks. Homogeneous multi-core systems include only identical cores, whereas heterogeneous multi-core systems have cores that are not identical.

Most of the computers and workstations these days have multicore processors. However most software programs are not designed to make use of multi-core processors and hence even though we run these programs on the new machines equipped with multicore processors, we don’t see sizable improvements in application performance. The idea behind improved performance is in parallelizing the code and distributing the work amongst multiple cores, but writing programming logic to achieve this is complex. The conventional model of lock-based parallelism for writing such programs is difficult in use, error-prone and does not always lead to efficient use of the resources but with the help of OpenMP, programmers have enhanced support for parallel programming. In this work I have implemented quicksort algorithm using OpenMP library and analysed the performance in terms of execution time.

# Contents

# Chapter 1

# Introduction

## 1.1 Background

In recent era, computer architects no longer focus on increasing the single-core processor clock-speed or small architectural improvements to enhance processor performance and found it challenging in getting more instruction-level parallelism from a single program. This lead to the consideration of thread-level parallelism. This strategy is well-known and can improve processor performance. This resulted in multithreaded processors. But most of the applications are not made for multithreaded systems. Therefore, adding cores to the processors resulted in very little performance enhancement. During course of time researchers proposed many programming languages to get benefitted from multiple cores. All these languages support high-level parallelism thus making parallel programming much easier than primitive methods.

## 1.2 OpenMP

The OpenMP (open multiprocessing) is an API(Application Programming Interface) which provides a portable and scalable model for shared memory parallel applications. OpenMP was introduced in 1997 for standardized programming in shared memory systems. It is used widely since then for parallelizing scientific applications. It consists of a set of compiler directives, library routines and environment variables which directs run time behaviour. These facilities allow users to specify the regions in the code that are parallel. Users also specify necessary synchronization like variable locks, etc. for correct parallel region execution. It supports fork-join model. During runtime, threads are forked into different parallel regions and are executed in different processors having same memory and address space. Another benefit of having multi-core is that we can use cores to extract thread level parallelism in a program and enhance the performance of a single program.

Multi-core programming in C/C++ on such architectures supports UNIX and Windows operating systems. Due to the exceptional hardware speeds and drop of costs, many developers does not give importance to code optimization. Consequently, earlier developed techniques have not been upgraded for modern compiler optimization techniques and hardware features. Programmers often choose language they are comfortable with, even if it’s not the most effective language for their work. Speeds, flexibility, ease of coding are few important points to consider while choosing which language to use. Compiler performs several optimizations faster than human programmer does. Optimizations like moving constant expressions out of loops, storing variables in registers, etc. should be performed by compiler in most cases. Parallelization of serial programs, parallelizing compiler depend on the analysis of subscript to detect data dependencies between array pair references inside loop nests.

OpenMP have been used till today only in traditional multiprocessor environments and we know that multi-core processors are very similar to a traditional multiprocessor. So it is obvious that OpenMP can be used in multi-core processors. In our work we achieve thread level parallelism and it reduces communication cost.

# Chapter 2

# Literature Survey

Earlier work has studied the implementations of quicksort algorithm in many programming languages such as C, C++ and Java. But there is necessity of understanding parallel programming and its implementation methods on multi processors systems to enhance the performance. OpenMP is a widely known parallel programming technique for multiprocessing environment. Various hardware and software techniques have been adopted for enhancement of performance. One of the traditional methods to enhance performance is increasing the clock frequency. There are various kinds of heterogeneous pipeline models that have been discussed by researchers. Latch based pipelines are most commonly used pipelines in asynchronous circuit pipeline models [5]. Fine grained and coarse grained pipeline structure which focuses on cell gate implementation were introduced and improved by many computer architecture researchers [4] [3].

[6] Tried to use speculative techniques to improve OpenMP programs. Here the threads do not always (depending on the OpenMP hint) wait at synchronization points. So violations could occur which are detected and the offending thread is squashed. In our work we use OpenMP to improve thread level parallelism of integer programs which are not inherently parallel.

## 2.1 Parallel Programming Models

There are several parallel programming models in use today. To list a few here are they:

- Shared memory (without threads)
- Threads
- Distributed Memory / Message Passing
- Data Parallel
- Hybrid

- Single Program Multiple Data (SPMD)
- Multiple Program Multiple Data (MPMD)

Parallel programming models exist as an abstraction above hardware and memory architectures. These models are not specified for a particular type of machine or memory architecture. Any of them can be implemented on any hardware system. A shared memory model can be implemented on a Distributed memory machine. For example Kendall Square Research (KSR) ALLCACHE approaches. Similarly vice-versa is also possible like in Message Passing Interface (MPI) on SGI Origin 2000. There always a question come which model to use? It is often combination of what is affordable and personal favour. There is no “best” model, although there are certainly better implementations of some models over others.

## Shared Memory Model (Without threads)

Tasks share a common address space in this model, which they read and write to asynchronously. Locks and semaphores are used for synchronization and controlled access to shared memory. It has some advantages from programmer’s point of view, the notion of “ownership” of data is lacking, so programmers don’t need to specify explicitly the communication of data among tasks. In terms of performance, major disadvantage is that it becomes very difficult to understand and manage data locally as keeping data local to the processor saves memory accesses, cache refreshes and bus traffic. But managing data locally is beyond the understanding of average user.

## Threads Model

It is basically a type of shared memory programming. In threads model, a single program has multiple units which executes concurrently. From programming point of view, threads implementations generally comprise of a library of subroutines which are called from inside parallel source code and a set of compiler directives either in serial or parallel source code.

Threaded implementation is not new in computing world. Earlier, hardware vendors developed their own versions of thread. But later various standardization efforts resulted in two different implementation of threads: POSIX threads and OpenMP.

### Distributed Memory / Message Passing Model

Here set of tasks use their own local memory for computation and they can reside on the same machine and/or across various number of machines. They exchange data through sending and receiving messages. Data exchange usually requires cooperation between the processes. For example, a send operation must have a receive operation matching to it. From programmer perspective, message passing implementations comprise of subroutines. Calls to these routines are there in source code. Here programmer is responsible for determining all parallelism. MPI is the industry standard for message passing, replacing all the existing message passing implementations used in industry. These specifications exist for all popular parallel computing platforms.

## 2.2 Ways to create parallel programs

In this section, we will compare OpenMP with other most important alternatives for programming in a shared-memory machine. Some vendors provide APIs on their platform. Even though these APIs may be fast, programs written using them have to be rewritten to execute correct in a different machine. Such APIs cannot be considered which cannot be used for broader perspective.

### 2.2.1 Automatic parallelization

Many compilers provide option or flag, for automatic parallelization of program. When this option is used, the compiler automatically analyses program for independent sets of instructions, especially loops with independent iterations. This information is then explicitly used for generating parallel code. One of the ways in which this can be achieved is by generating OpenMP directives, which therefore enables the programmer to view and possibly improve the resulting code. The problem with relying on compiler to detect and exploit parallelism is that it may sometimes lack the knowledge to do a good job. For example, it may require the values that will be assumed by loop bounds or the range of values of array subscripts: but these are unknown before run time. So in order to preserve correctness, compiler has to conservatively assume that a loop is not parallel whenever it cannot prove the contrary. As a result, the more complex a code is the more chances of occurrence of this. For programs with simple structure, it is worth trying.

### 2.2.2 MPI

The Message Passing Interface was developed to provide potable programming for distributed-memory architectures (MPPs), where memory processes execute independent of each other and communicate data whenever needed through exchange of messages. This API was designed to be highly expressive and to enable the development of efficient parallel code and can be used broadly. Consequently, it is most widely used API for parallel programming in the technical community where MPPs and clusters are common.

Creating an MPI program may be tricky. The programmer should create code that will be executed by each process and this required a good amount of reprogramming. It can be difficult to create a single program version that runs efficiently on many different systems, since the comparative cost of communicating data varies from one machine to another and this may suggest different approaches to extracting parallelism. Care has to take for programming errors, especially deadlock situation where two or more processes wait for each other to send a message.

### 2.2.3 Pthreads

It is a set of threading interfaces developed by IEEE committees in charge of POSIX (Portable Operating System Interface. It supports shared-memory programming through a collection of different routines for creating, managing and coordinating a collection of threads. Hence, like MPI it is also a library. Some features were primitively designed for single processor, where context switching enables a time-sliced execution of multiple threads, but it is also good for small SMPs programming. Its goal is to be expressive as well as portable and it provides considerable set of features to create, terminate, and synchronize threads and prevent other threads from trying to modifying the same variable at the same time. Or simple prevents race-around condition: it induces mutexes, locks, condition variables and semaphores. Moreover, programming in Pthreads is much more complicated than with OpenMP and the resulting code is likely to differ substantially from the prior sequential code. Even simple tasks are done via complicated steps, and thus a simple program will contain many calls to Pthread library. Compared to Pthreads, the OpenMP API directives make it easy to specify parallel loop execution, to synchronize threads and to specify whether data is shared or not. For lot of applications, it is sufficient.

# Chapter 3

# Algorithm and Implementation Methodology

## 3.1 Serial Quicksort

The implementation details for a serial quicksort as described by Anany Levitin in his book Design and Analysis of Algorithms are:

1. Choose a pivot item, generally by just picking the last item from the sorting area.
2. Iterate through the elements to be sorted, moving numbers smaller than the pivot to a position on its left, and numbers larger than the pivot to a position on its right, by swapping elements. After this the sorting area is divided into two sub list: the left one contains all numbers smaller than the pivot element and the right one contains all numbers larger than the pivot element. The pivot is now placed in its sorted position.
3. Go to step 1 for left and right sub-list (if there is more than one element left in that list)

It can be observed that this implementation strategy represents divide and conquer strategy where a particular problem is divided into sub-problems and each sub-problem is a mini version of the original problem. Each sub-problem can be solved recursively by applying the same technique.

Once partitioning of the data is done, different sections of the list can be parallel sorted. Suppose we have p processors, we can divide a list of n elements into p sub-lists in (n) average time, then sort each of these in ((n/p)log(n/p)) average time. The pseudo code for the serial implementation is as shown in figure 1.

```
quicksort( void *a, int lo, int hi
)
{
int pivot;
/* Termination condition! */
if ( hi > lo )
{
pivot = partition( a, lo, hi );
quicksort( a, lo, pivot-1 );
quicksort( a, pivot+1, hi );
}
}
```

Fig.1 Pseudo code for Serial Quicksort

## 3.2 Parallel Quicksort

Quicksort is considered as better performing sorting algorithm and on top of that it is one of reliable algorithms which can be easily adapted to parallelization. With quicksort, partitions can be parallel sorted and combined later with the help of operations like merge and gather the output sorted data.

### Basic implementation steps:

- Perform an initial partitioning of data till all the processes get a subset to sort sequentially.
- Sort the received data set by each process in parallel using OMP threads.
- Merge all the subsets to get a final sorted list.

#### Initial data partitioning and sorting

All the implementations is done by sorting integer values generated by a random function. According to the number of processes, the initial data partitioning is done so that all processes sort equal no of elements. Sorting of each data subset is done by individual processes.

### Sorting by individual process

After initial data partitioning all sub list of data are distributed to individual processor. Each processor sorts data through threads.

```
int *counter = (int *)malloc((threads+1)*sizeof(int));

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

        counter[i]=i*size/threads;

    counter[threads]=size;

    /* Main parallel sort loop */

     int j;

#pragma omp parallel for private(i,j)

    for(i=0; i<threads; i++)    {

            qsort(a+counter[i],counter[i+1]-
counter[i],sizeof(int),CmpInt);

    }
```

Fig. 2 code for sorting of individual sub-list

## Merging of sorted data

A tree based merge implementation is used where each process sends its sorted sub-list to its neighbour and a merge operation is done at each step. The figure below represents this implementation.

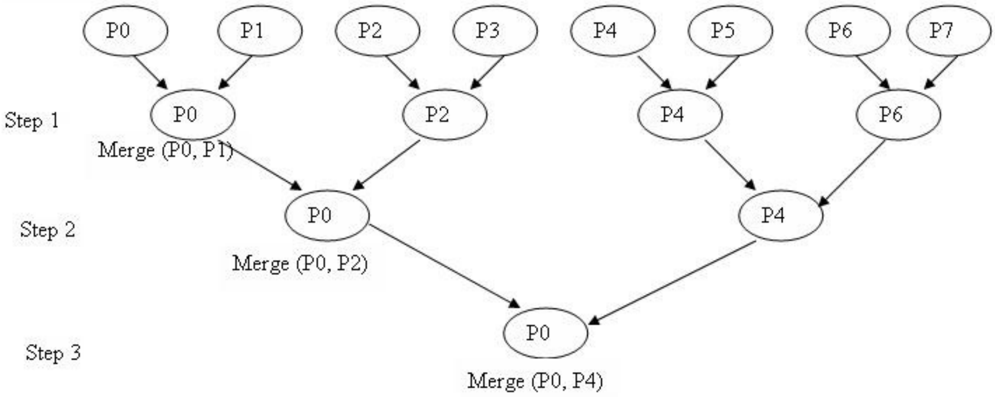
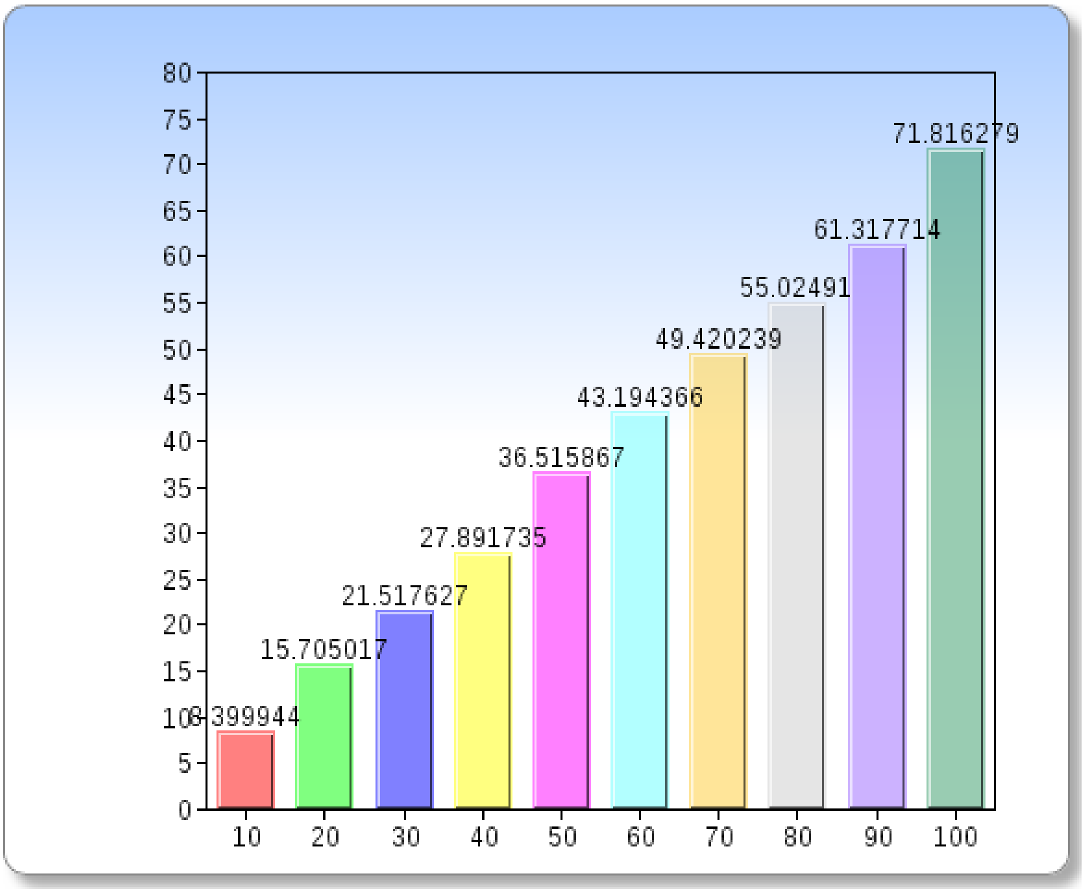


Fig. 3 Tree based implementation of merge operation

# Chapter 4

# Result

The performance of parallel quicksort is measured in terms of its execution time. The execution time for individual data set is measured. The data set size is kept as multiple of 1 million numbers. With the increasing size of data the execution time is recorded. We can see a gradual increase in the bar graph with the increase in data size. This results from the merging time of data. As data size increases, merging time also increases correspondingly and thus shows an increasing pattern in the graph.



# Chapter 5

# Conclusion

Successfully implemented the parallel quicksort program using OpenMP and performed analysis comparing with the serial program. The project gave a chance to know the techniques that can be used for shared memory architecture. The performance gain and speedup shows that such parallel methods should be applied into real world applications requiring sorting

# Bibliography

[1] Leonaxddoa Gum and Ramesh Menon. OpenMP: An Industry standard API for shared-Memory Programming.IEEE Comput. Sci.Eng.,1998.

[2] Nawal Copty. Taking Advantage of OpenMP 3.0 Tasking with Oracle Solaris Studio. An oracle White Paper, November 2010.

[3] Philippas Tsigas and Yi Zhang. A simple, Fast Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000. Chalmers University of Technology.2003

[4] Puneet C Kataria. Parallel quicksort implementation using MPI and Pthreads. 2008

[5] Shi Jung Kao. Managing C++ OpenMP code and its exception handling. In Proceedings of the OpenMP applications and tools 2003 international conference on OpenMP shared memory parallel programming, WOMPAT'03, pages 227–243, Berlin, Heidelberg, 2003. Springer-Verlag.

[6] Zenil Chavez. Applied Parallel Computing. IEEE Distributed Systems Online, March 2004.

[8].Alex Vrenios. A Tutorial on Parallel Systems Development. IEEE Distributed Systems Online, Volume 5, 2004.