Multicore Transformation of Sequential Applications for Achieving High Performance

Prabin R. Sahoo
Tata Consultancy Services Limited, Mumbai, India

ABSTRACT
Multicore processor [1] architecture brings a new dimension to the computing arena. Though the proliferation of multicore processor into the commodity market has promising effect on addressing hardware scalability to address the heat and power consumption, high performance computations, but with all these benefits to its credit, there are challenges in adapting multicore technology. In multicore processor architecture, CPU speed has been reduced to accommodate additional cores. A sequential application running in a higher CPU frequency needs to adjust with the reduced CPU frequency in multicore architecture. There are smart caching mechanisms which can speed up the data access, however in order to fully exploit the power of multicore requires a new way of program design and development. Therefore the existing applications require multicore transformation in order to be able to effectively utilize multicore computation capabilities [14]. This research work demonstrates why multicore transformation is required and how SMPF [16] can be used for transforming sequential application to achieve parallelism on a multicore architecture.

General Terms
Parallel Processing, Multicore Programming

Keywords
Parallelism, Multicore, Multithreading, Shared memory, Synchronization, UML, Template

1. INTRODUCTION
Multicore processors have shown promising results. Processor manufactures such as Intel®, AMD®, SUN® etc have started releasing multicore processors in the form of dual core, quad core, and processors with higher number of cores. The existing Symmetric multiprocessors [2], Single core processors have hit the bottleneck of scalability issues for achieving high performance. Increasing CPU speed for achieving high performance is no more a scalable solution at the cost of high heat and power consumptions, especially when the world is struggling to reduce global warming and reducing power consumptions. Multicore processors are going to play a major role in the IT computation. Though multicore processor compromises with reduced CPU cycles, but with introduction of multiple execution cores, it provides high computation ability. Currently parallel programming frameworks such as openMP [5], MPI [4], Cilk [3] etc are being revisited if these can be reused. However, the challenge is big as most existing applications are single threaded in nature, and requires code changes which consequently add huge man power and cost to redesign and implement. In addition learning parallel programming techniques, developing applications with multithreading programming models are complex to comprehend. This leads to the apprehension that many existing sequential applications may be deprived of fully utilizing the multicore computation power. In this paper I have demonstrated a case study, using SMPF how multicore programming can be carried out transforming sequential applications to run in parallel for achieving high performance in a multicore architecture.

2. LITERATURE REVIEW
Parallel processing methodologies [15] are being revisited if these can be used in multicore transformation. Some of the methodologies, frameworks such as Message passing interface MPI [4][17] is being considered for multicore transformation. However MPI is mainly used for massively parallel machines, SMP clusters, workstation clusters and heterogeneous networks. This framework is not easily adaptable for existing sequential applications as it requires extensive code changes. Perl based applications, single threaded C applications, Java applications are not fully compatible with it, though C, Java applications can be modified for compatibility. Even if changes are adapted for an existing application per se a C application, running it in a multicore server does not guarantee that performance would be improved on a multicore processor. Further the code changes involve the cost of development, testing would be high in such cases. openMP [5] is another option for C/C++ based application where openMP pragma are used for splitting a for loop among several threads, but adapting this also not an easy choice. For example if C/C++ based application uses embedded SQL codes it is not easy to use openMP. Most legacy applications use embedded SQL. Similarly perl and Java based application cannot use openMP. This necessitates the need for SMPF [16] framework as it is simple for programmers to adapt and cost effective for business enterprises for easier transformations.

3. MODEL AND ARCHITECTURE FOR MULTICORE TRANSFORMATION
Multicore processors are equipped with multiple execution units. This architecture therefore provides a platform in which multiple tasks can run in parallel. The experiments conducted for this paper is based on Linux Cent OS. This model is based on SMPF framework [16]. In figure 1.0, the application "A" is sequential which requires transformation for exploiting multicore. SMPF has been used to create multiple thread [6][7] objects which in turn invokes the application instances A1, A2, and A3 which are mapped to the execution core in multicore server using executor functionality. Each thread object invokes its executor function with the binary and supplied arguments to create process instance. It's components are described as follows.

3.1 Multicore Transformation Architecture

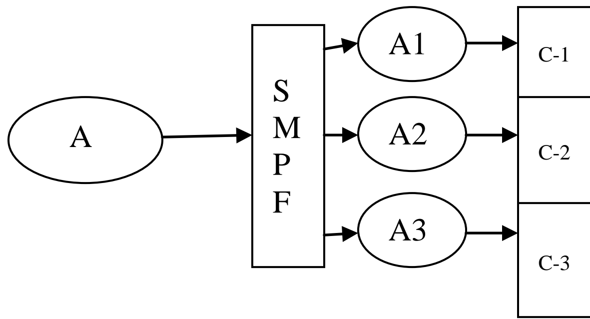

Figure 1.0 Multicore Transformation Architecture

3.1.1 Executor()

It invokes the single threaded application binary with the necessary parameter and holds a pipe [8] onto the created process instance to read any message from it. Each process instance runs in a core of the multicore processor. Any message from the process instance passes through the pipe to the executor. The executor stores the message which is finally retrieved by the main function.

3.1.2 Executor Class

This class is a template class [9]. The *Execute* method is used for executing the command. The first argument to this method is the binary name along with any arguments. The arguments types can be any data type such as string, long, float, int etc. The Executor returns a pointer to a pipe [8] which can be processed as a regular file processing. The *Execute_and_store* method executes the command and also stores the output in a vector of strings. For example: two processes are trying to search and get the order information out of which one is processing a file and other one is searching a database to get related information. In this case the Execute_and_store method is useful. Each process can store the search information in the vector container. Once the processes complete the search, the manger process processes the vector to retrieve the necessary information. The following pseudo code represents the class definitions. Line number 4 represents the vector which is used by Execute_and_store method to store the output received from the application through pipe, and the main retrieves the messages from this vector.

```
1. template <class V, class X>
2. class Executor{
3. public:
4. vector<string> vec;
5. fp  Execute(const char *cmd, V v, X x){
   }
6. void Execute_and_store(const char *cmd,
   V v, X x){}

7. };
```

The variable V and X (line 1) can be of data types such as long, int, float, char *, string etc. The parameter cmd at line 5 represents the binary need to be invoked and V, X are parameter which act as command line arguments. For arguments greater than 2 can be clubbed into V and X as strings with space in between. For example: a C/C++ binary Execute(“a.out V X)” can be invoked with more than 2 arguments as “a.out ” “123 345 567” “890 xyz” where argument V=”123 345 567” and X = “123 345 567”. A python program can be invoked as Execute(“a.py V X”). Similarly shell script can be invoked as Execute(“a.ksh V X”), and perl script can be invoked as Execute(“a.perl V X”). The basic idea here is to split the data among more than one process and execute them in parallel. Only the Python and Shell scripts cannot use shared memory in this framework for interprocess communication. If interprocess communication is required for python and shell script file can be used for this. However, Java, C/C++, PERL can use the shared memory as explained in following section 3.1.4 for inter process communication.

3.1.3 Manager Class

The manager class provides the main method which creates threads and splits data.

3.1.4 Interprocess communications

This framework provides an integrated communication mechanism. It uses pipe and shared memory [10] for communicating with each other.

3.1.5 Synchronizations

The synchronization is achieved through the combination of POSIX primitives [11] and token approach blended into object oriented paradigm for achieving powerful objected oriented interfaces for wide acceptance and reusability. The token approach uses 2 shared memory variables for synchronizations [11]. Figure 2.0 represents how the critical section for shared memory variable is synchronized. Line number 1 provides the method to invoke lock the critical section. At line 3, the invoking shared memory object gets the key which basically works as a token and line 4 checks to see if the value is 0. If the value is set to 0, it updates the shared memory and set the token for the reader to read it at line 6. The reader’s token value is 1. The reader thread polls on this key in its own address space, and when it reads the value 1, it reads the data from the shared memory at line 4 in figure 3.0.

```
1. This->lock();
2. While(true){
3. Char *str = shmkey.get();
4. If(atoi(str) == 0 ){
5. Shm.set((char *) k.c_str()));
6. Shm.set(“1”);
7. …….
8. This->unlock();
```

Figure 2.0 Pseudo code of synchronization for Writers

```
1. Int n = atoi(shmkey.get());
2. If(n == 1){
3. Int shmid=shmget(1234, 1024, 0666);
4. Token = (char *) shmat(shmid, (void
   *) 0, 0);
5. Shmkey.set(“0”);
6. }
7. Return Token;
```

Figure 3.0 Pseudo code of synchronization for the reader process

The benefit out of this approach is that processes invoked by SMPF framework do not directly act with shared memory. It is the thread object that reads the value from the process and sets the shared memory. Each process does its sequential computation without worrying about other process. The developer need not think much about shared memory, semaphores etc required for the parallel processing. Even a shell script program which has no features of shared memory, semaphore can be parallelized.

3.1.6 Address spaces

There are 2 types of address spaces discussed in this paper. Thread address space and process address space. The multicore thread objects are created in the thread address space which provides the facilities for accessing same memory locations, fast locking/unlocking mechanism provided by the thread libraries. Each process runs in its own address space i.e process address space, so an existing single threaded application has no impact on the computation or corruptions with other applications. However they can communicate with the corresponding threads for sharing data.

3.1.7 Shared Memory

It uses two shared memory classes i.e. one shared memory acts as a token and the other as the data store for processing.

4. EXPERIMENTAL SETUP

The following table describes the experimental setup for both the case studies.

Table 1.0 Hardware Configuration

Component	Details
Machine Name	4 CPU, 4 core Intel © Tigerton @ 2.93 GHz and 32 GB RAM.
Number of Cores	16 cores
Operating System	Redhat © Linux, 2.6.18-92.1.18.el5
Compiler	g++, gcc version 4.1.2 20071124 (Red Hat 4.1.2-42). Thread model: POSIX
Cache Size of each core	4096 KB

5. CASE STUDY

This case study involves a log file which captures various information an application writes when transaction occurs. During business hour it captures several information, and at the end of the day, the size of the file becomes huge. In this experiment I have demonstrated a file that contains FIX [18] messages and the size of the file is 8 GB.

This sample file contains multiple lines in the following format:

8=FIX.4.0; 11=<Order ID>; 35=<Order Qty>; 100=<Price>

Each line contains 4 name/ value pairs separated by a semicolon. Typically a search is performed on the 2nd field i.e. Order ID field for this file. A sequential operation would execute the search in the following manner:

5.1 Sequential Approach

I) Read the Filename and the search string
II) Check the File
III) If (File exists) then read line else go to VI.
IV) Match the line with the search pattern, If (match found) print at STDOUT else go to V.
V) Continue Step IV, till end of file
VI) Quit.

5.2 Parallel Approach

Using SMPF, the file size is split equally based on the number of cores. This division gives the file offset for each part. The offset is distributed among the applications.

I) Read the file name and the search pattern
II) Check the existence of file
III) Determine the number of cores in the server
IV) Determine the size of the file
V) Divide the size of file into a series of offsets
VI) Create multicore thread object using SMPF
VII) Distribute the file name, begin offset and range to each multicore thread object
VIII) Each multicore thread object invokes the application with the supplied parameters using Execute_and_store method discussed in section 3.1.2, and executes it as if it is running in a single processor mode.
IX) Each spawned application searches for the pattern in the slice and print the results to the pipe
X) Each multicore thread objects reads the data from pipe, and store it in a shared variable (vector container section 3.1.2 pseudo code). SMPF provides the synchronization mechanism to prevent data corruptions.
XI) SMPF finally reads the shared variable (vector container)

SMPF is written in C++ [12] which invokes a sequential application written in Perl which does the actual searching within the file chunk. Using SMPF this application is transformed to multicore. Figure 4.0 demonstrates high level view of the parallel approach using UML [13]. Figure 5.0 compares the execution time of normal sequential search with the parallel thread based search. The parallel search operation was implemented using SMPF. The observation is that for the above described 8 GB file a sequential search operation takes an average of 14.8 seconds over the 20 times the test was executed. However for SMPF**,** the search operation completes with an average of 3.73 seconds over the 20 test executions. Figure 6.0 shows the CPU usages.

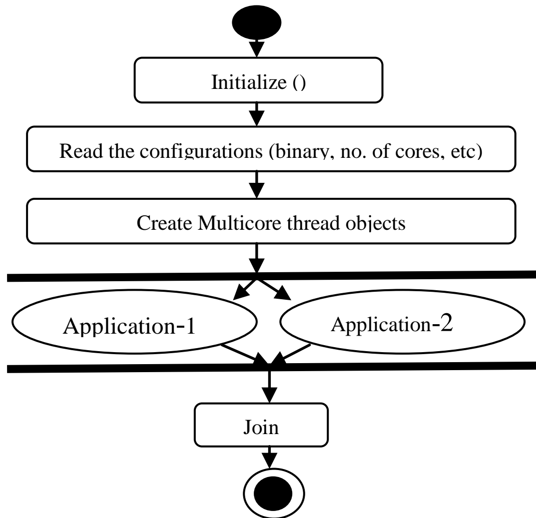

Figure 4.0 SMPF Message Flow

5.3 Results

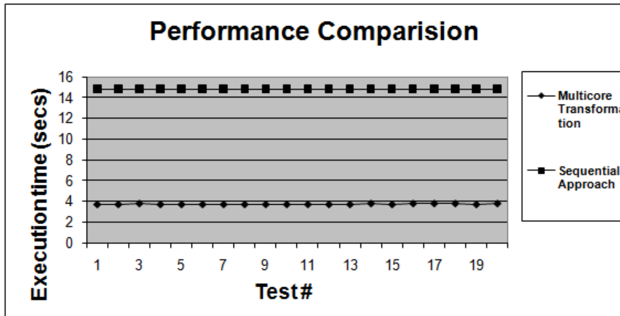

Figure 5.0 Sequential approach takes 14.8 secs and multicore transformation takes 3.7 secs for text search

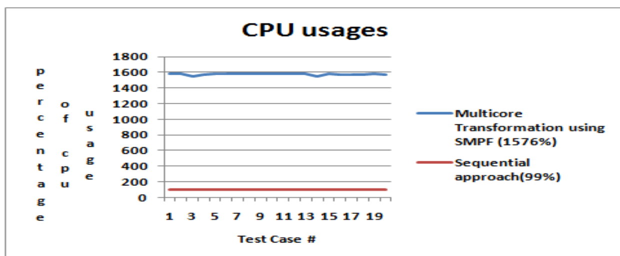

Figure 6.0 CPU usages by sequential approach and Parallel approach after multicore transformation

6. CONCLUSION

It is evident from the case study; multicore transformation produces better results than sequential approach. From CPU usage graph, it is further clear that when a sequential application is migrated to a multicore architecture, the application is confined to a core which can utilize only a core. The sequential application discussed in the case study uses a single core, where as when the application is transformed into multicore, it uses 16 cores to complete the search task in 3.7 secs with a speed up of 4X. The transform logic in the case study is simple to change which reduces the cost.

7. REFERENCES

[1] Planning Considerations for Multicore Processor Technology, John Fruehe, Dell Power Solutions, May 2005

[2] W W Gropp, E L Lusk, A taxonomy of programming models for symmetric multiprocessors and SMP clusters, IEEE Computer Society Washington, DC, USA, 1995

[3] Matteo Frigo, The Cilk Project, http://supertech.csail.mit.edu/cilk/, October, 2007

[4] R Badrinath, STSD Bangalore, Parallel Programming and MPI, hpce.iitm.ac.in/.../Parallel%20Programming%20and%20MPI.ppt, September, 2008

[5] OpenMP Architecture Review Board, The openmp api specification for parallel programming, January, 2009

[6] Felix Garcia and Javier Fernandez, POSIX thread libraries, Linux Journal, February 01, 2000

[7] Blaise Barney, POSIX Thread Programming, Lawrence Livermore National Laboratory, 2010

[8] Sandra Mamrak and Shaun Rowland, Unix Pipes, April, 2004

[9] Carlos Moreno, An Introduction to the Standard Template Library (STL), 1999

[10] Dave Marshall, IPC: Shared Memory, www.cs.cf.ac.uk/Dave/C/node27.html,1999

[11] William Stallings, Operating Systems, Internals and Design Principle, 2009

[12] Juan Soulié,C++ Language Tutorial, June, 2007

[13] Dr. Jon Siegel, OMG, Introduction to OMG's unified Modeling Language™ (UML®), http://www.omg.org/gettingstarted/what_is_uml.htm, June 2009

[14] Intel White Paper, Optimizing Software for Multi-Core Processors, 2007, USA

[15] A Generic Method of Parallel Processing in Base SAS® 8 and 9, Sassoon Kosian, Inductis, New Providence, 2007, NJ

[16] Simple Message Passing Framework for Multicore Programming Development and Transformations, Prabin R. Sahoo, ICTSM-2011, February, 2011, Mumbai

[17] Message Passing Interface, Wikimedia Foundation, Inc. http://en.wikipedia.org/wiki/Message_Passing_Interface, February 2011

[18] FIX protocol, FIX Protocol Limited http://www.fixprotocol.org/what-is-fix.shtml, 2011