Parallelism and Performance Comparison of FFT on Multi Core Machines

Faran Mahmood, Bilal A. Khan
Institute of Space Technology, Islamabad Highway, Islamabad, Pakistan,
Email: faran_isl@yahoo.com, bilal.khan@ist.edu.pk

Abstract

This work aims to propose various models for algorithm implementation on multi-core processors, and discusses pros and cons of each one. Procedural programming languages like C++ do not tend to fully utilize the multi-core processor as the program has only one thread which runs on one logical CPU. So for N cores, CPU utilization will be (100/N) % only unless the algorithm runs in parallel on different cores. This paper present three architectures for parallel processing on two cores _ spawning child processes/sharing data using file-based pipes, spawning child processes/using a hybrid of shared address space and message passing techniques, and lastly using multi-threads. All architectures are implemented using WIN32 Application Programming Interface. It is shown that for a complex algorithm, performance of our proposed hybrid model is superior to multi-thread based model and approaches Amdahl's ideality more closely.

1. Introduction

Parallel processing has been in use mostly in engineering and financial applications running on a super computer or a 'grid' since last few decades. Very recently, there is a paradigm shift in Moore's law **[14]** as there is an increasing trend to move away from single-expensive CPU's to multiple-CPU's on a chip. Arrival of these low cost multi-core processors for desktop and mobile users has brought this concept of parallel processing to a new level _ parallel processing within PC. The original purpose of these multi-core processors is to facilitate the end-user who tends to do a variety of jobs at the same time such as watching DVD while browsing. This makes the system more stable and less crash prone because even if some application hangs, windows shell **explorer.exe** can utilize the alternative core and system will be still responsive. However these multi-core processors are very unsuitable for batch processing. For instance, if you initiate a movie encoding job on a quad-core using any 3rd party software, you can observe using Windows Task manager that the best utilization of the processor is 25 % only as only 1 of the 4 cores is performing the actual task.

On multi core machines, users are encouraged **[15]** to do more than one task at the same time. Windows automatically sets their affinity to different processors for load distribution but when only one application is running, CPU cannot be fully utilized. So CPU intensive applications/simulations must be programmed using a parallel-processing framework for 100 % CPU utilization.

2. Related Work

Many authors **[1]-[2]** have implemented various algorithms for parallel processing on multi-core processors using multi-threading techniques. Due to native support for multiple-threads in Java/C#, they have been platform of choice for many developers. In **[3]**, the authors have proposed non-recursive and iterative implementation of FFT for parallel processing in super computer. In **[4],** Ian Foster suggested four stages in design process of parallel applications.

Majority of distributed applications utilize client-server paradigm where processes communicate through Remote Procedure Calls but they stress on distributed services instead of parallelism. In **[9]**, the author presented various paradigm such as pipelining and ring-based applications. Another paradigm is Single-Program Multiple-Data where different processes execute the same code but on different sets of data. High Performance Fortran **[11]** utilizes this paradigm supporting applications with regular structure. In **[12]**, authors implemented MPI (message passing interface) applications over desktop grids. Intel has introduced its thread building blocks library which supports multiple threads in an application.

Here, several models are proposed for dual-core implementation with bench-mark results provided. The algorithm of choice is a non-optimized recursive version of FFT. In fact, emphasis has been made on performance comparison using different models instead of an optimized and memory efficient implementation of FFT.

3. Using Parallel Programming Skeletons

Any sequential algorithm targeted on a multi-processor must be re-written so that parts of algorithm executes in parallel. Several parallel programming paradigms **[10]** can be used such as speculative decomposition, iterative decomposition and recursive decomposition. In recursive decomposition, the problem is split into two smaller problems which are then solved independently and in parallel. Iterative decomposition is used when various iterations of a loop are independent of each other. Speculative decomposition is present when many different solution techniques are applied in parallel with one of them completing first so that rest of them are abandoned. In this work, the paradigm used is recursive decomposition or divide and conquer approach.

In any program, if a main module utilizes independent sub-modules, then these independent sub-modules can be then executed at the same time by child processes, while the parent process (analogous to main module) is blocked till child processes (each running on separate core) finish. One underlying assumption is that both parent and child process can easily communicate and share data with each other.

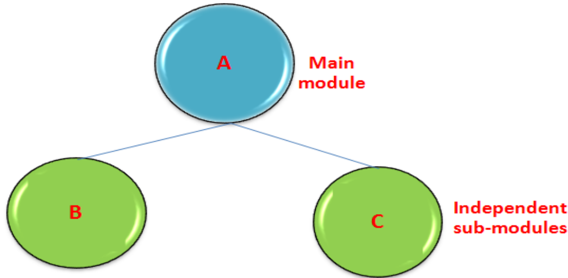

Figure 1: For job A to complete, job B and job C must be completed first, either sequentially or in parallel.

In order to use divide and conquer programming paradigm, we need to adapt the algorithm by implementing its recursive function. For example, Fibonacci series can be easily implemented in parallel as $f_n = f_{n-1} + f_{n-2}$. So, first call to recursive function executes parent process, which forks another child process with affinity set to run on different cores. Inductively, it is obvious that for **k** cores available, the algorithm must be written as **k**th order linear recursive relation i.e. $a_n = c_1a_{n-1} + c_2a_{n-2} + \ldots + c_ka_{n-k}$.

Algorithm implemented in this work is Fast Fourier Transform **[5]** (radix 2) with complexity n.log (n). Pseudo code for algorithm can be written as:

```
function y = fft(x)  // x must be array of r = 2^k elements
r = size of (x)
if (r == 2)
   y= {  (x(1)+x(0)) ,  (x(0)-x(1))  }
else
       for i=0 to r-1
          WN(i) = exp (-j*2*pi*i/r)
          // WN is array of twiddling factors
       for tt=0 to r-1
             if (tt % 2)==0
                e(tt)=x(tt)
             else
                o(tt)=x(tt)
             end
       end
       ex=fft (e)  // this & next recursive call may run
       ox=fft (o)  // in parallel
       y = concatenate (ex , ex) +
            dotproduct (WN, concatenate (ox,ox) )
end
```

If we analyze the above algorithm, it is revealed that every call to FFT function generates two recursive calls with some overhead operations. These overhead operations such as computing twiddling factors are blocking operations, and only the two recursive calls are parallelized. If we assume overhead to be approximately 0 as compared to the recursive calls, then by **Amdahl's law [13]**, α=0 and Amdahl's Ideal time is given by:

$$Amdahl's\ Ideal\ time = \frac{Time\ for\ single\ thread}{Speedup\ factor} -- (1)$$

$$= \frac{Time\ for\ single\ thread\ T_0}{\left(\alpha + \frac{1-\alpha}{no\ of\ processors}\right)} = \frac{T_0}{2}$$

4. Implementing Parent-Child Process Models

The above recursive algorithm says that to compute FFT of r elements, it computes FFT of r/2 elements twice in parallel. Using parent-child hierarchy as in figure 2, we

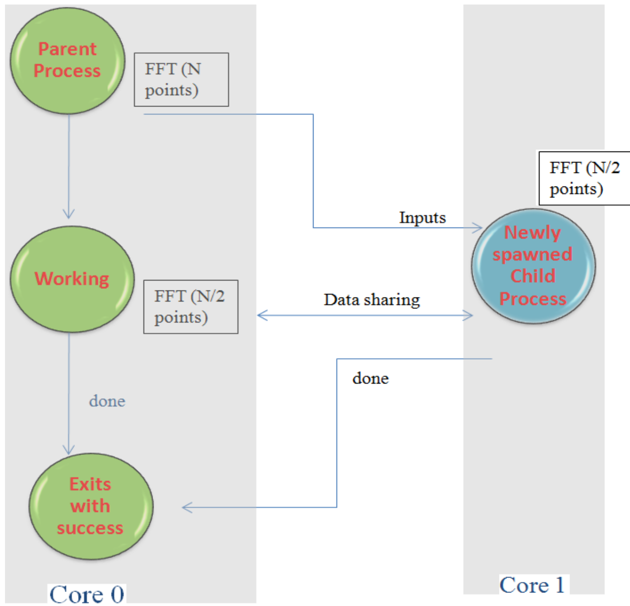

Figure 2: Parent process forks another child process which executes in parallel on alternate core

can say that parent process makes the first call to FFT function. Then when it has make two recursive calls, one recursive call is executed by parent process while the other recursive call is simultaneously executed by a newly- spawned child process. The main problem here is sharing of data and how child process signals parent process for availability of output.

Firstly, we deal with synchronization problem i.e. how to alert the parent process that child process has finished successfully. The proposed solution is to use mutex **[6]**. A mutex is a variable that can be in one of two states: unlocked or locked. The win32 API CreateMutexA is used by both parent and child processes trying to get mutual exclusion. In our case, the child process first creates mutex, and uses API WaitForSingleObject**,** which returns 0 to indicate that child process has locked the mutex. Now, if parent process creates mutex and then calls WaitForSingleObject**,** it does not return 0. Parent process, after completing its job, continuously performs this test unless it returns 0 which means that child process has also finished. The child process must call CloseHandle to release mutex. Normally, parent process does not have to wait because both processes, in theory, must finish together.

The second problem in the proposed model is sharing of data. Here is a list of various alternatives available in win32 platform for data sharing between processes.

a. Windows clipboard
b. Windows sockets
c. File-based piping
d. Shared address space (SAS)
e. Message passing interfaces (MPI) using API's
f. Hybrid of SAS and MPI

The first obvious solution is using windows clipboard. One process copies data on clipboard, and signals other process using mutex to retrieve data and clear the clipboard. Clipboard can easily be read using API GetClipboardData and written to using SetClipboardData**.** However this mode of data sharing is very crude and not recommended for most of the applications. Hence this option is not considered in this work.

Another option is using windows sockets for transfer of data over UDP employing client-server model. The client, say child process, retrieves data from server (parent process) by listening at a port 9876 and vice versa. However the major drawback of this solution is security. Besides our client process, some other process can also connect to server and retrieve data. So it requires authentication, which is again time consuming and hence, can't be afforded in given scenario.

The two processes can create a pipe using files for data transfer. Problem with file-based piping is that overheads such as hard disk latency and access times are also involved. In fact, these overheads are so significant that according to benchmarks in table 1, file-based piping takes more time than the initial single-threaded version. Hence, this is also not an acceptable idea especially when amount of data to be shared has large volume.

Due to inherent slow nature of file system as compared to main memory, this file I/O becomes the limiting factor in performance of model. This problem is solved by sharing process memory. Child process, instead of writing complete results to a file, writes only a 16-bit number which is actually a pointer to its data structure containing computed results. Then it releases mutex to signal parent process, which then accesses child process' memory using API WriteProcessMemory**[7]**. Similarly the input parameters for child process are provided by writing the pointer to input data structure so that child can access memory space of parent process. We have to code parent and child processes such that parent process has access privileges rwx while child process has privilege rx for accessing parent's address space.

This scheme drastically reduces the amount of file usage as only pointer's value (in bytes) is written to file instead of all input/output data amounting in several mega byte. However it does not completely eliminate the file based pipe but only minimizes its usage.

We can eliminate the pipe completely by passing messages using API. The pointer to memory of parent process can be passed as an argument, when creating an instance of child process. Child process can use API SendMessage along with constant WM_COPYDATA as an argument. Child process can find the handle to window of parent process by using FindWindow API. The required data to be sent is embedded in a struct

COPYDATASTRUCT. COPYDATASTRUCT contains size of data type and pointer to it. Parent process intercepts the data in its Window Procedure and transfers control when window event invoked is WM_COPYDATA. In this way, SAS model has evolved into a hybrid model by also utilizing message passing techniques.

MPI model is not a good candidate for this algorithm as a very large volume of data needs to be shared. SendMessage cannot be used for sharing large amounts of data because copying large data structures is a slow operation, whereas in SAS and its derivative hybrid model, there is no copying involved as the same data is shared by both processes.

Hence, the final hybrid model uses SendMessage for passing pointers to memory structures only, not complete data structures.

For benchmarking, we have used API GetTickCount which returns uptime of system in milliseconds. At the start of algorithm, a call is made and at the end, another call is made to calculate the computation time of algorithm. Results show that single-thread FFT consumed 50% of CPU. Model which uses file based pipe achieved 100 % CPU usage but mean time was not less than single-threaded one due to slower file I/O operations. Hybrid Model using process memory sharing with limited message passing gives drastic performance boost relative to others.

	Mean Time for 10 trials (in sec)	Standard Deviation	Mean CPU usage
Single-Thread	1.991	0.0090	50
File-based piping	2.077	0.0070	100
SAS	1.317	0.0065	100
Hybrid (SAS+MPI)	1.150	0.0087	100

Table 1: Performance comparison of multi-process models with single-threaded model for FFT of 2^{17} points

5. Implementing Multi-Threading SAS Model

Microsoft ships a dynamic link library shlwapi.dll **[8]** known as “Shell Light-weight Utility Library”. This DLL provides an API SHCreateThread for creating another thread and requires pointer to thread sub-routine. User must declare variables with global scope for sharing of data between threads and to update each other.

Threads inherently lack the stability of multi-process model because if even one thread hangs, the other thread will also be terminated by windows in an attempt to kill the process. Transfer of control in threads must be carefully planned to avoid any inter-thread interference leading the process to crash. During programming phase, a lot of crashes and hang-ups were experienced in setting up threads. Practically, debugging multi-thread based programs is not easy because the embedded debugging tools cannot be used.

The DLL is not part of operating system kernel, so it can not yield top notch performance and stability. However there is absolutely no file I/O and pipe, and this gives a major performance boost.

This performance boost is evident in bench mark results. Multi-thread based solution has a performance factor at par with process-memory sharing based solution with a only 0.5 % decreased performance. Institutively we expect multi-thread solution to excel but results show it is not the best option because the thread API is not part of operating system kernel and does not runs with real time priority.

	Mean Time for 10 trials (in sec)	Standard Deviation	Mean CPU usage
Single-Thread	1.991	0.0090	50
Multi threads	1.155	0.0031	100
Hybrid	1.150	0.0087	100

Table 2: Performance comparison of multi-thread model with other models for FFT of 2^{17} points

From table, execution time for multi-thread and hybrid model looks very comparable because the data size is very small to stress modern computers. In next section, computational load is increased in steps to magnify small differences.

6. Impact of Increased Algorithmic Complexity

Increasing number of points of input data increases the complexity O(n.log n) of FFT algorithm due to its recursive nature. It consumes more CPU time, more memory and more stack for recursion. Here, a comparison has been made between different models for increasing algorithmic complexity in terms of computational load and required memory. The results are depicted in table 3.

	Mean Time for 10 trials (sec)				
	FFT of 2^9 pts	FFT of 2^{12} pts	FFT of 2^{17} pts	FFT of 2^{18} pts	FFT Of 2^{22} pts
Single-Thread	**0.015**	0.076	1.991	4.186	78.517

Hybrid of SAS and MPI	0.155	0.172	1.150	2.285	41.076
Multi-Thread Based SAS	0.017	0.045	1.157	2.402	44.858
Amdahl's Ideal time (time for single thread/2)	0.0075	0.038	0.995	2.093	39.26
% difference between best alternative and Amdahl's time	-	18.42 %	15.57 %	9.17 %	4.6 %

Table 3: Performance comparison of two models with increased algorithmic complexity

The platform for experimentation is Microsoft VB6.0 running on Win XP SP3, 1.7GHz Core2Duo machine.

Results show that when computations are small i.e. number of points is less, there is no need to consider any of parallel processing models as the single-thread model itself can perform the computations more quickly than the time required for setting up threads or child processes. The overhead in parallel processing becomes more significant.

For algorithms with medium complexity, multiple threads offer best efficiency in terms of time required. However as we increase the complexity of algorithm by increasing FFT points, hybrid model stands out as the best. We can also observe that hybrid model approaches Amdahl's ideal line more closely when number of input points becomes very large. In case of very complex FFT computation (2^{22} points), the difference with Amdahl's ideal time is a mere 4.6 %.

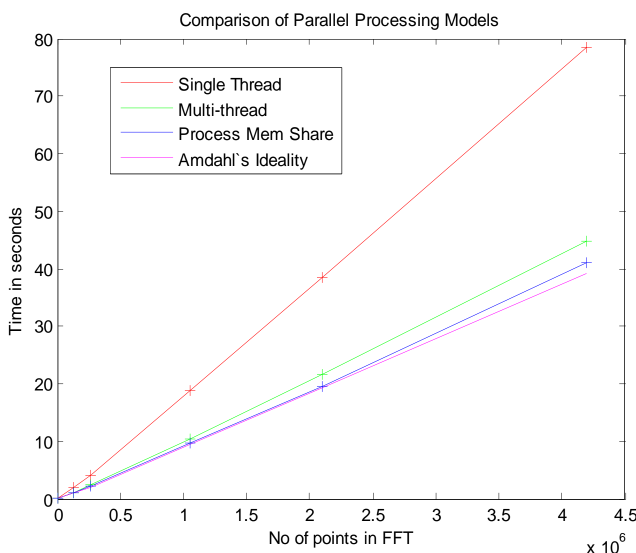

Figure 3: Comparison of model with increasing scalability in terms of data.

7. Conclusion

When an algorithm utilizes less computational and memory resources, multi-threading model is a good option due to its simplicity and absence of synchronizing / data sharing problems but with some stability issues. However for algorithms with increased complexity and scalability in terms of data, hybrid model is a good option. Implementation based on Hybrid model on Win32 platform follows Amdahl's ideality more closely than multi-thread based implementation. Hybrid model is also stable as compared to multi-threading model and can easily be extended for an implementation on a cluster.

References

[1] Stefan Naher and Daniel Schmitt. A Framework for Multi-Core Implementations of Divide and Conquer Algorithms and its Application to the Convex Hull Problem. CCCG 2008, Montr´eal, Qu´ebec, August 13–15, 2008

[2] Chang-le Lu and Yong Chen. Using Multi-Thread Technology Realize Most Short-Path Parallel Algorithm. CCCG 2008, Montr´eal, Qu´ebec, August 13–15, 2008

[3] Long Chen, Ziang Hu, Junmin Lin and Guang R. Gao. Optimizing the Fast Fourier Transform on a Multi-core Architecture. IEEE International Parallel and Distributed Processing Symposium, 2007

[4] I. Foster. Designing and Building Parallel Programs. Addison Wesley. 1996.

[5] Alan V. Oppenheim, Ronald W. Schafer and John R. Buck. Discrete-Time signal processing, 2nd edition, New Jersey: Prentice Hall, 2006.

[6] Andrew S. Tanenbaum and Albert S. Woodhull. Operating Systems Design and Implementation, 3rd edition, USA: Prentice Hall, 2006.

[7] Microsoft Developer's Network. WriteProcessMemory Function, *msdn.microsoft .com/en-us/library/ms681674(VS.85).aspx*

[8] Microsoft Developer's Network. SHCreateThread Function", *msdn.microsoft.com/en-us/library/bb759869(VS.85).aspx*

[9] P. B. Hansen. Model Programs for Computational Science: A Programming Methodology for Multicomputers. Concurrency: Practice and Experience, vol. 5 (5), pages 407-423, 1993.

[10] G. Wilson. Parallel Programming for Scientists and Engineers. MIT Press, Cambridge, MA, 1995.

[11] D. Loveman. High-Performance Fortran. IEEE Parallel and Distributed Technology, vol. 1 (1), February 1993.

[12] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali,

G. Fedak, C. Germain, T. H´erault, P. Lemarinier, O. Lodygensky, F. Magniette, V. N´eri, and A. Selikhov. Mpich-v: toward a scalable fault tolerant mpi for Volatile nodes. In Proceedings of the 2002 ACM/IEEE conference on Supercomputing, pages 1–18, Baltimore, USA, 2002.

[13] Amdahl, Gene. "Validity of the Single Processor Approach to Achieving Large-scale computing capabilities". AFIPS Conference Proceedings (30). Pg 483-485.

[14] "The Technical Impact of Moore's Law". IEEE solid-state circuits society newsletter. 2006.

[15] Intel Hyper-Threading. www.intel.com/technology/platform-technology/hyper-threading/