2010

Parallel Programming Recipes

Thuy C. Nguyenphuc
San Jose State University

Master Project:

Parallel Programming Recipes

Thuy C. Nguyenphuc

Computer Science

San Jose State University

Spring 2010

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

____

Dr. Robert Chun

____

Dr. Mark Stamp

____

Mr. Alexey Khromov

ACKNOWLEDGEMENTS

I am heartily thankful to my professor, Robert Chun, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the subject.

Special thanks to my dear friends, Alexey Khromov, Vanitha Shirwal, and Hoa Nguyenphuc, and to Mom and Dad for their encouragement and support during the completion of the project.

Also, I would like to thank and offer my regards to all of those who supported me in any respect during the completion of the project.

Table of Contents

List of Figures

List of Tables

1. Abstract

Parallel programming has become vital for the success of commercial applications since Moore's Law will now be used to double the processors (or cores) per chip every technology generation. The performance of applications depends on how software executions can be mapped on the multi-core chip, and how efficiently they run the cores. Currently, the increase of parallelism in software development is necessary, not only for taking advantage of multi-core capability, but also for adapting and surviving in the new silicon implementation. This project will provide the performance characteristics of parallelism for some common algorithms or computations using different parallel languages. Based on concrete experiments, where each algorithm is implemented on different languages and the program's performance is measured, the project provides the recipes for the problem computations. The following are the central problems and algorithms of the project: Arithmetic Algebra: Maclaurin Series Calculation for e^x, Dot-Product of Two Vectors: each vector has size *n*; Sort Algorithms: Bubble sort, Odd-Event sort; Graphics: Graphics rendering. The languages are chosen based on commonality in the current market and ease of use; i.e., OpenMP, MPI, and OpenCL. The purpose of this study is to provide reader a broad knowledge about parallel programming, the comparisons, in terms of performance and implementation cost, across languages and application types. It is hoped to be very useful for programmers/computer-architects to decide which language to use for a certain applications/problems and cost estimations for the projects. Also, it is hoped that the project can be expanded in the future so that more languages/technologies as well as applications can be analyzed

and compared. The larger comparison data is made in this manner, the better project decisions can be made by programmers when design parallel systems.

2. Introduction

Parallel programming has become a key factor in determining a system's life span and performance in the rapidly changing nature of hardware parallelism. Parallel programming is generally referred to as ways of building systems that deploy multi-core processors. The goals of parallel programming are not only to achieve the best performance from multiprocessor hardware today, but also the best performance when the number of processors increases in the near future. In order to build such software applications, programmers should understand the architectures of parallel computers in today's market as well as the future's. There are three categories of computer architectures: shared memory, distributed memory and hybrid distributed-shared memory.

2.1. Shared Memory System Characteristics:

Shared memory refers to a large block of random access memory (RAM) that can be accessed by several central processing units (CPUs) in a multiprocessor system. Processors in shared memory systems can execute tasks independently from each other, but they share memory resources. Changes in shared memory by one processor are visible to other processors. There are two main classes among shared memory systems based upon memory access time: UMA (uniform memory access) and NUMA (non-uniform memory access).

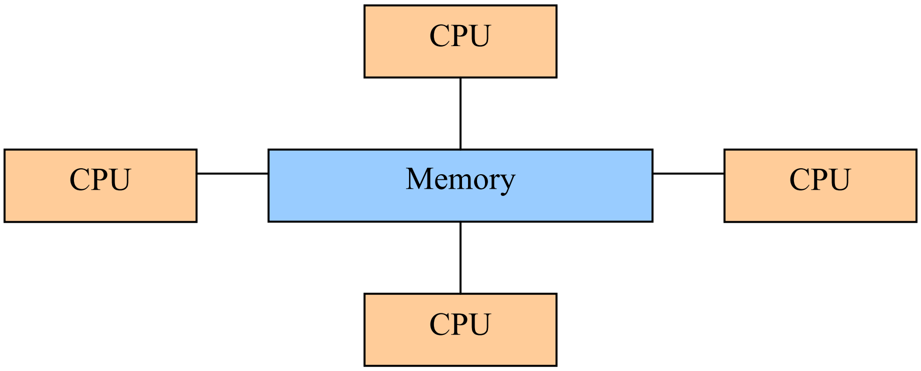


Figure 1: Shared Memory – UMA

UMA is most commonly represented by Symmetric Multiprocessor (SMP) machines, where processors in the system are identical and they have equal access rights and equal access time to memory. UMA is often called CC-UMA (cache coherent UMA). Cache coherent means if one processor modifies a piece of shared memory, all other processors know about the update. Cache coherency is accomplished at the hardware level.

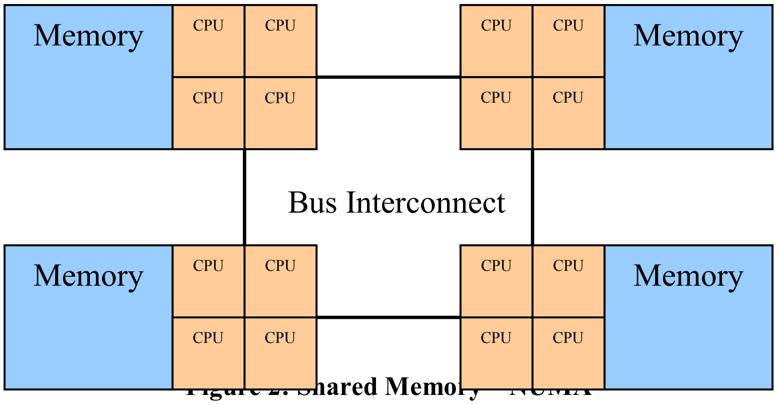


Figure 2: Shared Memory – NUMA

NUMA is often a composition of two or more SMPs. One SMP can directly access memory of another SMP and not all processors have equal access time to all memories. Memory access across the SMP link is slower. Also, if cache coherency is maintained in the system, it is called CC-NUMA (cache coherent NUMA).

Shared memory – pros and cons:

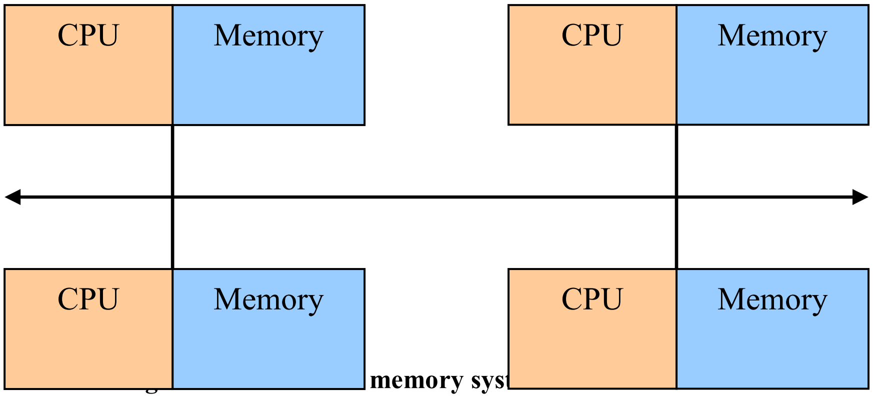
The advantages of shared memory models are that they are easy to program, and data sharing between tasks is fast and uniform. The global address space provides an easy programming perspective to memory. Shared data is fast and uniform due to the proximity of the memory to CPUs.

The main disadvantage of shared memory is the lack of scalability due to the cache coherency is hard to maintain. Adding more CPUs increases traffic geometrically between the shared memory and the CPUs. Also, the synchronization of global shared data is difficult, and it is the responsibility of programmers to implement this.

The most common languages that perform shared memory are POSIX threads or Pthreads, and OpenMP. OpenMP will be described in more depth in the next chapter (Problem Addressed).

2.2. Distributed Memory System Characteristics:

Distributed memory systems require a communication network to connect inter-processor memory. Each processor has its own private memory, and changes in its memory do not affect other processors. Hence, there is no cache coherency. The communication between processors in a distributed system is necessary to synchronize tasks among processors. The programmer is responsible for defining communication messages explicitly. The media used for the communication networks can range from local inter-connection between CPUs in a multiprocessor system to Ethernet between network systems.



memory syst

Distributed memory – pros and cons:

The advantages are scalability, accessibility, independent local memory maintenance, and system cost effectiveness. Increasing the number of processors will increase memory to the system proportionately. A distributed memory system does not have cache coherency; therefore, processors can access their local memory without interference from others. This system is cost effective because it can be built using off-the-shelf processors or computers and their networking.

The disadvantages are difficulties in: programming the communications between processors, mapping the existing global memory data structure to distributed memory organization, and handling non-uniform memory access (NUMA) time.

2.3. Hybrid Distributed-Shared Memory Characteristics

The fastest computers today employ both shared and distributed memory architectures. A shared memory component in the hybrid system is usually a cache coherent SMP computer. Processors in a SMP machine can access the machine's memory as global. The distributed memory component is the network connection between multiple SMPs. Each SMP sees only its own memory, and network communications are required to move data from one SMP to another. This architecture seems to be the most advanced hardware system and can be scaled to large numbers of computers in the network.

Hybrid distributed-shared memory - pros and cons:

The advantages are scalability, independent task assignment to SMPs in the network and quick task execution.

The disadvantages are the problems of load balancing, communications, and synchronization among SMP systems. These are not simple and are programmers' responsibilities.

2.4. Parallel Programming Tips:

Software developers need to build parallel application models and carefully map the models on the hardware systems to maximize performance and efficiency. Research at the Electrical Engineering and Computer Science Departments at UC Berkeley suggested the following factors for successful software parallelism [9].

- Design patterns should be deployed to reduce cost of parallel software implementation. It also provides a maximum achievement of performance on highly parallel computer systems.
- Targets of the patterns are to support efficient executions on thousands of cores per chip and the efficiencies are measured not only in MIPS (million instruction per second) per Watt, but also MIPS per area of silicon, and MIPS per development cost.
- Patterns of parallel programming help programmers to decide how to model a system and achieve the best performance from a set of hardware. Patterns of computations and communications are categorized into thirteen algorithmic methods: dense linear algebra, sparse linear algebra, operation on structured grids, operation on unstructured grids, spectral methods, particle methods, Monte Carlo, combinational logic, finite state machine, graph traversal, dynamic programming, back track, branch and bound, and graphical models. Depending on the nature of the application, a subset of these algorithms can be deployed.
- Binary applications from compilers tend to be out of date, because compilers do not tune for parallelism of programs. Instead, "Autotuners" should play a larger role in translating parallel programs to binaries.
- Programming models must be human-centric or carry metaphors that are familiar to programmers so that they can maximize a programmer's productivity.
- Programming models should be independent from the number of processors so that programs can run on the future processors. In other words, software will not need to be re-written if its model does not depend on the number of processors.

- Programming models should support a wide range of data types and different level of parallelisms such as task-, word-, and bit- level parallelisms. This factor will maximize reused codes and models.
- Architects should not include features that affect performance or energy if programmers cannot measure the impact using performance counters and energy counters.
- Models should use libraries to utilize the operating system functionalities. This factor can significantly reduce development and debug time.
- System emulators should be used based on FPGAs (field programmable gate arrays) to explore the design space. This way, programmers can estimate the necessary resources for the projects.

2.5. Programming Architectures: An abstract of programming architectures can be described as the following:

Flynn's Taxonomy described four possible programming architectures: Single Instruction, Single Data (SISD); Single Instruction, Multi Data (SIMD); Multi Instruction, Single Data (MISD); and Multi Instruction, Multi Data (MIMD).[10]

Single Instruction, Single Data (SIMD): In this system, one stream of instructions processes one stream of data. This architecture is known as *von Neumann* model, where a central processing unit (CPU) is used to process stored-instructions, and data (see Figure 4). This model is virtually used in all single-processor computers.

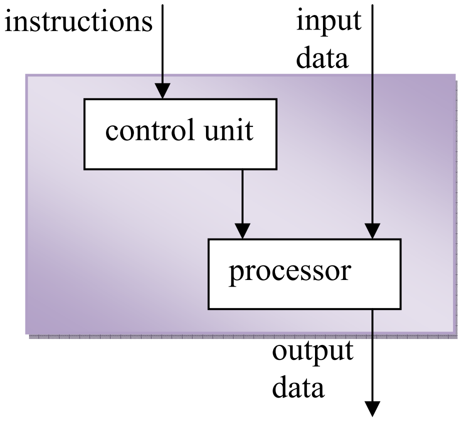


Figure 4: Illustration of instruction and data streams in SISD architecture

Single Instruction, Multiple Data (SIMD): One instruction stream concurrently broadcasts to multiple processors and each processor has its own data stream (see Figure 5). The CPP DAP Gamma II and Quadrics Apemille are the recent examples of this model. This architecture is deployed in specialized applications, which process fine-grained parallelism such as vector computations or digital signal processing.

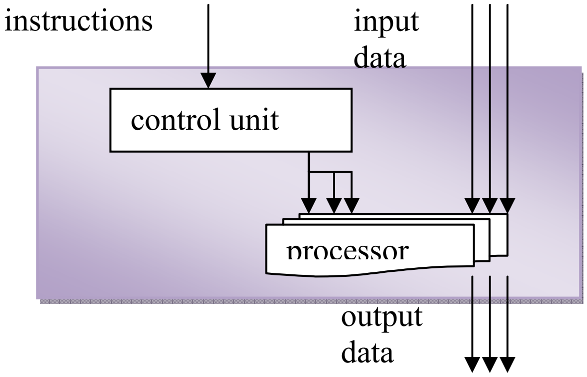


Figure 5: Illustration of Instruction Stream and Data Streams in SIMD architecture

Multiple Instruction, Single Data (MISD): no well-known systems using this model. This model can be ignored.

Multiple Instruction, Multiple Data (MIMD): In this system, each processing element has its own stream of instructions operating on its own data (see Figure 6). This architecture is the most general model, in that other models can be mapped onto MIMD architecture. The most majority of parallel computers nowadays fit onto this model.

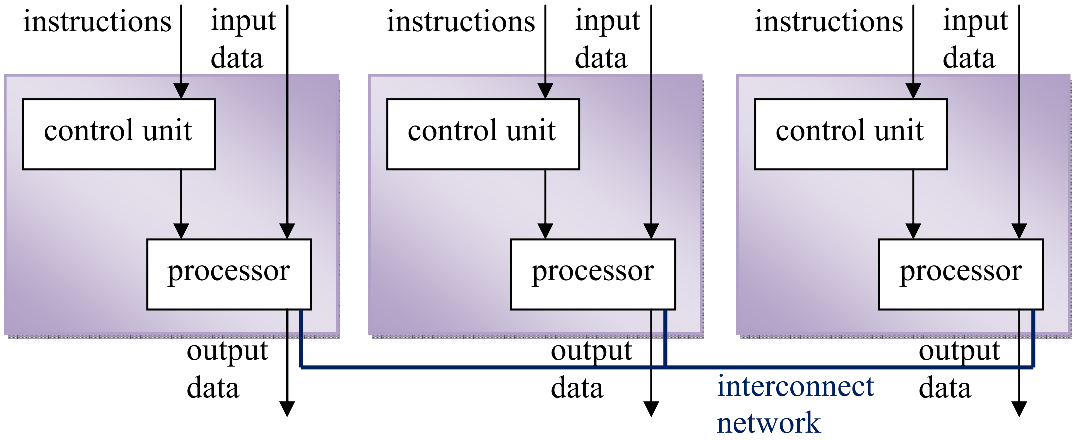


Figure 6: Illustration of Instruction Streams and Data Streams in MIMD Architecture

In the late 1990s, the programmers tend to converge predominantly on two different environments for parallel programming: OpenMP for shared memory and MPI for message passing [10].

OpenMP's implementations are available for Fortran, C, and C++. It performs parallel executions based on compiler directives. In OpenMP, compiler directives are often added to sequential code to perform parallelism. For example, compiler generates parallel instructions to execute iterations of a loop when a compiler directive is added around the loop. The compiler is responsible for the creation and management of threads. It tends to work well on SMP systems. However, because of the lack of non-uniform memory access time descriptions, OpenMP does not work well on ccNUMA or distributed-memory systems.

MPI, on the other hand, is a set of library routines that provide process management, message passing, and collective communication operations such as barrier, broadcast, and reduction. MPI programs are difficult to write because programmers are responsible for data distribution and the communication between processes. MPI is a good fit for MPPs (massively parallel processors) and other distributed-memory systems because it supports distributed memory characteristics.

Neither OpenMP nor MPI alone can be used for hybrid architectures that have more than one multiprocessor node. Each node has multiple processes and a shared memory. Nodes are part of a large system with separate address spaces. The OpenMP model does not handle non-uniform memory access times correctly and its data allocation can lead to poor performance on non-SMP machines. On the other hand, MPI does not construct data structures residing in a shared memory. The combination of both OpenMP and MPI models is used to compromise a solution for the hybrid system. OpenMP is used on each shared-memory node and MPI is used between nodes.

Recently, OpenCL becomes an emerging new standard for applications across processor cores and GPUs (graphics processing units). In some scientific applications, this technology has been proven to provide a significant speed up of the processing time for parallelizable tasks. For example, one biological molecule analysis called *Molecular Boundary Value Computation* took 56.7 s when using a single high speed CPU, while it took 4.76 s on an eight-core Mac Pro with Hyper Threading enabled (16 threads), and it took 0.17 second when using Nvidia GTX 285 [15].

These facts show that each language can be well fit in certain hardware architectures, and application types. When designing a parallel system, programmers need to carefully consider which language to use to satisfy the hardware platforms and the application types. Implementation cost, on the other hand, is another factor that needs to be considered to predict the success of the system. OpenMP, MPI, and OpenCL are three different flavors of parallel programming. Each has advantages in certain hardware architectures, or certain types of applications. In this project, the performance of these languages will be compared based only on the application types and assumes that the hardware platform is unchanged in all experiments.

3. Problem Addressed

Parallel programming uses a pattern language to achieve productivity, efficiency and accuracy. The scope of this project is limited to providing comparisons of performances between specific languages and their implementation costs which are determined by their complexity and development time.

Experiments in this project will be performed on a 64-bit Windows Vista operating system, with Intel ® Core ™ 2 Duo CPU. Intel Thread Profiler and Vtune will be used to compare performances and efficiencies between languages.

4. Related Works

Recent research has discovered the human central in parallel programming. This is one of the most important factors in parallel programming. In fact, the three broad categories of a parallel programming library are faster programs, larger data, and easier development [19]. These factors are equally important in parallel programming to produce a system with high performance, powerful computing capabilities and easy to maintain and extend.

Programmers will play the main roles to produce software programs. Researchers have learned how to help novice parallel programmers to gain more skills in parallel programming. Novice parallel programmers tend to group problems based on the difficulty and domain type of the problems while experts consider the solution types for the problems [19]. Understanding of the human factor in parallel programming is promising the success in software industry. This paper is intended to bring overall knowledge about parallel programming to programmers with both solution-based and problem-domain-based approaches.

Unlike other parallel programming tools, which intend to provide programmers ways to convert the sequential programs into parallel pattern execution with parallel code hidden [26], [27], [28], [29], [30], this research approaches differently. The template- or pattern-based approaches do not provide an easy way to maintain or extend a software project. Programmers are struggling with debugging and tuning performance and the worst of all they have to deal with features that don't have a pattern/template supported.

Several researchers have compared the performances of OpenMP and MPI programs on shared memory machines [31], [32], [33]. Their common results showed the advantages of OpenMP on a shared memory machine over MPI. In this research, the comparison of OpenMP and MPI are based on different hardware platforms. OpenMP is used in the shared memory paradigm, while MPI is used in distributed memory paradigm.

Instead, this project provides programmers with knowledge and hands-on experience in parallel programming in different technologies such as shared memory, distributed memory, and heterogeneous platform programming [13], [14], [15], and [16]. The comparison of performances of different algorithms and the implementation costs across parallel API libraries provide programmers with knowledgeable and flexible ways of implement parallel software systems and the system extensions.

5. Approach

For an accurate comparison of performance and implementation costs between languages, algorithms are defined and implemented in different languages, and different metrics are used to measure performance and software complexity.

5.1. Algorithms

In this section, the chosen algorithms will be described in detail, specifically the decomposition of independent tasks and data to achieve parallelism of program execution. The following are the descriptions of some concrete arithmetic algebra, data sorting, and graphics algorithms.

5.1.1. Maclaurin Series Computation:

One example of Maclaurin Series is the calculation of exponential $\boldsymbol{e^x}$:

$e^x = 1 + x + x^2/2! + x^3/3! + ... + x^n/n!$

If processing this series sequentially, *O(n)* time is needed for a system to complete the computation. Observe that each term of the series can be calculated independently. If each term of the series is calculated simultaneously (assuming number of processor equal to the size *n* of the series), the execution time can be reduced to *O(1+1)*, where the first clock cycle is to calculate *n* terms and the second clock cycle is to sum up the *n* terms.

In reality, the number of processors, p, are normally less than n. In order to divide the computation evenly among processors, n/p term-chunks are assigned to processors. In a worst case scenario, the execution time is $O((q+1)+1)$, where q is the whole number of dividing n by p, and one additional clock cycle is for the remainder of dividing n by p, and another clock cycle is for the sum of the results from p processors.

The algorithm for p processors calculating n terms of a Maclaurin Series of exponential $\boldsymbol{e^x}$, where $p < n$, is described in the following pseudo code:

Initially:

x is a given rational number

n is the number of terms in the series.

p is the number of processors in the system.

m is the number of terms in the series that are assigned to processor p_i, where i is an integer and $0 \leq i < p$.

$sum_i = 0$. sum_i is the partial result of m terms from processor p_i.

$sum = 0$. is the final result of the computation.

j is index of the first term and $M\ (= j + m)$ is index of the last term that are assigned to p_i.

Code for p_i:

```
for $j$; until $j \leq M$
    $sum_i = sum_i + x^j$ / factorial($j$)
    increment $j$
```

Synchronized code:

```
$sum = \sum sum_i$
```

5.1.2. Dot Product of two vectors

Let *A* and *B* be the vectors of size *n*. The scalar dot-product *S* of *A* and *B* is:

$S = S = A_1*B_1 + A_2*B_2 + \ldots + A_n*B_n$

Time complexity analysis of Dot-product is as similar as that of Maclaurin Series. The algorithm for *p* processors calculating a dot-product of vectors *A,B*, of size *n,* where $p < n$, is described in the following pseudo code:

Initially:

A and *B* are the given vectors.

n is the of the vectors *A, B*.

p is the number of processors in the system.

m is the number of terms in the series that are assigned to processor p_i, where *i* is an integer and $0 \leq i < p$.

$sum_i = 0$. sum_i is the partial result of m terms from processor p_i.

$sum = 0$. is the final result of the computation.

j is index of the first term and $M (= j + m)$ is index of the last term that are assigned to p_i.

Code for p_i:

```
for $j$; until $j \leq M$
    $sum_i = sum_i + A_j*B_j$
    3:  increment $j$
**Synchronized code:**
    4: $sum = \sum sum_i$
```

5.1.3. Bubble Sort or Odd-Event-Transposition Algorithm

Bubble sort algorithm is for sorting a sequence of data $S = (a_1, a_2, ..., a_n)$ by comparing and exchanging the neighbor elements a_i , where $1 \leq i \leq n$, in the sequence *S*.

If processing the compare-exchange operations sequentially, there are first $n - 1$ operations for sequential pairs of elements $(a_1, a_2), (a_2, a_3),..., (a_{n-1}, a_n)$. As a result, the element which has the largest value is moved to the end of the sequence *S*. In the next iteration of compare-exchange operations, the last element of *S* can be omitted. Consider $S' = (a'_1, a'_2, ..., a'_{n-1})$ as a transformation of *S*. Repeat the compare-exchange operations. Sequence *S* is sorted after $(n - 1)$ iterations. The execution time for sequential Bubble sort is $O(n^2)$.

The bubble sort algorithm can be executed in parallel. The modified algorithm is known as *the odd-even transposition.* Assume that *n* processors operate on the bubble sort algorithm. Each processor holds one element of the sequence and compare-exchange operations occur between neighboring processors. In order to perform compare-exchange operations independently, there are two different rules that apply to each processor – odd and even iterations. In other words, the processors with odd or even indices will perform the compare- exchange operation with the right or left neighbors in odd or even iterations, respectively.

As a result, the following are snapshots of operations between the neighbors in odd iterations and in even iterations.

Odd iterations: $(a_1, a_2), (a_3, a_4), \dots, (a_{n-1}, a_n)$ (if n is even),

Even iterations: $(a_2, a_3), (a_4, a_5), \dots, (a_{n-2}, a_{n-1})$.

After n iterations, the original sequence is sorted. Time complexity of this algorithm is $O(n)$.

Odd-even transposition algorithm for n processors to sort n-element sequence:

Initially:

A sequence S has n unsorted elements. There are n processors and each has an ID from one to n. Each element in S is assigned to each processor.

$i = 1$, where i is the index of the algorithm's iteration and $1 < i < n$.

$P_{odd,}$ and P_{even} are the processors which have odd and even ID, respectively.

Loop the iteration i of the algorithm from 1 to n.

In odd iteration: P_{odd} does compare-exchange data with its right neighbor, and P_{even} does compare-exchange data with its left neighbor.

In even iteration: P_{even} does compare-exchange data with its right neighbor and P_{odd} does compare-exchange data with its left neighbor.

End of loop.

Synchronized code:

Put result data from processors (after sorting operation) back to sequence *S*.

However, if the number of processors is p and $p < n$, where n is the number of elements in a sequence *S*, then *S* will be segmented into p sub-sequences of size n/p. Each sub-sequence is assigned to each processor. The algorithm's iteration will loop through p, the number of sub-sequences. In each iterations, there are sub-tasks to sort $(2n/p)$ data from two neighbor-processors. The time complexity is $O(p(1+(n/p)^2)$.

Odd-even transposition algorithm for p processors to sort n-element sequence, where $p < n$:

Initially:

A sequence *S* has n unsorted elements. There are n processors and each has an ID from one to n. Each element in *S* is assigned to each processor.

$i = 0$, where i is index of the processors $(0 < i < p)$, and algorithm's iterations are i.

P_{odd}, and P_{even} are the processors which have odd and even ID, respectively.

Loop the iteration i of the algorithm from 0 to p-1.

In odd iteration:

P_{odd} and its right neighbor put their sub-sequences together to sort and then the sorted data is split into two sub-sequences. P_{odd} gets the first half sub-sequence or the min, and the right neighbor gets the second half sub-sequence or the max.

In even iteration: P_{even} and its right neighbor put their sub-sequences together to sort and then the sorted data is split into two sub-sequences. P_{even} gets the first half sub-sequence or the min, and the right neighbor gets the second half sub-sequence or the max.

End of loop.

Synchronized code:

Put result data from processors (after sorting operation) back to sequence *S*.

5.1.4. Graphics Displays

In a graphics problem, assume there is no limitation for a system to allow multiprocessors to render an *n*-pixel bitmap. In theory, if it takes time *t* for a single processor to draw one pixel, then it will take *n*t* for a single processor to draw a bitmap of size *n* pixels. In other words, the time complexity for a single processor to draw an *n*-pixel bitmap is *O(n)*.

It is possible to use an *n*-processor system to render an *n*-pixel bitmap, assuming each processor equally draws one pixel of the bitmap. The time necessary to complete *n*-pixel bitmap is *t* and time complexity is *O(1)*.

In reality, the number of processors is *p* and *p* is typically much smaller than the number of pixels to be drawn, *n*. Therefore, the time complexity for *p* processors to draw an *n*-pixel bitmap is *O(n/p)*.

Algorithm for p processor to draw n bitmaps:

Initially:

Assigned n/p bitmap's indexes or positions to p processors.

$i = 0$, where i is the index or ID of the processors. $0 < i < p$.

Code for P_i: draws n/p bitmaps.

5.2. Software complexity metrics

This section describes and compares the complexities the languages. The time to setup a language framework, the number of libraries needed for a language, the length of the language specification, and the parallel structures in a program are the metrics used to measure the complexity of a language.

5.2.1. Compilation Environment Settings

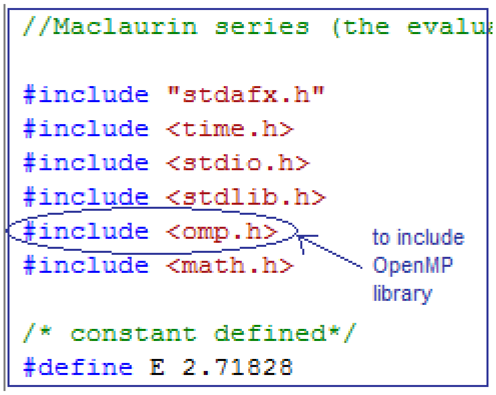
Environmental settings of the languages are carefully described in this section. The complexity of a language-framework bases on the number of components that users need to install, and time consumed to setup the framework.

5.2.1.1. OpenMP Environment Setting

OpenMP's API and library are embedded in Microsoft Visual Studio 2008. This fact makes OpenMP very handy. Developers who use Visual Studio in Windows platform can easy enable OpenMP by including OpenMP's library to the code and setup the project's properties.

The following are steps to enable OpenMP in Visual Studio: Enable the *OpenMP Support* flag in the *Language* sub-tab under the *C/C++* sub-tab of the *Configuration Properties* tab in the project's *Property Pages*. Also, users need to include OpenMP's header file (omp.h) in source files (*.cpp files). Once users have these two settings in Visual Studio, the OpenMP's API is ready to use.

The following are screen captures of the "include" part of the OpenMP source code and project's property enabling OpenMP.

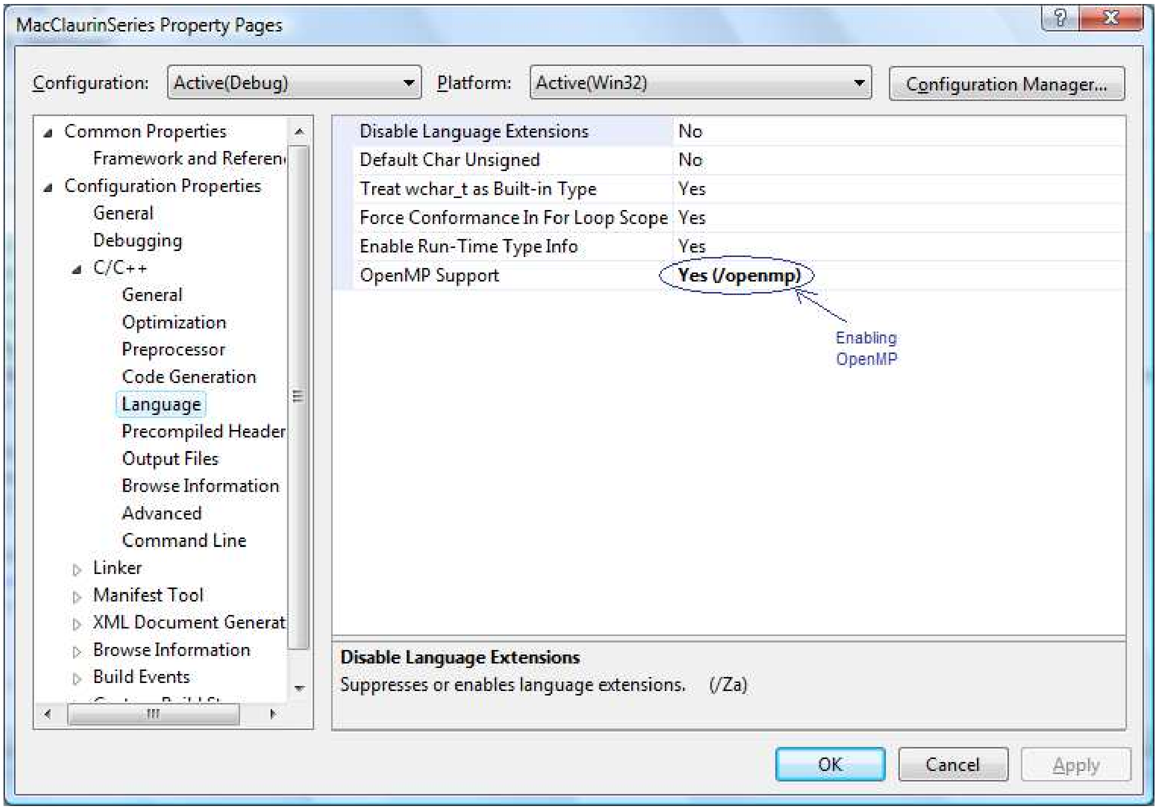


```
//Maclaurin series (the evalu

#include "stdafx.h"
#include <time.h>
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <math.h>

/* constant defined*/
#define E 2.71828
```

Figure 7: OpenMP's header file is included in source code



MacClaurinSeries Property Pages

Configuration: Active(Debug)

Platform: Active(Win32)

Disable Language Extensions	No
Default Char Unsigned	No
Treat wchar_t as Built-in Type	Yes
Force Conformance In For Loop Scope	Yes
Enable Run-Time Type Info	Yes
OpenMP Support	Yes (/openmp)

Disable Language Extensions
Suppresses or enables language extensions. (/Za)

Figure 8: Visual C++ Project Setting to Enable OpenMP

To summarize, OpenMP needs two simple actions from Visual Studio users. There is no installation necessary for the framework and the environmental programming can be achieved in a second.

5.2.1.2. MPI Environment Setting

MPI libraries can be obtained by installing the correct set of MPI for the correct system platform. In this project, the platform used is 64-bit Windows Vista and the MPI installation packet is MPICH2. The installation packet for 64-bit Windows Vista can be downloaded from the following link: http://www.mcs.anl.gov/research/projects/mpich2/downloads/index.php?s=downloads

After installing MPI, the following are the steps for the creation of a console application in MPI using Visual Studio 2008 and in a 64-bit Window Vista.

First, create a Win32 console application. Open the project properties window to set the following parameters to setup MPI library.

In the *General* tab, modify the *Character Set* to *Use Unicode Character Set*. This setting is for the conversion of the data type of variables *argc* and *argv* to be used in MPI program. Then, set the *Additional Include Directories* to point to the *include* folder of MPI library. This parameter can be found in the *C/C++* tab. Set *Additional Library directories* to point to *lib* folder of the MPI installed folder. Finally, set the *Additional Dependency* to contain *mpi.lib*. These four steps should be sufficient for a 32-bit Windows application. The following are steps to convert the Win32 Console to be compiled in 64-bit system.

Appendix DAppendix DAppendix DAppendix DAppendix DAppendix DAppendix DFrom project property window, open *Configuration Manager* window; create new *Active solution platform*; choose *X64* platform from the drop-down menu; also, create new *Platform* and choose *x64* platform. These additional operations set the project to 64-bit platform application.

Although there are about six steps in the procedure of creating an MPI project in Visual Studio, MPI environment needs only one library installation. The time needed to download and install the library is about one hour.

5.2.1.3. OpenCL Environment Setting

OpenCL was intended to be platform independent. It provides the API framework in different hardware and Operating System platforms. Not only do the differences in the CPU chips and Operating Systems matter, but also do the differences in the GPU chips that require OpenCL's library, CUDA's library and the device driver for the GPU to be accurately selected. This fact makes the task of setting OpenCL framework extremely difficult. Users must carefully select the correct version of OpenCL, CUDA libraries, and the GPU driver according to their Operating System and CPU, GPU hardware.

OpenCL was originally developed by Apple. It was then submitted to a non-profit technology consortium, Khronos Group, to review and manage the implementation across technical teams of AMD, Intel, Nvidia and more.

In order to set up OpenCL environment in Microsoft Visual Studio 2008, users need to get the required hardware and the accordant SDKs (software development toolkits).

In this project, Intel Core 2 Duo is the host system; Nvidia GeForce 9200 M GS is the graphics device; 64-bit Windows Vista is the operating system; and OpenCL libraries are from Nvidia. The Nvidia libraries include Nvidia Computing SDK, OpenCL SDK version 2.3, and Nvidia display driver version 190.89.

Notes: Nvidia's SDK and driver can be downloaded from the following link: http://developer.nvidia.com/object/opencl-download.html. Before installing the Nvidia SDKs, users should make sure to uninstall any previous version of SDK. Also, the antivirus software must be disabled while installing Nvidia driver or else the driver cannot be installed correctly.

Depending on hardware platforms, users have to select the correct vendor library packets to install. In addition, there are three different packets need to be installed to achieve the framework setting. The time necessary for downloading and installing the framework is about three hours. These tasks make OpenCL the most complex language in term of environmental setting. The necessary installation packets include:

- Nvidia drivers for 64-bit Windows.
- GPU Computing SDK.
- CUDA Toolkit.

After installing OpenCL packets, the following are the steps for the creation of a console application in OpenCL using Visual Studio 2008 and in a 64-bit Window Vista.

First, create a Win32 console application in either release or debug mode. Open the project properties window to modify the following parameters to setup the OpenCL libraries.

Then, set the *Additional Include Directories* to point to the *include* folders of OpenCL libraries. The steps of library setting is as same as that of MPI setting, except there are two different paths from the installed packet are needed. They are "..\\NVIDIA Corporation\NVIDIA GPU Computing SDK\OpenCL\common\inc", and "..\\NVIDIA Corporation\NVIDIA GPU Computing SDK\shared\inc". Set *Additional Library directories* to point to *lib* folder of the OpenCL installed folder. There are three different paths will be needed. They are "..\\NVIDIA Corporation\NVIDIA GPU Computing SDK\OpenCL\common\lib", "..\\NVIDIA Corporation\NVIDIA GPU Computing SDK\shared\lib", and "..\\NVIDIA Corporation\NVIDIA GPU Computing SDK\OpenCL\common\lib\x64". Finally, set the *Additional Dependency* to contain *oclUtils64D.lib OpenCL.lib shrUtils64D.lib*. These nine steps should be sufficient for a 32-bit Windows application. There are two more additional steps to set the project to compile in 64-bit system.

These steps are as same as the settings in MPI. Totally, there are eleven steps necessary to setup for an OpenCL project.

Additionally, in order to compile the Nvidia sample codes, there will be more parameters that need to be modified. One example is the *Runtime Library*. It must be set to *Multi Threaded* for the compiler to recognize some library calls.

5.2.2. Framework Complexities

The complexity of a framework (or API) depends on the number of APIs in its specification document. The more APIs are defined in the framework the more complex and harder for a new parallel programmer to learn. This factor will be used as a metric to measure the complexity of languages.

5.2.2.1. OpenMP Specification Summary

OpenMP's specification can be found from the following link: http://www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf. The OpenMP specification document contains over sixty pages. It describes fifteen directives, eight clauses, thirty one runtime library routines, eight environment variables, eight operators legally allowed in a reduction, and five schedule types for the loop construct. The total number of APIs in OpenMP is seventy five. The OpenMP user guide can also be found from the following link:

http://www.nd.edu/~hpcc/solaris_opt/SUNWspro.s1s7/SUNWspro/prod/lib/locale/C/html/manuals/pdf/openmp.pdf

This document has about fifty pages describing the syntax and use cases of OpenMP. Overall, the OpenMP specification and user guide have simple rules and are easy to use.

5.2.2.2. MPI Specification Summary

There are two commonly used MPI's implementations. They are LAM/MPI (or LAM) and MPICH (or MPI). Both can be downloaded free of charge. MPICH2 was used in this project. It can be downloaded at http://www.mcs.anl.gov/research/projects/mpich2/downloads/index.php?s=downloads.

The MPI Specification document includes sixteen chapters, where the *Data-types* chapter alone occupies about sixty-pages. Unlike OpenMP, MPI has complex point-to-point communication, data-types, process synchronization, and communication space to support library functions, process topologies, and environmental management. Its specification document contains over six hundred pages. The length of the document can provide users with a broad idea about the complexity of the language.

5.2.2.3. OpenCL Specification Summary

There are many different sources of OpenCL frameworks that can be downloaded free of charge from Internet. Depending on the operating systems, CPU chips, and graphic cards, users can download and install OpenCL frameworks from different vendor sources. OpenCL specification, however, can be downloaded from Khronos source as the following: http://www.khronos.org/registry/cl/specs/opencl-1.0.29.pdf.

Specifications from different vendors can be different in number of library interfaces depending on the vendor's interested. For example, Nvidia does not support creating CPU's context and CPU optimizations because it manufactures graphics cards only.

The OpenCL Specification has about three hundred pages. It includes two hundred twenty seven constants; five hundred twenty one functions; fifty data structures and typedefs; seven qualifiers; and forty directives and macros. OpenCL intensively supports mathematical functions such as trigonometric and statistical functions.

Due to the young age of OpenCL, the language specification is still developing and therefore the length of its specification is a lot shorter than MPI's specification. Not to mention that MPI exists in the market for about twenty years.

5.2.3. Abstractions of Parallelism in Languages:

This section describes the parallelism of the languages using data flow diagrams. These illustrations abstract the ways that multi processes can be deployed simultaneously in the languages. The number of responsibilities of programmers in the thread synchronizations determines the complexity of the languages.

5.2.3.1. Parallelism in OpenMP

Programs in OpenMP are normally driven by one main thread or process. The compiler is responsible for involving other processors or threads and balancing tasks among processes within the parallel regions. Threads can access their own private memories and the shared memories within the parallel scopes. Programmers are responsible for specifying the parallel regions in the program, and determining the private and shared memories in the parallel regions. Programmers are also responsible for synchronizing the shared memories among threads. The optimizations of parallel performance can be accomplished by compiler via release mode of the compilation.

In general, compiler creates threads in the parallel regions of the program and balances tasks among processes. Programmers only need to perform the mutual exclusion in the use of shared variables. These facts make OpenMP the least complex compare to other languages.

5.2.3.2. Parallelism in MPI

Unlike OpenMP, parallel region in MPI exists in the entire of the programs. Threads are created by MPI library. These threads remain in the programs until the end of programs. Programmers are responsible for dividing tasks, and creating communication channels within processes. Programmers are also responsible for the performance tuning and providing a minimum of data transferred between processes. This way, program can reach the best performance of the computations.

Although, programmers don't create threads in MPI programs, the balancing tasks among processes and designing distributed memories within processes are the complex responsibilities. The parallelism structure is therefore more complex than that of OpenMP.

5.2.3.3. Parallelism in OpenCL

OpenCL has different parallelism approach. The parallelism of the language deploys the rapidly increasing in number of hardware cores not only on the CPU chips but also on the GPU chips. OpenCL programs contain two main components, the host system and the device. The host system performs the main operations of the program in sequence and the device performs the parallel tasks. Programmers have to create and setup the workflow from the host system to the device so that the orders of instructions are reserved and data from host memories can be correctly copied to the memories of the kernels or hardware threads.

The structure of an OpenCL program is rather complex. Five portions of a program must be explicitly defined [15] as the following:

a. Initialization: Selecting a device or creating a context in which the parallel computations occur.

```
cl_int err;
cl_context context;
cl_device_id devices;
cl_command_queue cmd_queue;

err = clGetDeviceIDs(CL_DEVICE_TYPE_GPU,  1, &devices, NULL);

context = clCreateContext(0, 1, &devices, NULL, NULL, &err);

cmd_queue = clCreateCommandQueue(context, devices, 0, NULL);
```

b. Allocation: memory allocation for buffers that are used on device and copy data from host memory to the device.

```
cl_mem vectorA = clCreateBuffer(context, CL_MEM_READ_ONLY,
buffer_size, NULL, NULL);

err = clEnqueueWriteBuffer(cmd_queue, vectorA, CL_TRUE, 0,
buffer_size, (void *)ax, 0, NULL, NULL);

cl_finish(cmd_queue);
```

c. Program and Kernel Creations: Binary program for the kernel will be built by the compiler. This portion of the program takes at least about a third of a second. In the performance analysis of OpenCL, the time to build the kernel program is excluded.

```
cl_program program[1] ;
cl_kernel kernel[1];

program[0] = clCreateProgramWithSource(context, 1, (const char
**)&program_source, NULL, &err);

err = clBuildProgram(program[0], 0, NULL, NULL, NULL, NULL);

kernel[0] = clCreateKernel(program[0], “add”, &err);
```

d. Execution: Arguments to the kernel are set and the kernel is executed on all data mapped to it. In this portion of the program, parallel computations are actually occurred.

```
size_t global_work_size[2]; local_work_size[2];

global_work_size[0] = x_dim; global_work_size[1] = y_dim;

local_work_size[0] = x_dim/2; local_work_size[1] = y_dim/2;

err = clSetKernelArg(kernel[0], 0, sizeof(cl_mem), &vectorA);

err = clEnqueueNDRangeKernel(cmd_queue, kernel[0], 2, NULL,
&global_work_size, &local_work_size, 0, NULL, NULL);
```

e. Clean up: Read back the results to host and clean-up memory

```
err = clEnqueueReadBuffer(cmd_queue, result_src, CL_TRUE, 0,
grid_buffer_size, dest_ptr, 0, NULL, NULL);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(cmd_queue);
clReleasecontext(context);
```

In overall, with five portion structure of program, OpenCL is proven the most complex language among other two languages, i.e.: OpenMP and MPI.

6. Architecture/Implementation

In this chapter, the implementations of the aforementioned algorithms will be described in detail for three languages: OpenMP, MPI, and OpenCL.

6.1. OpenMP:

OpenMP stands for open specification for multi-processing. It is a collaborative work between interested parties from the hardware and software industry, government, and academia. OpenMP is an application program interface (API) that is used to direct multi-threaded, shared memory parallelism. Its API is specified for C/C++ and Fortran. It is comprised of three main API components: compiler directives, runtime library routines, and environment variables. More importantly, most major platforms support this API, including Unix/Linux and Windows NT.

In order to measure the performance and compare the speedup ratios of OpenMP, some small programs are used. The chosen programs are in different aspects of computations so that the language can be analyzed in a wider range of applications. The programs represent for arithmetic computations, sorting algorithms, and graphic rendering. Maclaurin calculation of e^x (up to five hundred terms), and scalar dot product of two vectors (contain five million items in each vector) represent for arithmetic computations; Odd-even transposition sort of sixteen thousand items of a data set represents for complex algorithms; and Rendering graphics simulation represent for graphics displacements. Run-times of algorithms will be measured using time library in Microsoft Visual. Each program will be compiled with and without enabling OpenMP. The run-time with OpenMP enabled is the execution time of the multi-processor simultaneously performing tasks, while the run-time without OpenMP is the execution time of one processor performing the same tasks. These run-times are used to calculate the speedup ratios of algorithms in parallel OpenMP. Speedup ratio formula is the following:

$$\text{Speedup Ratio} = \frac{\text{Sequential Execution Time - Dual Core Execution time}}{\text{Sequential Execution Time}} * 100$$

	Sequential Execution time, s	Dual core – Execution time, s	Speedup Ratio, %
Maclaurin Series, e^x of 500 terms	0.017	0.011	36.9
DotProduct of 2 vectors of 5 billion items	0.016	0.016	0.0 (no speed up)
Bubble Sort of 16,000 items	0.686	0.359	47.67
Graphic Rendering of 10,000 items	1.124	1.126	-0.2 (no speed up)

Table 1: Speedup ratio of algorithms in parallel OpenMP

The execution time of these algorithms is relatively small (less than a second). In order to obtain accurate run-time values for the experiments, each algorithm will be executed in loops and the total run-time will depend on the number of loops. The algorithm run-time is the average value of each loop. In addition, each program takes one command argument, *argv*, as a number of loop-backs at run-time so that the total execution time of the programs can be controlled. The time shown in Table 1 is the total execution time, where the loop-back factor is fixed and is equal to one. See appendix A for setting up Visual C++ 2008 projects to accept the command argument.

6.1.1. Maclaurin Series Calculation in OpenMP

Recall the Maclaurin Series of e^x, the expression is the following:

$e^x = 1 + x + x^2/2! + x^3/3! + ... + x^n/n!$

The calculation of the series includes the calculation of each term, and the sum of n terms. The terms’ calculations are independent from each other and can be set to be simultaneously computed. In this experiment, the number of terms, *n*, is set to be one thousand and the tasks to obtain these terms are equally divided to two processors. Processor 0 calculates the first half of the series in the sequence, and processor 1 calculates the second half. The results from two processors will be sum up to obtain the final value of the series.

The following table will show implementation of Maclaurin Series e^x using OpenMP:

```
/* Here’s the OpenMP pragma that parallelizes the for-loop. */
/* This parallel construct has 4 private variables:         */
/*   1) outer loop index “I” by default                     */
/*   2) inner loop index “j” by explicit declaration        */
/*   3) “nfactorial_value” by explicit declaration          */
/*   4) “nfactorial_base” by explicit declaration           */
/* Schedule claude specify 2 chunks of 500 terms assigned   */
/* statically to 2 processors.                              */
/* Reduction claude synchronized the results of processors  */
/* and sum them up to the final result.                     */
```

```
#pragma omp parallel for \
default(shared) private(I,j,nfactorial_value,nfactorial_base)\
schedule(static,chunk) \
reduction(+:sum)
    for (i=1; I < N; i++)
    {
        nfactorial_value = 1;
        nfactorial_base = 0;
        for(j = 1; j <= I; j++)
        {
            nfactorial_value *= j;
              if(nfactorial_value > 1000)
              {
                  nfactorial_value /= 1000;
                  nfactorial_base += 3;
              }
          }
          sum = sum + pow(x,i) / (nfactorial_value *
                            (float)pow(10.0,nfactorial_base));
    } //end of parallel region and sum holds the result of e^x
```

Table 2: Implementation of Maclaurin Series of e^x using OpenMP

6.1.2. Dot-Product of Two Vectors in OpenMP

Let *A* and *B* are the vectors of size *n*; and the scalar dot product is *S*.

$S = A_1*B_1 + A_2*B_2 + \dots + A_n*B_n$

Like Maclaurin Series, calculations of terms in dot product are independent and can be set to be simultaneously computed. The final sum of *n* products can be obtained by using OpenMP barrier. In this experiment, vectors *A* and *B* have the size of five million coordinates.

The OpenMP program has chopped the dynamic arrays *A* and *B* into chunks of five hundred items of data. Processors *0* and *1* will be assigned to compute each five hundred chunk at a time, and the sum *S* will be synchronized between two processors every five hundred chunk of data of each processor. In other words, each processor will sum up five hundred products in a chunk and then OpenMP's barriers will sum up the results so far from the processors every two chunks (one thousand items) of data.

The following table will show implementation of Scalar Dot Product of two vectors using OpenMP:

```
/* Here's the OpenMP pragma that parallelizes the for-loop. */
/* This parallel construct has 1 private variable:          */
/*   1) outer loop index "I" by default                     */
/* Schedule claude specify chunks of 500 terms assigned     */
/* statically to 2 processors.                              */
/* Reduction claude synchronized the results of processors  */
/* and sum them up to the final result.                     */

#pragma omp parallel for private(i) schedule(static,chunk)
reduction(+:DotProduct_Val, DotProduct_Pow)
for (i=0; I < MAXITEMS; i++)
{
      long temp = A[i] * B[i];
      if(DotProduct_Pow == 0)
      {
            DotProduct_V = DotProduct_V + temp;
            if(DotProduct_V > BILLION)
            {
                  DotProduct_V = DotProduct_V/BILLION;
                  DotProduct_P = DotProduct_P + 9;
            }
      }
      else if(DotProduct_P >= 9)// DotProduct_P >= 9
      {
            DotProduct_V = DotProduct_V + temp/DotProduct_P;
            if(DotProduct_V > BILLION)
            {
                  DotProduct_V = DotProduct_V/BILLION;
                  DotProduct_P = DotProduct_P + 9;
            }
      }
} //end of parallel region. DotProduct_V holds the scalar value
//of dot product of 2 vectors and DotProduct_P holds its power
```

Table 3: Implementation of Dot Product in OpenMP

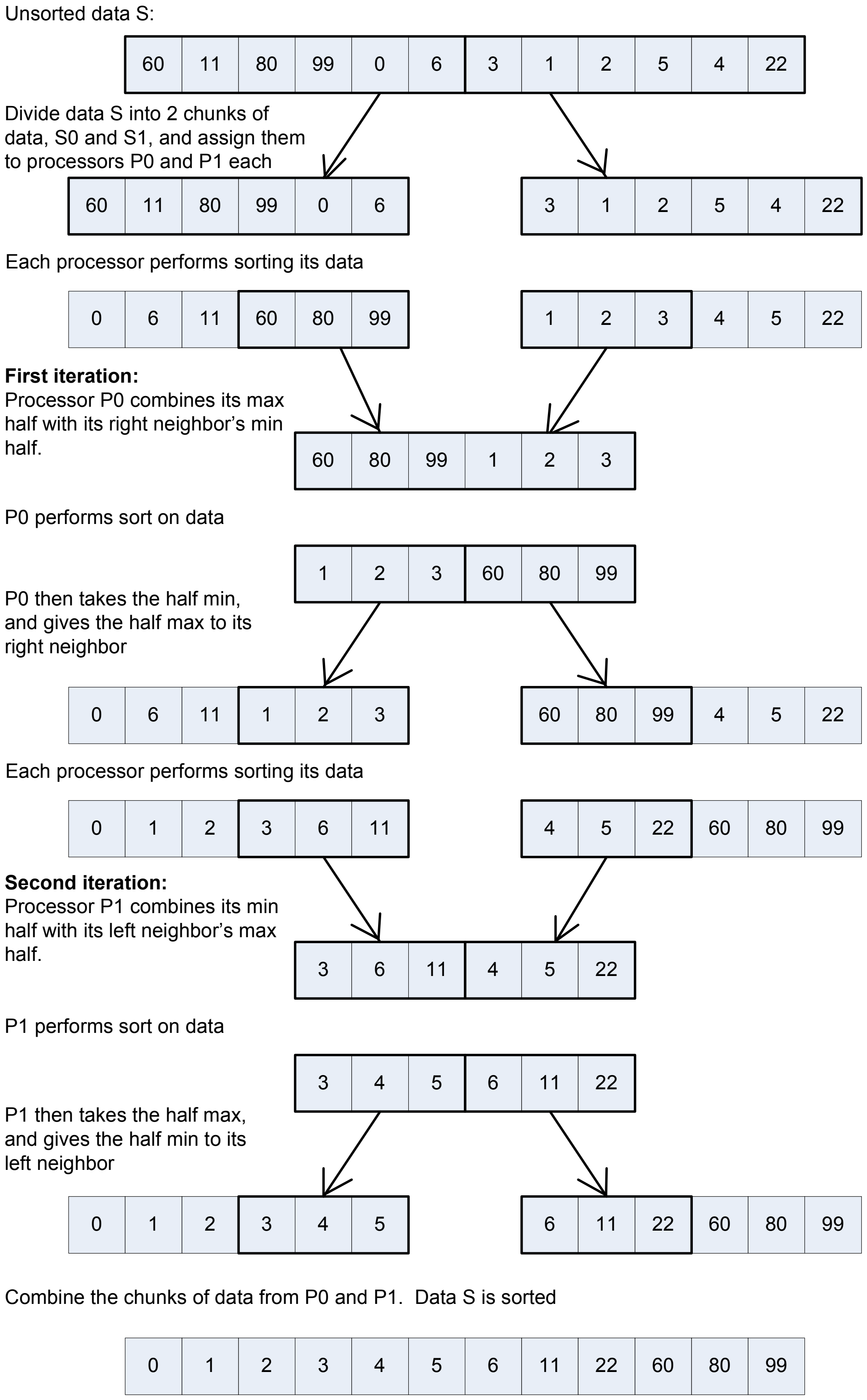
6.1.3. Bubble Sort in OpenMP

Unlike the Maclaurin Series algorithm, Odd-Even Transposition allows the OpenMP's *for* directive to perform parallelism in some blocks and single execution in others. The following implementation is for a system of two processors (P_0 and P_1). The data set has a large number of items, i.e.: sixteen thousands of items and items are out of order. The data set is evenly divided into two halves. The first half of data is assigned to the first processor (P_0) and the second half is assigned to the second processor (P_1). Each processor then performs sorting its chunk of data within the *for*-directive. After sorting the chunks of data, each chunk will be split into two portions, e.g.: max and min portions.

During odd iteration, the combination of the max portion data of processor P_0 and the min portion of its right neighbor, P_1, will be sorted by processor P_0 and then P_0 keeps the min portion of the sorted data and gives the max portion to P_1. Each processor again performs sorting its chunk of data.

During even iteration, the combination of min portion of P_1 and the max portion of its left neighbor, P_0, is sorted by processor P_1 and then P_1 keeps the max portion of sorted data and gives the min portion to P_0. Each processor again sorts its data. The combination of the data chunks from P_0 and P_1 in that order will be sorted after two iterations.

The following is the illustrated algorithm with the set of twelve items of unsorted data:



Unsorted data S:

60 11 80 99 0 6 3 1 2 5 4 22

Divide data S into 2 chunks of data, S0 and S1, and assign them to processors P0 and P1 each

60 11 80 99 0 6 | 3 1 2 5 4 22

Each processor performs sorting its data

0 6 11 60 80 99 | 1 2 3 4 5 22

First iteration:
Processor P0 combines its max half with its right neighbor's min half.

60 80 99 1 2 3

P0 performs sort on data

1 2 3 60 80 99

P0 then takes the half min, and gives the half max to its right neighbor

0 6 11 1 2 3 | 60 80 99 4 5 22

Each processor performs sorting its data

0 1 2 3 6 11 | 4 5 22 60 80 99

Second iteration:
Processor P1 combines its min half with its left neighbor's max half.

3 6 11 4 5 22

P1 performs sort on data

3 4 5 6 11 22

P1 then takes the half max, and gives the half min to its left neighbor

0 1 2 3 4 5 | 6 11 22 60 80 99

Combine the chunks of data from P0 and P1. Data S is sorted

0 1 2 3 4 5 6 11 22 60 80 99

Figure 9: Odd-Even Transposition or Parallel Bubble Sort algorithm

6.1.4. Graphics Rendering in OpenMP

The graphics rendering algorithm does not form fit the scope of OpenMP because OpenMP does not support OpenGL. In this experiment, "print a line of characters" is used to simulate the pixels painted on the screen. The goal is to compare the execution time of the same task when using multi processor and single processor.

The implementation deploys the *for* loop and the *for* directive to display ten thousand lines of text. Although two processors execute simultaneously, they both share the same resource *stdout*. The execution time of the multi-processor has no difference compared to the same program running on a single processor.

6.2. MPI:

MPI stands for message passing interface. It is a portable message-passing standard that facilitates the implementation of parallel applications and libraries. It is used for communications in parallel computers, clusters, and heterogeneous networks. It is widely supported by many hardware vendors and widely used by academic researchers, government laboratories, and industry. MPI not only performs message passing from one process to another, but also synchronizes processes, scatters data across a processes, and sums numbers distributed among a collection of processes.

The same small programs of Maclaurin Series, dot product of two vectors, parallel bubble sort, and graphic rendering will be implemented in MPI so that the performance and implementation cost of each program can be compared.

In these tests, MPI is used in the communication between two processors in a local machine. Like OpenMP, the algorithm of MPI programs will divide the work evenly among two processors in 64-bit Windows Vista. *Processor 0* has the responsibility to divide work evenly to itself and to *processor 1*. It then concludes and merges the final result from itself and from *processor 1*.

6.2.1. Maclaurin Series Calculation in MPI

The total number of terms in Maclaurin Series program in MPI should be the same as the number of terms in the OpenMP program, one thousand terms. Each processor will calculate five hundred terms and sum them up. The final result will be the sum of the results from all processors.

Processor 1 calculates the second five-hundred terms of the series and sums them up. It then sends the result to *processor 0*.

Processor 0 calculates the first five hundred terms of Maclaurin Series and sums them up. *Processor 0* then waits for the result from *processor 1* to makes a final sum of the results.

6.2.2. Dot-Product of Two Vectors in MPI

Similar to Maclaurin Series calculation, the dot-product of two vectors of size *n* will be the sum of *n* products of each dimension of the two vectors. The work is evenly divided among two processors, *P0* and *P1*.

Processor 1 calculates the second halve terms, *n/*2, of the dot-product of two vectors. It then sends the result to *processor 0*.

Processor 0 calculates the first halve terms, *n/*2, of the dot-product of two vectors. *Processor 0* then waits for the result from *processor 1* to make a final sum of the results.

6.2.3. Bubble Sort in MPI

Parallel bubble sort or odd-even-transposition will perform the exact algorithm as explained in section 6.1.3 Appendix D(see Figure 9). The algorithm can be described as the following.

A data set of size *n* will be evenly divided to two processors. Each owns an *n/2* subset of the data. Processors can perform serial bubble sort on their set of data and then exchange a subset of its data to the other processor. The synchronization among processors is determined by the odd or even iteration of the loop through the odd or even of processor's ID.

In the first iteration (or even iteration) processors *P0* and *P1* both sort their data sets. Then, processor *P0* performs a send operation of its half-max-data to processor *P1* using *MPI_Send*. Processor *P1* receives data from *P0*, using *MPI_Rec*, and then performs sequential bubble sort on the half-max-data from the left-neighbor with its own half-min-data. *P1* then returns the half-min-data from the sorted buffer to processor *P0*.

In the second iteration (or odd iteration) again both processors sort their data sets. Then, processor *P1* performs a send operation of its half-min-data to processor *P0*. Processor *P0* performs a sequential bubble sort on the half-min-data from the right neighbor and its own half-max-data. It then returns the half-max-data from the sorted buffer to processor *P1*.

Data of size *n* is sorted after two iterations.

6.2.4. Graphics Rendering in MPI

In a manner similar to the OpenMP program, graphics rendering in MPI will be simulated by printing characters to the computer screen. In this test, MPI performs the communication between processors within the same computer. Even though the print out work is divided evenly among processors, the shared standard output blocks the speedup of processors. In other words, each processor is blocked when the other processor is using standard output to print. The result of the MPI test program shows that the multiprocessor does not speed up the process of printing characters. However, in a system of two computers which has two different standard outputs the speed would be expected to improve.

6.3. OpenCL:

OpenCL, or Open Compute Language, is the framework for writing programs that execute across heterogeneous platforms including CPUs and GPUs, and other processors. It is an API designed for massively parallel processing using task-based and data-based parallelism.

OpenCL involves running the program on two different platforms – a host system that includes one or more CPUs to perform tasks, and a device system includes one or more OpenCL-enabled Nvidia GPUs. Nvidia devices are not only for rendering graphics, but also for powerful arithmetic engines, which can run thousands of lightweight threads in parallel. This capability makes them well suited to computations that leverage parallel execution. In order to use OpenCL efficiently, it's important to understand the differences in design between host and device systems in a server.

In a host system, the execution pipelines can support a limited number of concurrent threads. Four quad-core processors today can support sixteen threads in parallel or thirty-two threads if the CPUs support Hyper-Threading. By comparison, the smallest execution unit of parallelism on a GPU device, called a warp, composes thirty-two threads. All Nvidia GPUs can support 768 active threads per multiprocessor, and some GPUs support 1024 active threads per multiprocessor. On devices that have thirty multiprocessors, i.e.: Nvidia® GeForce® GTX 280, will provide more than thirty-thousand active threads.

In addition, threads on a CPU are generally heavyweight entities. The operating system must swap threads on and off execution channels to provide multithreading operations. Context switches, when two threads are switched, are slow and expensive. By comparison, GPUs run extremely lightweight threads. In a typical system, hundreds of threads are queued up in warps. If the GPU processor must wait on a warp of threads, it simply begins executing tasks on another warp. There are no registers and state swapping between GPU threads. Resources stay allocated to a thread until the thread completes its execution.

Both host system and device have Random Access Memory (RAM). On the host system, RAM is equally accessible to all code. On the device, RAM is divided virtually and physically into different types, each of which has a special purpose and fulfills different needs (see Figure 10). Memory optimizations are the most important factor for performance in OpenCL.

Device code can:
- R/W per-thread registers
- R/W per-thread local memory
- R/W per-block shared memory
- R/W per-grid global memory
- Read only per-grid constant memory

Host code can:
- Transfer data to/from per-grid global and constant memories

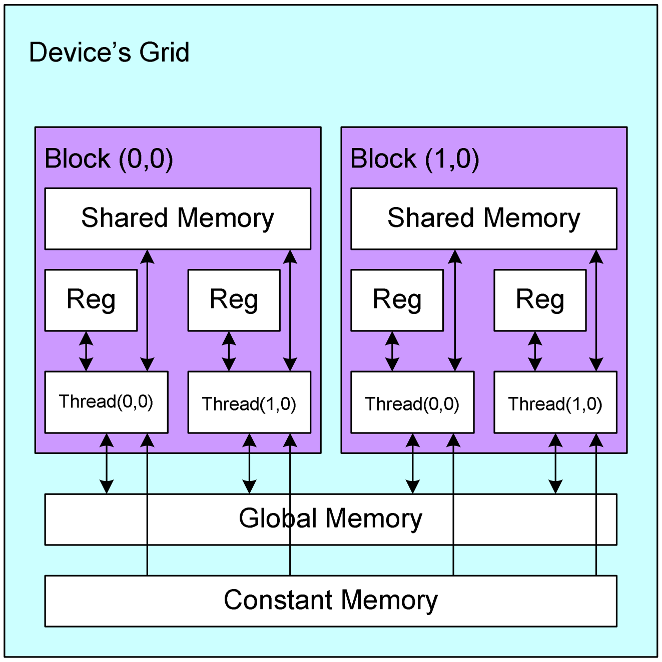


Figure 10: Device memory Architecture

The data transfer rate between device memory and the GPU is much higher (for example, Nvidia GeForce GTX 280 has data rate of 141 GBps) than the rate between host memory and device memory (for example, PCI Express X16 Gen2 has data rate of 8GBps). For the best performance on an application, it is important to minimize data transfer between host and device, and maximize the computations within the device.

The following are the implementation and performance analysis for Dot Product, Bubble Sort, and Graphics Rendering programs in OpenCL.

6.3.1. Dot-Product of Two Vectors in OpenCL

Similarly to other languages, the dot-product in OpenCL is a scalar result of two vectors of size *n* (*n* = 5,000,000). It is the sum of *n* products of each dimension of the two vectors. The work is evenly divided among available processors in GPU chip.

Although the GeForce 9200M GS chip has eight cores, each core has much lower clock rate than the CPU clock rate in the test system. The GPU has the clock rate of 550 MHz while the Intel Core 2 Duo has the clock rate of 2 GHz. This factor explains the longer execution time in OpenCL other languages and even in sequential program.

In addition, the maximum data transfer size between host and device is 33,554,432 bytes (this value was obtained from the execution of the *oclBandwidthTest* program from Nvidia sample codes.) This maximum data size has limited the size of the calculated vectors. This limit explains the maximum vector size in OpenCL program. The vector size cannot be greater than ten millions.

6.3.2. Bubble Sort in OpenCL

Knowing the GPU chip has eight cores, the *Odd-Even Transposition* algorithm used eight processes instead of two. The data set is divided into eight subsets in GPU memory. The algorithm of this experiment is as same as described in section 5.1.3, where number of processor p (= 8), and the size of the unsorted data is n (= 4,000).

The greater number of processors used, the smaller size of chunks of data can be assigned to processes, and therefore, the quicker the processes can perform the sequential sorts in each iteration. After eight iterations, the data set is sorted. The outcome of this experiment is shown in section 7.1.3.2.

6.3.3. Graphics Rendering in OpenCL

In this program, OpenCL and OpenGL are used to display animation graphics. OpenCL was used to compute the next positions, color and light of pixels in a graphical rotating object. The result positions were fed into OpenGL buffer to display the next frame of the graphics. The quicker the next graphical frame was fed to OpenGL buffer the faster the movement of the object. In this program, the performance metric is the time necessary for the animation graphics to complete one cycle of rotation. The performances of the operations with and without OpenCL computation will be compared. The graphics with OpenCL rotates a lot faster than the graphics without OpenCL. The result of this experiment is described in section 7.1.3.3.

7. Experiments

In this chapter, experiment measurements are collected and performance values of different languages are compared. The experiment variables are the performance of the languages in different algorithms; the implementation time of each program; and the complexity of the languages.

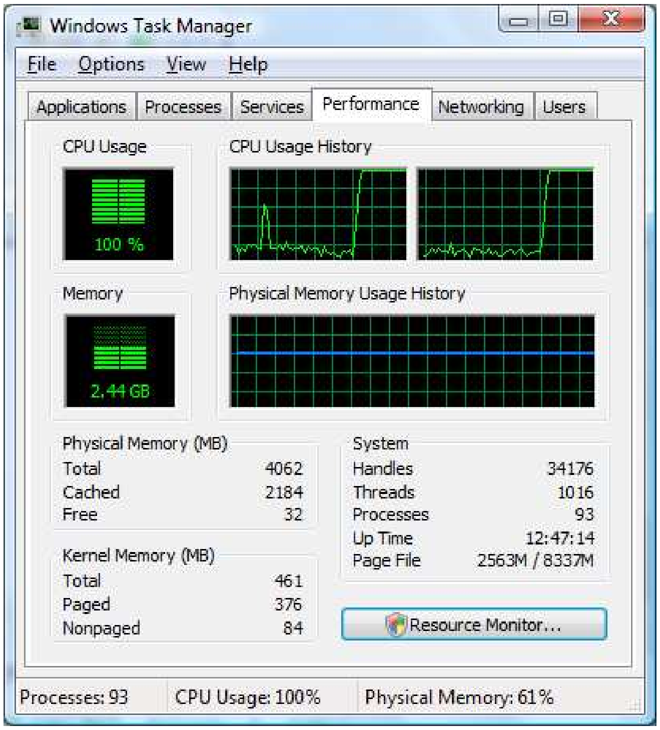
7.1. Performance Analysis:

First, the execution time of each language written for each aforementioned algorithm is carefully examined. The goals of the experiments are to collect the data of algorithm performance versus the data size; discuss the performance speed up factors on particular prototypes; and compare performance across languages.

In general, the performance is measured based on the Visual C++ clock across languages. MPI has its own clock that returns the same value of time from different threads. In MPI programs, both clocks are observed, but the result is based on the MPI clock. In the graphics program that used OpenGL, the performance was measured differently. In animation graphics, performance is measured by the time to complete one cycle of animation. More detail about graphics performance measuring will be discussed in section 7.1.1.3.

7.1.1. OpenMP:

In general, OpenMP programs run with the maximum usage of the CPU resources. The performance optimizations are automatically set by the compiler when programs are built. Figure 11 was captured when running OpenMP's Dot-product of vectors which have the size of fifty million dimensions and the execution was looped back one thousand times. This figure is used as the sample of efficiency of CPU usage when running the OpenMP program.



CPU Usage: 100 %

Memory: 2.44 GB

Physical Memory (MB)	
Total	4062
Cached	2184
Free	32

Kernel Memory (MB)	
Total	461
Paged	376
Nonpaged	84

System	
Handles	34176
Threads	1016
Processes	93
Up Time	12:47:14
Page File	2563M / 8337M

Processes: 93 CPU Usage: 100% Physical Memory: 61%

Figure 11: CPU usage 100% when running parallel OpenMP program

7.1.1.1.OpenMP – Dot-Product Analysis.

In dot-product of two vectors, the execution time increases with the vector size (see Table 4 and Figure 12). From the experiments, the vector's size cannot exceed four hundred million dimensions. The system hanged when the vector size was five hundred million dimensions. Experiments also showed that the larger a vector size is used in the computation, the more memory resource is consumed and the less optimal CPU usage (see Figure 13 and Figure 14).

Observing data in Table 5, we see that OpenMP does not improve the performance of dot products over the sequential program when the vector size is less than twenty million dimensions. The speed up ratio increases significantly when the vector size is twenty million dimensions and the vectors are segmented into chunks of two thousand dimensions. Chunks are segments of the vectors' data that are assigned to one processor at a given time for the computation. The speedup ratios from Table 5 are not the maximum values. These ratios are expected to be further improved at each vector size by changing the chunk size. However, this work is not covered in this project. Table 5 proves that OpenMP does increase the performance compared to sequential dot product programs.

Vector size, dimensions	Execution time, s
5,000,000	0.011
8,000,000	0.016
10,000,000	0.016
20,000,000	0.032
100,000,000	0.187
400,000,000	0.813

Table 4: Dot Product of Two Vectors in OpenMP – Execution Time versus Vectors' Sizes

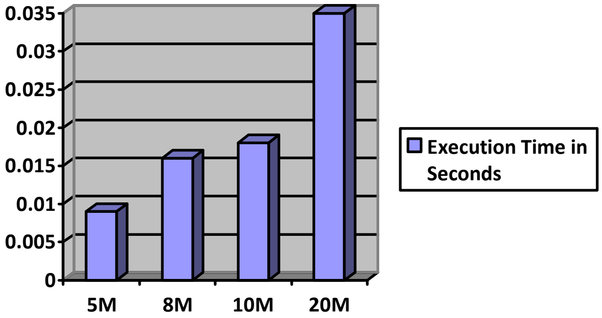
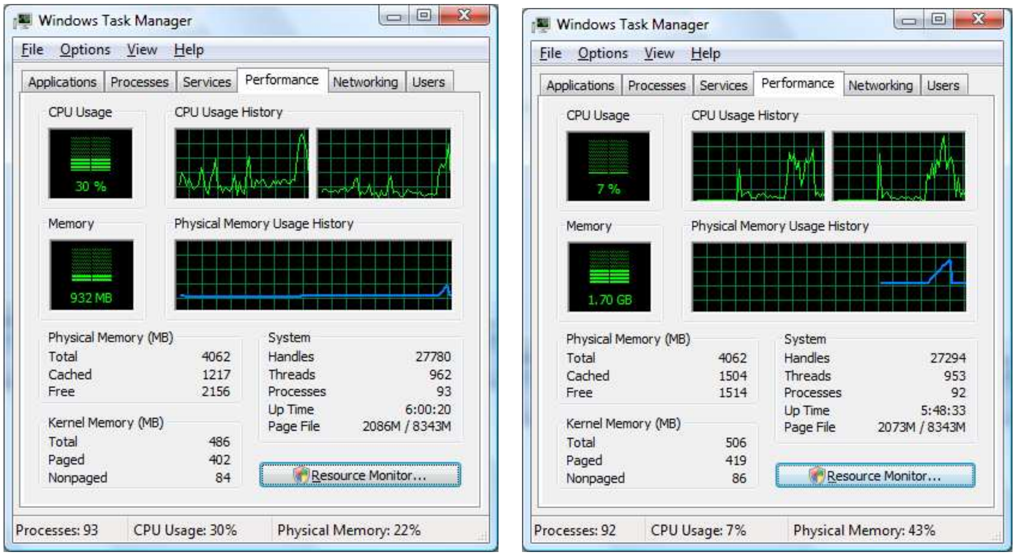


Figure 12: Dot Product Program in OpenMP – Execution Time versus Vectors' Sizes



Left (100M):

CPU Usage: 30 %
Memory: 932 MB

Physical Memory (MB)
Total 4062
Cached 1217
Free 2156

Kernel Memory (MB)
Total 486
Paged 402
Nonpaged 84

System
Handles 27780
Threads 962
Processes 93
Up Time 6:00:20
Page File 2086M / 8343M

Processes: 93 CPU Usage: 30% Physical Memory: 22%

Right (200M):

CPU Usage: 7 %
Memory: 1.70 GB

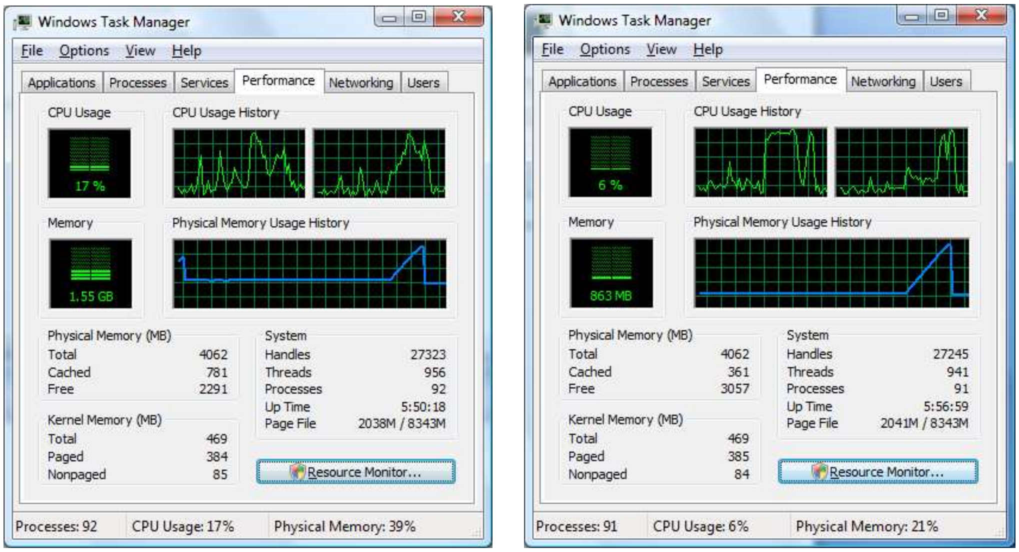
Physical Memory (MB)
Total 4062
Cached 1504
Free 1514

Kernel Memory (MB)
Total 506
Paged 419
Nonpaged 86

System
Handles 27294
Threads 953
Processes 92
Up Time 5:48:33
Page File 2073M / 8343M

Processes: 92 CPU Usage: 7% Physical Memory: 43%

Figure 13: CPU & memory usage when vector size is 100M (left) and 200M (right)



Left (300M):

CPU Usage: 17 %
Memory: 1.55 GB

Physical Memory (MB)
Total 4062
Cached 781
Free 2291

Kernel Memory (MB)
Total 469
Paged 384
Nonpaged 85

System
Handles 27323
Threads 956
Processes 92
Up Time 5:50:18
Page File 2038M / 8343M

Processes: 92 CPU Usage: 17% Physical Memory: 39%

Right (400M):

CPU Usage: 6 %
Memory: 863 MB

Physical Memory (MB)
Total 4062
Cached 361
Free 3057

Kernel Memory (MB)
Total 469
Paged 385
Nonpaged 84

System
Handles 27245
Threads 941
Processes 91
Up Time 5:56:59
Page File 2041M / 8343M

Processes: 91 CPU Usage: 6% Physical Memory: 21%

Figure 14: CPU & memory usage when vector size is 300M (left) and 400M (right)

Data size	Chunk size	Execution time, s Without OpenMP	Execution time, s With OpenMP	Speed up, %
5M	500	0.011	0.011	0.00
10M	1K	0.016	0.016	0.00
20M	2K	0.047	0.032	31.91
100M	10K	0.219	0.187	14.61
200M	20K	0.422	0.358	15.17
400M	40K	Varied: 0.874-0.905	Varied: 0.702-0.936	Not specified
500M	40K	System hang	System hang	Not specified

Table 5: Performance speed up of OpenMP compare to sequential dot product program

7.1.1.2.OpenMP – Odd_Even Transposition Analysis.

In this program, the unsorted data is divided evenly to processors. This program was written for two processors and each processor owns half a chunk of data. Obviously, the execution time for the sorting increases with the size of data (see Table 6 and Figure 15).

The maximum data size of this experiment is one hundred twenty thousand items. The size was arbitrary selected due to the limitation of the test time.

In order to measure the performance speed up ratios of OpenMP over a sequential algorithm, the run time of odd-even transposition in OpenMP was compared to that of the sequential bubble sort program. The results of experiments are shown in Table 7. The parallel algorithm in OpenMP has significantly improved the performance of the sort operation, especially if the unsorted data has a large size; i.e.: eight thousand items.

Data size	Execution time, s
80	0.000
4,000	0.031
8,000	0.093
16,000	0.390
40,000	2.434
60,000	5.491
80,000	9.672
100,000	15.032
120,000	21.568

Table 6: Odd-Even Transposition in OpenMP – Execution Time versus Data Size

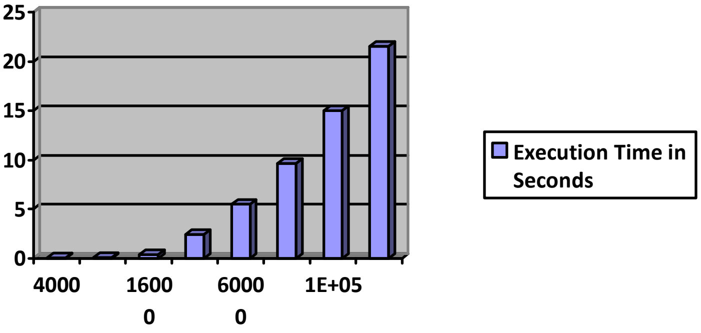


Figure 15: Odd-Even Transposition in OpenMP – Execution Time versus Data Size

Data size	Execution time, s Sequential bubble sort	Execution time, s Odd-even transposition	Speed up ratio, %
80	0.000	0.000	0.00
4K	0.047	0.031	34.04
8K	0.156	0.093	40.38
16K	0.592	0.390	24.12
40K	3.759	2.434	35.25
60K	8.409	5.491	34.70
80K	14.883	9.672	35.01
100K	23.464	15.032	35.94
120K	33.841	21.568	36.27

Table 7: Performance speed up of Odd-even Transposition in OpenMP over sequential bubble sort

7.1.1.3.OpenMP – Graphics Rendering Analysis.

In general, graphics programs use OpenGL to render images. The idea of using multi core and parallel computing is to gain more power computing for the pixels positions, lights, colors and shades so that the graphics can be close to reality. In particular, the animation graphics needs more computing power to constantly fill in the pixels' attributes of the next graphic frame. The quicker the information of the next frame is filled, the quicker the next frame can be displayed.

Unfortunately, OpenMP does not have an easy way to support multiple threads that can concurrently fill into the GL buffer. A simulation algorithm was used to test the performance of multi processes on displayed graphics. The detail of the simulation algorithm is described in section 5.1.4.

The simulation of the graphics rendering is the printing of a string line. The performance measurement is the time necessary to print a number of lines to screen. We hope this simulation proves that each process will print half the amount of lines that need to be printed. Therefore, the amount of time to print will be cut in half. However, the flaw of this simulation is that each thread cannot simultaneously access the standard input/output (I/O) to print strings, while the OpenCL/OpenGL allows multi cores to simultaneously fill in the vertex buffer object (VBO) for GL.

The results of the experiment are recorded in Table 8 and Figure 16. In fact, the performance of OpenMP program is worse than the sequential program. Perhaps the mutual exclusion in the low level of I/O adds more delay in OpenMP program.

Number of lines printed	Execution time, s (sequential program)	Execution time, s (OpenMP program)
10,000	2.091	2.55
80,000	15.366	20.592
100,000	19.578	24.944
120,000	23.509	29.874

Table 8: Graphics Rendering Simulation in OpenMP – Execution Time versus Data Size

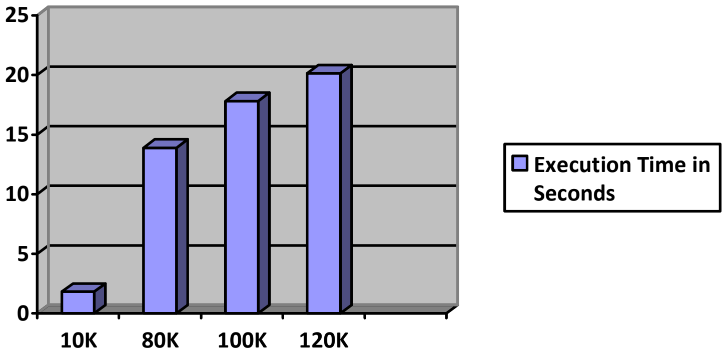
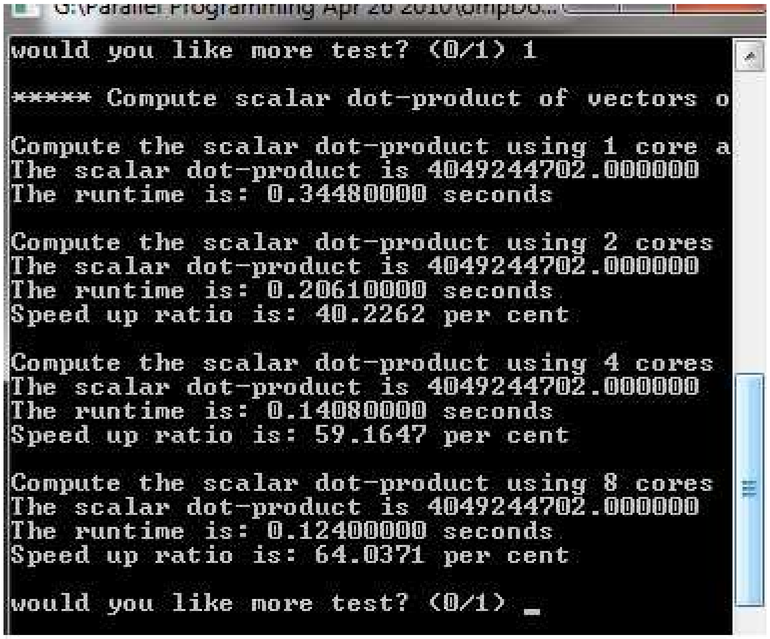


Figure 16: Graphics Rendering Simulation in OpenMP – Execution Time versus Data Size

7.1.1.4. OpenMP – Eight core test results

The test algorithms, dot-product, parallel sort, are used to test an eight core Windows 7 machine. The applications can directly run on Windows 7 with full support of OpenMP's API.

The tests show the significant benefits of OpenMP in arithmetic computation when the number of cores are increased (see Figure 17 and Figure 18). Observation from Figure 18, the performance increase from one-core to two-core is greater than that from two-core to four-core and the performance increase of the two-core to four-core is greater than that from four-core to eight-core. This result agrees that OpenMP or shared memory system in general is not scalable.



```
would you like more test? (0/1) 1

***** Compute scalar dot-product of vectors o

Compute the scalar dot-product using 1 core a
The scalar dot-product is 4049244702.000000
The runtime is: 0.34480000 seconds

Compute the scalar dot-product using 2 cores
The scalar dot-product is 4049244702.000000
The runtime is: 0.20610000 seconds
Speed up ratio is: 40.2262 per cent

Compute the scalar dot-product using 4 cores
The scalar dot-product is 4049244702.000000
The runtime is: 0.14080000 seconds
Speed up ratio is: 59.1647 per cent

Compute the scalar dot-product using 8 cores
The scalar dot-product is 4049244702.000000
The runtime is: 0.12400000 seconds
Speed up ratio is: 64.0371 per cent

would you like more test? (0/1) _
```

Figure 17: Output of dot-product program on the eight-core Windows 7 machine

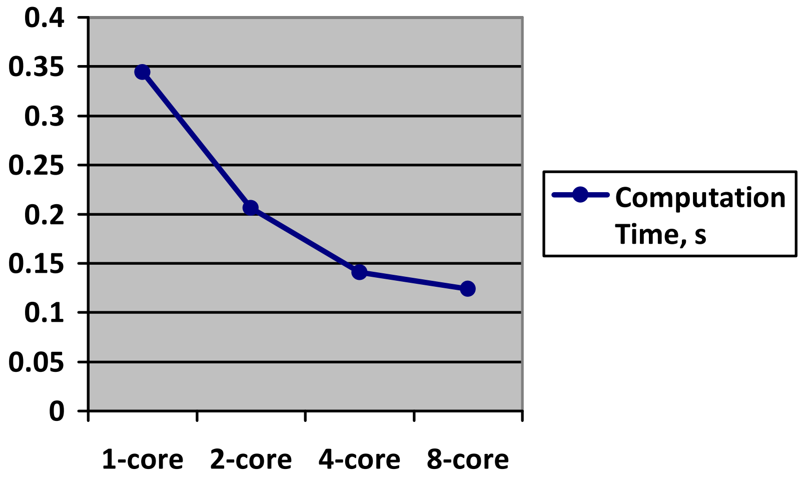


Figure 18: Performance increase with the number of cores in OpenMP-Dot-Product program

7.1.2. MPI:

MPI programs can be optimized by programmers to gain efficiency of CPU usage. The guidelines for effective MPI programming are to minimize the number of messages as well as the size of messages across processes, evenly divide tasks among processes and maximize the computational tasks per data set.

7.1.2.1.MPI – Dot Product Analysis.

Unlike the OpenMP program, the MPI program splits the vectors into two chunks each. Processor *P0* plays a master role and processor *P1* plays a slave role. *P0* sends the second half of each vector to *P1* to compute the second half portion of the dot product. *P0* itself computes the first half of the dot product and then waits for the result from *P1* to make the final sum for the scalar value of the dot product of two vectors.

There are two different clocks used to measure the performance of MPI programs. They are C++ clock and MPI clock. The run time measurements across languages are the time each language runs the parallel computation (not including overhead time). In MPI programs, it is impossible to exclude the overhead using the C++ clock. Also, the C++ clock shows different values of time from different processes in a same program, while the MPI clock does not. Therefore, the MPI clock is used to measure the performance instead of the C++ clock. Table 9 contains the execution time of a program with different clocks from different processes.

Overall, the performance of MPI in the dot product program is much slower than that of the sequential algorithm. MPI-send and –receive functions used in this program contain a synchronization mechanism, such that the program will wait for the complete send and receive of data before proceeding to the next computations. The delay caused by the synchronization between send and receive increases significantly when the vector size increases. Table 10 illustrates this point.

The arithmetic calculation, such as dot product program, does not seem to be beneficial in MPI, especially when the vector size is large. MPI is not recommended for this type of computation.

Data size	Thread0 run time, s (C++ clock)	Thread1 run time, s (C++ clock)	Thread0 run time, s (MPI clock)	Thread1 run time, s (MPI clock)
5,000,000	0.156	0.156	0.14	0.14
8,000,000	0.234	0.234	0.27	0.27
10,000,000	0.432	0.452	0.40	0.40
20,000,000	0.639	0.655	0.59	0.59

Table 9: Dot Product of Two Vectors in MPI – Execution Time versus Vectors' Size

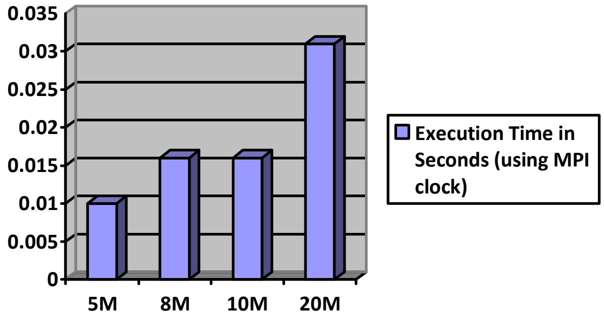


Figure 19: Dot Product of Two Vectors in MPI – Execution Time versus Vectors size

Data size	Execution time, s (MPI clock)	Send time, s (from *P0*)	Receive time, s (from *P1*)
5,000,000	0.14	0.07	0.13
8,000,000	0.27	0.14	0.27
10,000,000	0.40	0.22	0.39
20,000,000	0.59	0.32	0.57

Table 10: MPI Dot Product – Time Analysis

7.1.2.2.MPI – Odd-Even Transposition Analysis.

Odd-Even Transposition program in MPI increases the performance significantly compared to sequential bubble sort, and even OpenMP program. The delay time due to communication APIs seems insignificant because the size of the data sent is relatively small. Table 11 and Figure 20 record the execution time and the data size values are proportional. Figure 21 shows the samples of commands to run MPI program and the outputs.

Table 12 computes the performance speed up ratios of parallel sort in MPI compared to that of the sequential bubble sort and the parallel sort in OpenMP. The results from Table 12 shows that MPI provides good performance for those algorithms that have a small set of data transferred, and the computation tasks for each process are rather complex.

MPI is a good candidate for the complex parallel programs with relatively small chunks of data, such as Odd-even transposition algorithm in data range of hundreds of thousands of items.

Data size	Thread0 run time, s (C++ clock)	Thread1 run time, s (C++ clock)	Thread0 run time, s (MPI clock)	Thread1 run time, s (MPI clock)
4,000	0.046	0.062	0.02	0.02
8,000	0.093	0.109	0.07	0.07
16,000	0.297	0.312	0.27	0.27
80,000	7.010	7.016	6.89	6.98
100,000	11.029	11.029	11.01	11.01
120,000	16.021	16.021	16.00	16.00

Table 11: Odd-Even Transposition in MPI – Execution Time versus Data Size

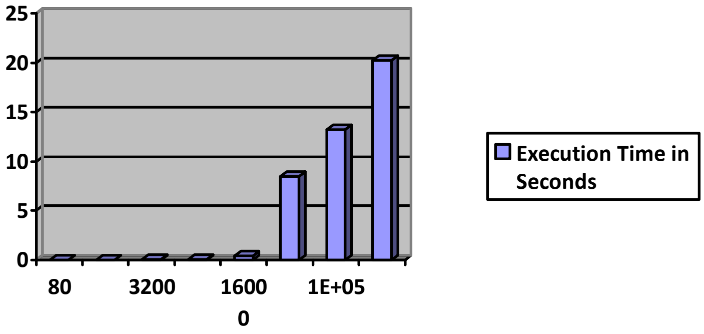
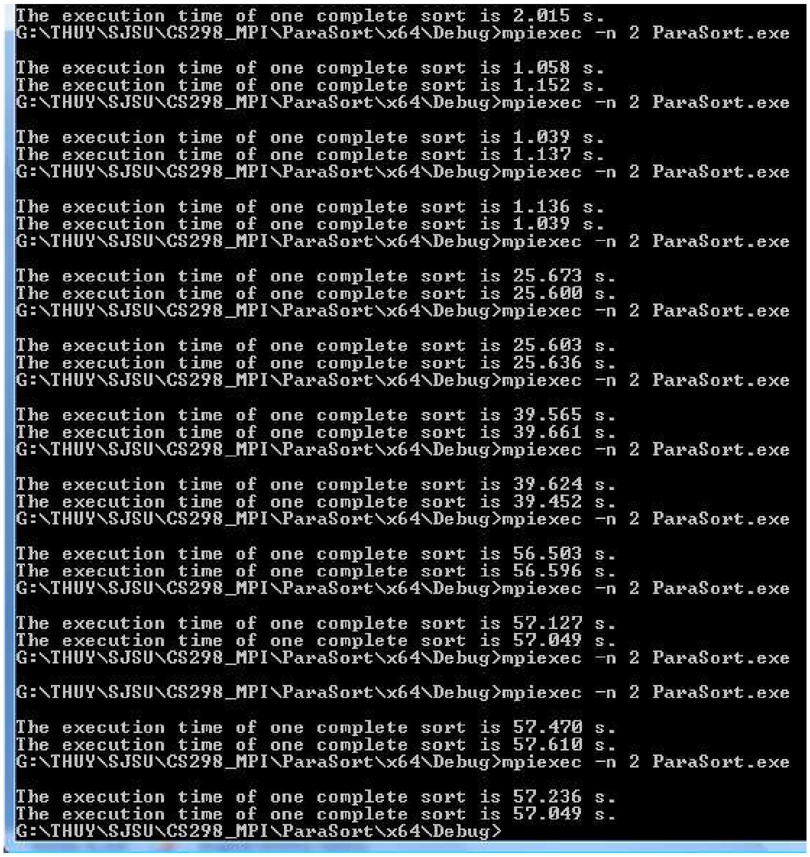


Figure 20: Odd-Even Transposition in MPI – Execution Time versus Data Size



```
The execution time of one complete sort is 2.015 s.
G:\THUY\SJSU\CS298_MPI\ParaSort\x64\Debug>mpiexec -n 2 ParaSort.exe

The execution time of one complete sort is 1.058 s.
The execution time of one complete sort is 1.152 s.
G:\THUY\SJSU\CS298_MPI\ParaSort\x64\Debug>mpiexec -n 2 ParaSort.exe

The execution time of one complete sort is 1.039 s.
The execution time of one complete sort is 1.137 s.
G:\THUY\SJSU\CS298_MPI\ParaSort\x64\Debug>mpiexec -n 2 ParaSort.exe

The execution time of one complete sort is 1.136 s.
The execution time of one complete sort is 1.039 s.
G:\THUY\SJSU\CS298_MPI\ParaSort\x64\Debug>mpiexec -n 2 ParaSort.exe

The execution time of one complete sort is 25.673 s.
The execution time of one complete sort is 25.600 s.
G:\THUY\SJSU\CS298_MPI\ParaSort\x64\Debug>mpiexec -n 2 ParaSort.exe

The execution time of one complete sort is 25.603 s.
The execution time of one complete sort is 25.636 s.
G:\THUY\SJSU\CS298_MPI\ParaSort\x64\Debug>mpiexec -n 2 ParaSort.exe

The execution time of one complete sort is 39.565 s.
The execution time of one complete sort is 39.661 s.
G:\THUY\SJSU\CS298_MPI\ParaSort\x64\Debug>mpiexec -n 2 ParaSort.exe

The execution time of one complete sort is 39.624 s.
The execution time of one complete sort is 39.452 s.
G:\THUY\SJSU\CS298_MPI\ParaSort\x64\Debug>mpiexec -n 2 ParaSort.exe

The execution time of one complete sort is 56.503 s.
The execution time of one complete sort is 56.596 s.
G:\THUY\SJSU\CS298_MPI\ParaSort\x64\Debug>mpiexec -n 2 ParaSort.exe

The execution time of one complete sort is 57.127 s.
The execution time of one complete sort is 57.049 s.
G:\THUY\SJSU\CS298_MPI\ParaSort\x64\Debug>mpiexec -n 2 ParaSort.exe

G:\THUY\SJSU\CS298_MPI\ParaSort\x64\Debug>mpiexec -n 2 ParaSort.exe

The execution time of one complete sort is 57.470 s.
The execution time of one complete sort is 57.610 s.
G:\THUY\SJSU\CS298_MPI\ParaSort\x64\Debug>mpiexec -n 2 ParaSort.exe

The execution time of one complete sort is 57.236 s.
The execution time of one complete sort is 57.049 s.
G:\THUY\SJSU\CS298_MPI\ParaSort\x64\Debug>
```

Figure 21: Data output of MPI running Parallel Sort program

Data size	Run time, s (sequential)	Run time, s (OpenMP)	Run time, s (MPI)	Speed up, % (MPI/sequ.)	Speed up, % (MPI/OMP)
4K	0.046	0.046	0.02	56.52	56.52
8K	0.234	0.125	0.07	70.09	44.00

16K	0.686	0.359	0.27	60.64	24.79
80K	15.101	8.373	6.98	53.78	16.63
100K	23.318	13.090	11.01	52.78	15.89
120K	33.612	20.121	16.00	52.40	20.48

Table 12: Speed up ratios of Odd-Even Transposition in MPI

7.1.2.3.MPI – Graphics Rendering Analysis.

Like OpenMP, a graphics simulation program is applied for MPI. The detail algorithm is described in section 5.1.4. A printing string program is used to simulate a graphics rendering. The simulation algorithm has the same flaw as described in section 7.1.1.3.

Table 13 and Figure 22 record the results of experiment of graphics simulation. The performance of the graphics simulation in MPI is much better than that of OpenMP. Observing the output screens, we see that the MPI program tends to allow a process occupying the standard I/O longer before other process can get access to I/O while in OpenMP, processes take turns to access the I/O more frequently.

Number of printed lines	Runtime, s (sequential program)	Thread0 runtime, s (C++ clck)	Thread1 runtime, s (C++ clck)	Thread0 runtime, s (MPI clck)	Thread1 runtime, s (MPI clck)
10K	2.091	1.420	1.435	0.03	0.04
80K	15.366	11.232	11.232	0.21	0.28
100K	19.578	14.258	14.258	0.30	0.33
120K	23.509	17.050	17.050	0.35	0.39

Table 13: Graphics Rendering Simulation in MPI – Execution Time versus Data Size

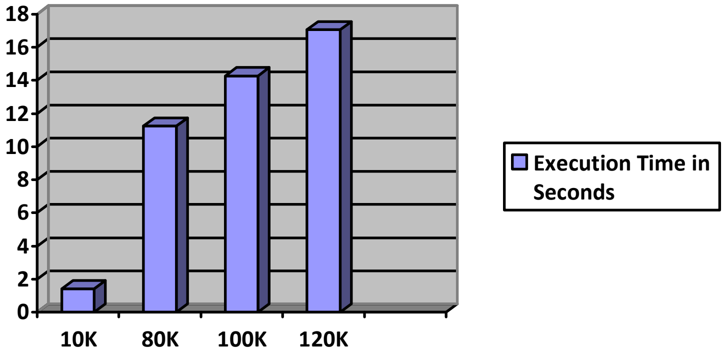


Figure 22: Graphics Rendering Simulation in MPI – Execution Time versus Data Size

7.1.3. OpenCL:

Although OpenCL has advantages of multi core power in computations, it has limitations in the size of processing data. In other words, to optimize the performance of the OpenCL program, the data transferred between host and device should be minimized, and the amount of tasks for each kernel should be maximized.

Throughout the experiments, OpenCL programs need extra time to build the kernel program. Even though this time causes long delays when running experiment programs, it is excluded from the execution time. This is because in the reality of software production, this delay can be eliminated by using a binary kernel program.

On the other hand, OpenCL is not capable of transferring a large chunk of data from host to device. This limitation depends on the version of the GPU chip. The GPU chip in these experiments provides the communication channel between host and device with page-able memory of about thirty three million bytes and the bandwidth of about one thousand kilobytes per second. This constraint has limited the data size of the dot-product computation and the parallel sort operation significantly compared to other languages.

7.1.3.1.OpenCL – Dot-Product Analysis.

In OpenCL experiments, the vector size of the dot-product computation cannot exceed ten million dimensions. This is the limitation of the maximum size of the data transferred between host and device.

Table 14 provides the time to build the kernel program; the total run time of the program (excluding the kernel program build time); and the parallel kernels run time versus the data size. Total run time includes the execution time of the kernels, results from the kernels being read back to host and the time for host to sum up the result of the scalar value of the dot-product.

The total run time of the OpenCL program is greater than that of the sequential program. One reason for this delay is because the program uses the device's global memory. The tradeoff of using global memory is that the memory space is large but slow. However, a strategy for reducing global memory traffic is not in the scope of this project.

Data size	Kernel program built time, s	Total run time, s	Parallel kernels run time, s
5,000,000	0.328	0.873	0.046
8,000,000	0.328	1.388	0.078
10,000,000	0.312	1.653	0.093

Table 14: Dot Product of Two Vectors in OpenCL – Execution Time versus Vectors Size

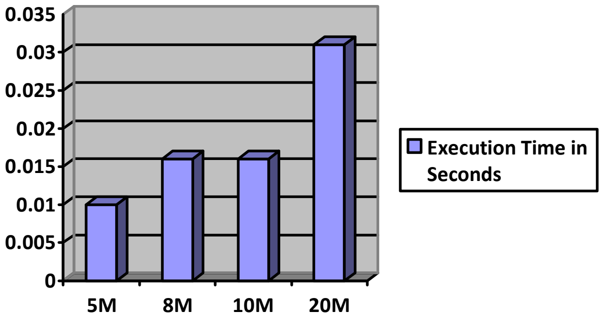


Figure 23: Dot Product of Two Vectors in OpenCL – Execution Time versus Vectors Size

7.1.3.2.OpenCL – Odd-Even Transposition Analysis.

In order to take advantage of eight cores in a GPU, the experiment uses the maximum of eight cores. Each core has a maximum of five hundred slots of integers and therefore the maximum number of data to be sorted is four thousand integers. Table 15 collects the build time of the kernel program, the total run time (including the kernel runtime and copying the sorted data from device to host), and the kernel runtime versus data size.

In general, the kernel program build time does not change when the data size is changed. This time is necessary for the compiler to build the kernel binary program. It does not depend on the size of the data, but on the CPU resource.

The total runtime in this experiment does not show any better performance than sequential bubble sort. However, the kernel runtime shows significantly improved performance. If OpenCL did not have the bottleneck performance in copying data between host and device, then the kernel runtime of OpenCL would be the best performance in parallel sort algorithm.

Data size	Kernel program build time, s	Total runtime, s	Parallel kernels runtime, s
512	0.374	0.031	0.000
800	0.375	0.093	0.000
2400	0.375	0.686	0.008
3200	0.390	1.201	0.008
4000	0.374	1.888	0.009

Table 15: Odd-Even Transposition in OpenCL – Execution Time versus Data Size

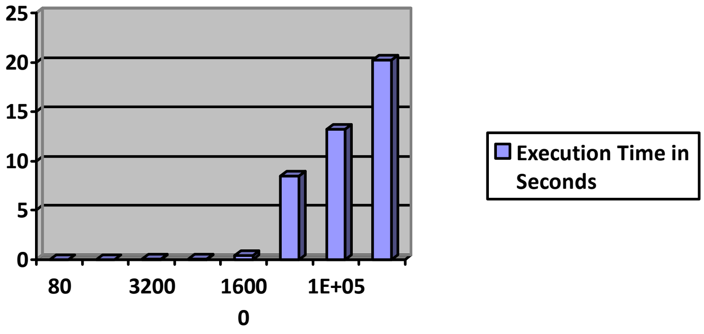


Figure 24: Odd-Even Transposition in OpenCL – Execution Time versus Data Size

7.1.3.3.OpenCL - Graphics Rendering Analysis.

The graphics rendering program is obviously favorable to OpenCL because OpenCL supports the OpenGL API and the specifications of both languages are controlled by the same Khronos group.

An implementation of a graphical output to illustrate the performance benefit of OpenCL is difficult and not in the scope of this project. In this experiment, the OpenCL program uses a pre-processing graphics API provided by Nvidia's library. The kernel program of OpenCL computes the post-processing of the graphics. The results computed by kernels are simultaneously filled into the vertex buffer object (VBO) for OpenGL. As soon as the GL buffer is filled, OpenGL swaps buffers to produce the next frame of the graphics. Therefore, the duration of a complete graphics animation cycle will represent the performance of the program. Figure 25 shows frame samples of the experiment output.

Also, this test program has a built-in switch that can switch to sequential computing post-processing graphics. Thus, the performance of OpenCL and non OpenCL can be measured and compared.

GLUT pre-built models sub-API	Duration of complete animation cycle, s (sequential/non OpenCL)	Duration of complete animation cycle, s (OpenCL)
glutSolidTorus	60	30
glutSolidTeapot	65	30
Stars	5	

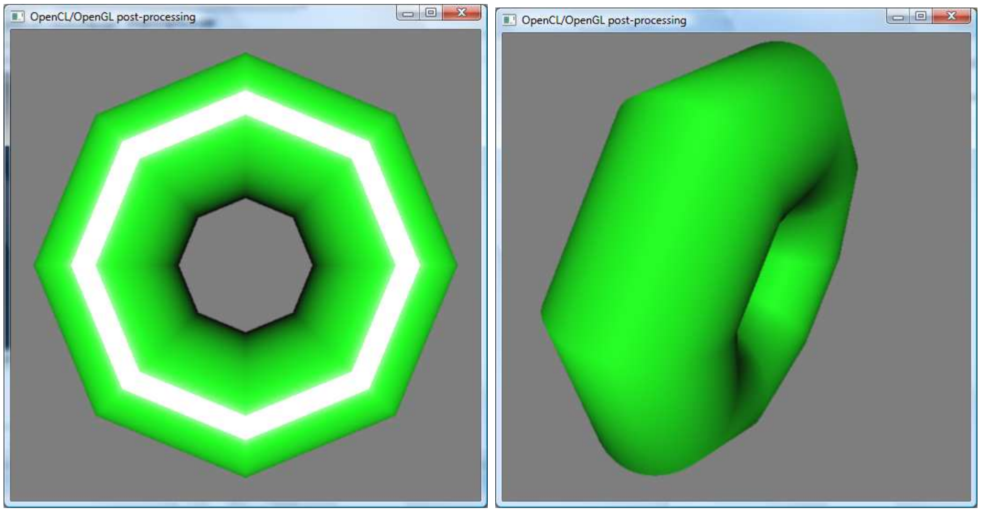


Figure 25: glutSolidTorus animation demonstration

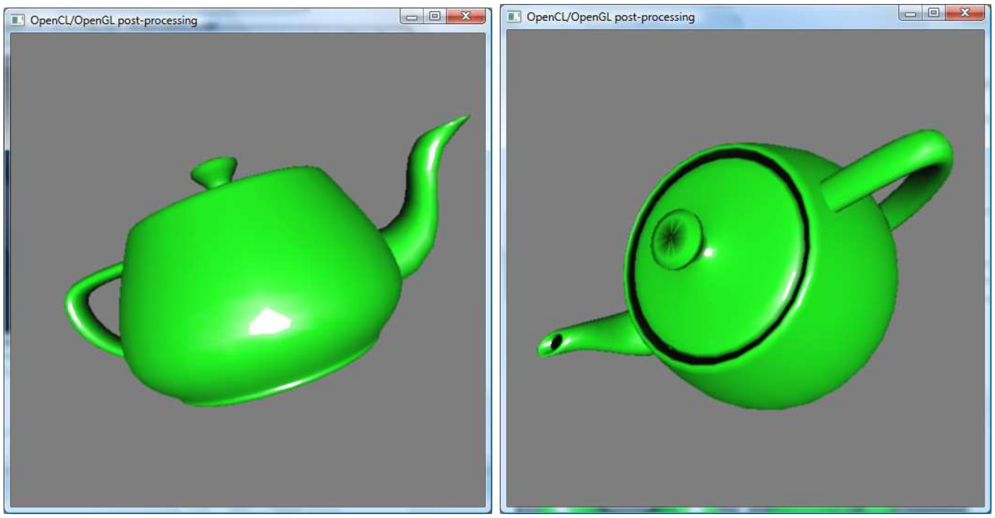


Figure 26: glutSolidTeapot animation demontration

7.2. Performance Comparison across Languages:

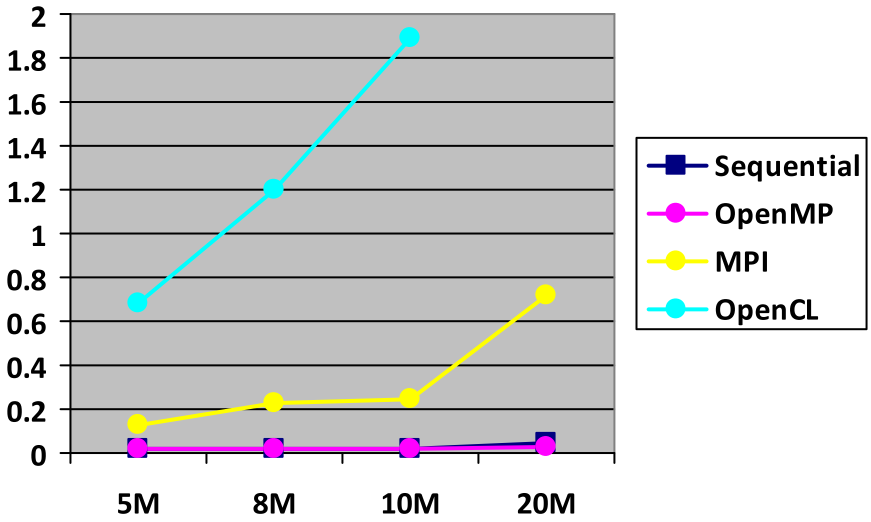


Table 16 is the summary of performance comparisons across languages. In arithmetic computations, the OpenMP program seems to be the best candidate for performance (see Figure 27). On the other hand, the MPI program seems to have the best performance for sorting (see Figure 28). Although OpenCL is a most complex language, its superior performance on graphics makes it the best candidate for graphics applications.

Algorithms	Test Scenario	Single Core	OpenMP, Dual core	MPI	OpenCL
Dot Product of vectors size 5M	Algorithm Run Time, s	0.015	0.015	0.13	0.046
Vectors size 8M		0.016	0.015	0.23	0.078
Vectors size 10M	Algorithm Run Time, s	0.016	0.016	0.25	0.093
Vectors size 20M		0.047	0.031	0.72	out of rsc
Bubble Sort 2400 items		0.035	0.020	0.01	0.686

.Bubble Sort 3200 items		0.061	0.043	0.01	1.201
Bubble Sort 4K items		0.095	0.055	0.02	1.888
Bubble Sort 8K items		0.234	0.125	0.08	out of rsc
Bubble Sort 16K items	Algorithm Run Time, s	0.686	0.359	0.26	
Bubble Sort 80K items	Algorithm Run Time, s	15.101	8.373	6.87	
Bubble Sort 100K items	Algorithm Run Time, s	23.318	13.090	10.86	
Bubble Sort 120K items	Algorithm Run Time, s	33.612	20.121	15.71	
Pre-processing animation graphics, from Nvidia	60				30

Table 16: Performance Comparison

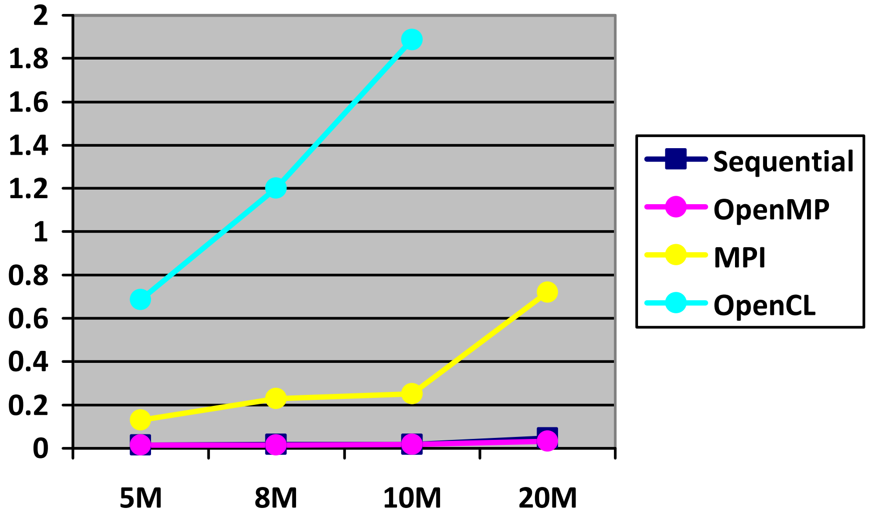


Figure 27: Execution Time in Seconds of Languages in Dot Product Algorithm

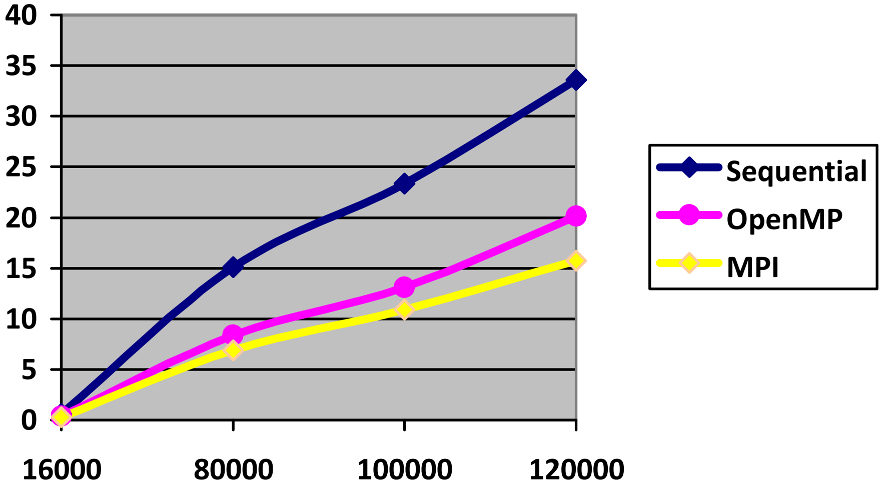


Figure 28: Performance (in Seconds) of Languages in Bubble Sort Algorithm (Larger Data Set)

7.3. Implementation Cost:

In this section, the specifications of the languages, the languages' library setting, and the time necessary for programmers to learn and create programs will be carefully reviewed and compared.

The compilation setting for a language is used to measure the complexity of the language. This factor reflects the willingness to use the language and the complicated environment of the language. In this aspect, OpenMP has the most advantages because it is embedded in Microsoft Visual Studio 2008, while other languages have to be downloaded and installed. Users have to carefully select the correct version of software to their hardware system. Download time is another complex factor that could cause the failure of the environment setting. See 5.2.1 for more details of the download and installation processes. The comparison values across languages are shown on the first row of Table 17.

Additionally, the number of steps in Visual Studio to setup a project with the supported language is measured, and compared. See 5.2.1 for the measurement values, and see the comparison values across languages on the second row of Table 17.

The parallel structure of a language is also used to determine the complexity of a language. See detail measurements in section 5.2.3 and the comparison values across languages on the third row of Table 17.

The fourth through the eighth rows of Table 17 are values collected from the languages' specifications. Detailed calculations can be found in section 5.2.2

Finally, the implementation hours of the projects were recorded on the last three rows of Table 17.

Data collected in Table 17 seem to be in favor of OpenMP. Except the graphics program, this language has the least complexity and is the easiest to use.

		Sequential	OpenMP	MPI	OpenCL

Compiler/ environment setup time		0	1 minute using Visual Studio 2008	1	3
Number of steps to setup a project		0	2	6	11
Parallel structure complexity		0	1	2	3
API/ Library complexity	Number of directives	0	15	0	0
	Number of clauses	0	8	0	0
	Number of constants and handles	0	0	483	227
	Number of Object declarations	0	0	33	0
	Number of routines	0	31	355	521
Programming time, in hours	Dot-product algorithm	1	2	8	16
	Odd-even transposition	8	2	8	16
	Graphics Redering	2	2 (simulation)	2 (simulation)	80

Table 17: Implementation Cost Comparison

8. Future Work

The goal of this project is to provide programmers patterns of solving problems in parallel programming. This project analyzed only three categories of parallel algorithms. In software world there are other well-known problems such as finite state machine, circuits, graph-algorithms, etc. The more problems are analyzed and documented the better information for programmers to design the real world solutions.

In addition, this project provides solutions using three different parallel programming languages on Windows platform. In real world, programmers have much wider range of languages and platforms to work on. The future work will also be extended to these areas.

9. Conclusions

There are several categories that need to be carefully examined when designing a parallel system. Depending on the hardware resources, the nature of the business, the cost effectiveness, and engineering resources, the chosen parallel language must be optimal for all factors. The following are a list of business models and the recommended languages.

1. Quick and small business with no future extension: With this business model, OpenMP should be considered for the parallel programming. OpenMP will be quick to implement and provide good performance in shared memory system.

Parallel algorithms like arithmetic computations are the best match for this language.

2. Large systems in long term business and future expansion: This business type has the expansion factor and a large size of parallelism. MPI should be the best for this type of system. In particular, the parallel sort-like algorithms, which require nodes to perform a large amount of tasks and very little communications across processes, are the best candidate for this system.

3. Animation graphics, scientific modeling graphics: These types of applications will need OpenCL to take advantage of OpenGL supports and the powerful multi core of the GPU chips.

10. References

[1] Faculty of Computational & Cybernetics - University of Nizhni Novgorod (2006), Introduction to Parallel Programming, URL http://www.software.unn.ru/ccam/mskurs/ENG/HTML/cs338_pp_materials.htm

[2] Berkeley University of California (2009), 2009 Par Lap Boot Camp – Short Course on Parallel Programming, URL http://parlab.eecs.berkeley.edu/bootcampagenda

[3] Blaise Barney, Lawrence Livermore National Laboratory (2009) POSIX Thread Programming, UCRL-MI-133316, URL https://computing.llnl.gov/tutorials/pthreads/

[4] Kayvon Fatahalian & Mike Houston, A Closer Look at GPUs, URL http://graphics.stanford.edu/~kayvonf/papers/fatahalianCACM.pdf

[5] Vasily Volkov & James Demmel, Using GPUs to Accelerate Linear Algebra Routines,

URL http://www.cs.berkeley.edu/~volkov/volkov08-parlab.pdf

[6] Wikipedia,
Shared Memory,
URL http://en.wikipedia.org/wiki/Shared_memory

[8] Timothy Mattson, Beverly Sanders, Berna Massingill, (2004)
Patterns for Parallel Programming,
Addison-Wesley Professional

[9] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams and Katherine A. Yelick
The Landscape of Parallel Computing Research: A View from Berkeley,
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf

[10] Timothy. G. Mattson, Beverly A. Sanders, and Berna L. Massingill,
Patterns for Parallel Programming,
Addison-Wesley, 2008

[11] Hagit Attiya, Jennifer Welch,
Distributed Computing, second edition,
Wiley-Interscience, 2004

[12] Kevin A. Huck, Oscar Hernandez, Van Bui, Sunita Chandrasekaran,
Barbara Chapman, Allen D. Malony, Lois Curfman McInnes, and Boyana Norris,
Capturing Performance Knowledge for Automated Analysis,
Austin, Texas, CS2008 November, 2008

[13] Jorge Luis Ortega Arjona,
Architectural Patterns for Parallel Programming – Models for Performance Estimation,
Department of Computer Science, University College London, November 2006

[14] Nathan R. Tallent, John M. Mellor-Crummey,
Effective Performance Measurement and analysis of Multithreaded Applications,
Rice University, PPOPP's 09, February 14-18, 2009

[15] David W. Gohara,
Introduction to OpenCL
Center of Computational Biology, Washington University, 2009

[16] David B. Kirk, Wen-mei W. Hwu,
Programming Massively Parallel Processors,
Morgan Kaufmann, 2010

[17] Ryan Eccles, Deborah A. Stacey,
Understanding the Parallel Programmer,
University of Guelph, HPCS' 06, 0-7695-2582-2/06, IEEE Xplore, 2006

[18] Mathew J. Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, David I. August,
Revisiting the Sequential Programming Model for Multi Core,
Computer Science of Princeton University, 1072-4451/07, IEEE Xplore, 2007

[19] Lorin Hochstein, Jeff Carver, Forrest Shull, Sima Asgari, Victor Basili, Jeffrey K. Hollingsworth, Marvin V. Zelkowitz,
Parallel Programmer Productivity: A Case Study of Novice Parallel Programmers,
1-59593-061-2/05, IEEE Xplore, 2005

[20] Anwar Ghuloum, Eric Sprangle, Jesse Fang, Gansha Wu, Xin Zhou,
Ct: A Flexible Parallel Programming Model for Tera-scale architecture,
Intel Leap Ahead, 2007

[21] Hsin-Chu Chen, Alvin Lim, Nazir A. Warsi,
Multilevel master-slave parallel programming models,
Clark Atlanta University, DAAL-O3-G-92-0377, 2006

[22] Luis Moura E Silvay, Rajkumar Buyyaz,
Parallel Programming Models and Paradigms,
Monash Univerity, Melbourne, Australia 2000

[23] Michael J, Quynn,
Parallel Programming in C with MPI and OpenMP,
0-07-282256-2, McGraw-Hill, New York 2004

[24] Barbara Chapman, Gabriele Jost, Juud Van De Pas,
Using OpenMP: Portable Shared Memory Parallel Programming,
978-0-262-53302-7, Massachusetts Institute of Technology, 2008

[25] Clay Breshears,
The Art of Concurrency: A Thread Monkey’s Guide to Writing Parallel Applications,
978-0-596-52153-0, O’Reilly Media Inc, California 2009

[26] Ajit Singh, Jonathan Schaeffer, Duane Szafron,
Views on Template-Based Parallel Programming,
IBM Centre for Advanced Studies Conference, Toronto, Ontario, Canada, 1996

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides,
Design Patterns: Abstraction and Reuse of Object-Oriented Designs,
Addison-Wesley, 1995

[28] G. Andrews, R.A. Olsson, M.A. Coffin, I. Elshoff, K. Nilson, T. Purdin, and G. Townsend,
An Overview of the SR Language and Implementation,

AMC Trans. on Prog. Languages and Systems, 10(1): 52-86, 1988

[29] Kai Tan, Duane Szafron, Jonathan Schaeffer, John Anvik, Steve MacDonald,
Using Generative Design Patterns to Generate Parallel Code for a Distributed Memory Environment,
University of Alberta, Edmonton, 2003, ISBN: 1-58113-588-2

[30] Arun Kejariwal, Alexander V. Veidenbaum, Alexandru Nicolau, Milind Girkar, Xinmin Tian, Hideki Saito,
On the Exploitation of loop-level parallelism in embedded applications,
ACM, New York, NY, USA, 2009, ISSN: 1539-9087

[31] Geraud Krawezik, Franck Cappello,
Performance Comparison of MPI and three Programming Styles on Shared Memory Multiprocessors,
ACM Symposium on Parallel Algorithms and Architectures, San Diego, California, USA, 2003, ISBN: 1-58113-661-7

[32] B. Chapman, A. Patil, and A. Prabhakar,
Performance Oriented Programming for NUMA Architectures,
In Springer-Verlag Berlin Heidelberg, editor, LNCS 2104,
International Workshop on OpenMP Applications and Tools,
WOMPAT 2001, West Lafayette, IN, USA, 2001

[33] H. Jin, M. Frumkin, and J. Yan,
The OpenMP Implementation of the NAS Parallel Benchmarks and its Performance,
In NASA Ames Research Center, editor, Technical Report NAS-99-01, 1999.

Appendix A

Configuring Visual C++ 2008 program to run command arguments, *argv*:
The default setting of *Character Set* in Visual 2008 project's properties is *Use Unicode Character Set*. This property will cause a compilation error due to incompatibility between *_TCHAR* and *char* pointers. This property should be set to *Use Multi-Byte character Set.*

Step-by-step to configure *Character Set* in Visual 2008 projects.

- Open project's Property Pages.
- Go to Configuration Properties/General
- Select Character Set field
- Change its value to Use Multi-Byte Character Set

Appendix B - OpenMP source codes

1/ OpenMP – Maclaurin Series of e^x:

```
// Author: Thuy Nguyenphuc
// Date: 9/18/2009

/* MaclaurinSeries.cpp : Defines the entry point for the console
   application.
   Taylor series:
   e^x = 1 + (x+a) + (x+a)^2/2! + (x+a)^3/3! + ... +(x+a)^n/n!
   Maclaurin series(the evaluate point a = 0):
   e^x = 1 + x + x^2/2! + x^3/3! + ... + x^n/n! */

#include "stdafx.h"
#include "windows.h"
#include "iostream"
#include "stdafx.h"
#include "tchar.h"

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

```
/* constant defined*/
#define E 2.71828
#define PROCESSORS 2    //number of processors
#define CHUNKSIZE 500
#define N         CHUNKSIZE*PROCESSORS

double walltime( double* ); /* the clock on the wall */

int _tmain(int argc, _TCHAR* argv[])
{
      int i,j,chunk, myloop = 0;
      int tid;
      int myRunTime = atoi(argv[1]);
      float x = 0.5;
      double sum = 1;
      double nfactorial_value;
      long nfactorial_base;

      // time variables to measure the draw operation's execution time
      clock_t start, finish;
      double dur, total_rt = 0.0; // execution time of program

      while (myloop < myRunTime)
      {
      /* Now begins the real work which we want to parallelize. */
      /* Mark the starting time of parallel execution.
       **********************************************************/
            start = clock();
            sum = 1;
            chunk = CHUNKSIZE;

      /* Here's the OpenMP pragma that parallelizes the for-loop. */
      /* This parallel construct has 4 private variables:         */
      /*   1) outer loop index "i" by default                     */
      /*   2) inner loop index "j" by explicit declaration        */
      /*   3) "nfactorial_value" by explicit declaration          */
      /*   4) "nfactorial_base" by explicit declaration           */
      /* Schedule claude specify 2 chunks of 500 terms assigned   */
      /* statically to 2 processors.                              */
      /* Reduction claude synchronized the results of processors  */
      /* and sum them up to the final result.                     */

      #pragma omp parallel for \
      default(shared) private(i,j,nfactorial_value,nfactorial_base) \
      schedule(static,chunk) \
      reduction(+:sum)
          for (i=1; i < N; i++)
          {
              nfactorial_value = 1;
              nfactorial_base = 0;
              for(j = 1; j <= i; j++)
              {
                  nfactorial_value *= j;
                  if(nfactorial_value > 1000)
                  {
                      nfactorial_value /= 1000;
                      nfactorial_base += 3;
                  }
              }
              sum = sum + pow(x,i) / (nfactorial_value *
                                 (float)pow(10.0,nfactorial_base));
          }/* end of parallel */

          /* Work's done. Get the elapsed wall time. */
          finish = clock();
          total_rt += (double)(finish - start) / CLOCKS_PER_SEC;
          myloop++;
      }
      dur = total_rt / myRunTime;
      /* Print the sequential execution time.
       ***************************************************************/
    fprintf( stdout, "\n\nParallel execution time of one calculation is
            %f s.", dur);
    fprintf( stdout, "\nParallel execution time of %d calculations is
            %f s.\n", myRunTime, total_rt);

return 0;
}
```

2/ OpenMP – Dot Product:

```
// Author: Thuy Nguyenphuc
// Date: 10/05/2009

// DotProduct.cpp : Parallel calculation at each dimension of vectors.
//

#include "stdafx.h"

// for clock_t type
#include <time.h>
#include <stdlib.h>

// for OpenMP
#include <omp.h>

/* constant defined*/
#define PROCESSORS 2    //number of processors
#define CHUNKSIZE 1000
#define MAXITEMS         CHUNKSIZE*PROCESSORS*5000 // Array size
#define BILLION  1000000000

int _tmain(int argc, _TCHAR* argv[])
{
      int i,j,chunk, myloop = 0;
      int tid;
      int myRunTime = 1;//atoi(argv[1]);
      // prepare arrays A and B
      int* A, *B;

      A = new int[MAXITEMS];
      B = new int[MAXITEMS];


      for(i = 0; i < MAXITEMS; i++)
      {
            A[i] = rand()%10;
            B[i] = rand()%10;
      }

      // the result of Dot-Product stored here
      double DotProduct_Val;

      // time variables to measure the draw operation's execution time
      clock_t start, finish;
      double dur, total_rt = 0.0; // for execution time of sequential
program of bubble sort algorithm

      while (myloop < myRunTime)
      {
/* Now begins the real work which we want to parallelize. */
/* Mark the starting time of parallel execution.
*****************************************************/
            start = clock();
            DotProduct_Val = 0.0;

            chunk = CHUNKSIZE;

/* Here's the OpenMP pragma that parallelizes the for-loop. */
/* This parallel construct has 1 private variable:          */
/*   1) outer loop index "i" by default                     */
/* Schedule claude specify chunks of 500 terms assigned     */
/* statically to 2 processors.                              */
/* Reduction claude synchronized the results of processors  */
/* and sum them up to the final result.                     */

            #pragma omp parallel for private(i) schedule(static,chunk)
reduction(+:DotProduct_Val)//, DotProduct_Pow)
            for (i=0; i < MAXITEMS; i++)
            {
                  long temp = A[i] * B[i];
                        DotProduct_Val = DotProduct_Val + (double)temp;
            }
            fprintf(stdout, "\nThe scalar dot product of vector A and B
is: (%f)", DotProduct_Val);//, DotProduct_Pow);

            /* Work's done. Get the elapsed wall time. */
            finish = clock();
            total_rt += (double)(finish - start) / CLOCKS_PER_SEC;
            myloop++;
      }
      dur = total_rt / myRunTime;
      /* Print the sequential execution time.
***************************************************************/
    fprintf( stdout, "\n\nParallel dot product calculation of data set
of %d is %f s.",MAXITEMS, dur);
//    fprintf( stdout, "\nParallel execution time of %d calculations is
%f s.\n", myRunTime, total_rt);

      delete [] A;
      delete [] B;

      return 0;
}
```

3/ OpenMP – Bubble Sort:

```
// Author: Thuy Nguyenphuc
// Date: 9/23/2009

// bubble_sort.cpp : This program perform bubble sort of an array
// of 16000 random integers.
//

#include "stdafx.h"
#include <stdlib.h>
#include <time.h>

#define MAXDATA 16000

void compare_exchange(int &a, int &b)
{
      int tmp;
      if (a > b)
      {
            tmp = a;
            a = b;
            b = tmp;
      }
}

int _tmain(int argc, _TCHAR* argv[])
{
      int myRunTime = atoi(argv[1]);
      int myloop = 0;
      /*execution time measurement*/
      clock_t start, finish;
      double dur, total_rt = 0.0;/*run time of bubble sort algorithm */
      int i,j; // indeces of loops

      int A[MAXDATA]; // holding unsorted data array

      while(myloop < myRunTime)
      {
            for(i = 0; i<MAXDATA; i++)// create unsorted array data
                  A[i] = rand();

            start = clock();
            for(i = 0; i<MAXDATA; i++)
                  for(j = 0; j< MAXDATA - i - 1; j++)
                        compare_exchange(A[j], A[j+1]);

            finish = clock();
            total_rt += (double)(finish - start) / CLOCKS_PER_SEC;

            myloop++;
      }

//    for(i = 0; i<MAXDATA; i++)
//          printf("%7d,", A[i]);

      dur = total_rt / myRunTime;
      printf("\n\nSerial execution time of one complete sort is %1.3f
              s.", dur);
      printf("\nSerial execution time of %d calculations is %f s.\n",
              myRunTime, total_rt);

      return 0;
   }
```

4/ OpenMP – Odd Even Transposition:

```
// Author: Thuy Nguyenphuc
// Date: 11/23/2009

// Odd_Even_Transposition.cpp : Parallel sort a data set of size n
// with the number of processors p, where p << n.  Divided data set
// into p chunks and perform odd-even transposition algorithm.
// Data is sorted after pth iteration.

#include "stdafx.h"
#include <omp.h>
#include <stdlib.h>
#include <time.h>

#define HALF_CHUNK 4000
#define CHUNK_SIZE HALF_CHUNK*2
#define MAX_ITEMS CHUNK_SIZE*2

void compare_exchange(int &a, int &b)
{
      int tmp;
      if (a > b)
      {
            tmp = a;
            a = b;
            b = tmp;
      }
}

int _tmain(int argc, _TCHAR* argv[])
{
    int S[MAX_ITEMS];
    int chunk0[CHUNK_SIZE], chunk1[CHUNK_SIZE];
    int workchunk[CHUNK_SIZE];
    int i, j, k, tid, nthreads;

    /*execution time measurement*/
    clock_t start, finish;
    double dur, total_rt = 0.0; // run time of odd even transposition

    /*take the command argument as the number of loop-back*/
    int myRunTime = atoi(argv[1]);
    int myloop = 0;

    while(myloop < myRunTime)
    {
    /*initialize the unordered list S and divide it into 4 segments:
      min_half0, max_half0, min_half1, max_half1*/
    for (i = 0; i<MAX_ITEMS; i++)
    {
        S[i] = rand();
        if(i<CHUNK_SIZE)
            chunk0[i] = S[i];
        else
            chunk1[i-CHUNK_SIZE] = S[i];
    }
    start = clock();
    /*Assign tasks to sort lists A*/

    #pragma omp parallel shared(nthreads, chunk0, chunk1)
    {
        nthreads = omp_get_num_threads();
        #pragma omp for private(i,j,k,tid,workchunk)
        for(i = 0; i < nthreads; i++)
        {
            tid = omp_get_thread_num();
            for(j = 0; j<CHUNK_SIZE; j++) // sort their assigned data
            {
                for(k = 0; k< CHUNK_SIZE - j - 1; k++)
                {
                    if(tid%2 == 0)
                        compare_exchange(chunk0[k], chunk0[k+1]);
                    if(tid%2 == 1)
                        compare_exchange(chunk1[k], chunk1[k+1]);
                }
            }
//These for loops are to put 2 middle halves together and sort
            for(j = 0; j<HALF_CHUNK; j++)
                workchunk[j] = chunk0[HALF_CHUNK+j];
            for(j = 0; j<HALF_CHUNK; j++)
                workchunk[HALF_CHUNK+j] = chunk1[j];
            for(j = 0; j<CHUNK_SIZE; j++)
            {
                for(k = 0; k< CHUNK_SIZE - j - 1; k++)
                    compare_exchange(workchunk[k], workchunk[k+1]);
            }
//return data from workchunk back to chunk0 and chunk1
            for(j = 0; j<HALF_CHUNK; j++)
                chunk0[HALF_CHUNK+j] = workchunk[j];
            for(j = 0; j<HALF_CHUNK; j++)
                chunk1[j] = workchunk[HALF_CHUNK+j];
        } // end for loop i

        #pragma omp for private(i,j,k,tid,workchunk)
        for(i = 0; i < nthreads; i++)
        {
            for(j = 0; j<CHUNK_SIZE; j++) //sort their assigned data
            {
                for(k = 0; k< CHUNK_SIZE - j - 1; k++)
                {
                    if(tid%2 == 0)
                        compare_exchange(chunk0[k], chunk0[k+1]);
                    if(tid%2 == 1)
                        compare_exchange(chunk1[k], chunk1[k+1]);
                }
            }
        }

    } /*end of parallel sections*/
      finish = clock();
      total_rt += (double)(finish - start) / CLOCKS_PER_SEC;
      myloop++;
      }


      dur = total_rt / myRunTime;
      printf("\n\nSerial execution time of one complete sort is %1.3f
             s.", dur);
      printf("\nSerial execution time of %d calculations is %f s.\n",
             myRunTime, total_rt);

      return 0;
}
```

5/ OpenMP – Odd-Even-transposition (second version)

```
/*
File name: ompOddEven.cpp
Author: Thuy Nguyenphuc
Date: 04/24/2010
This program performs sort on a data set of N items.  It sorts data
from small to large in that order.
*/
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>

// constants
#define MAXCHUNKSIZE 1000
#define MAXPROCESSORS 4
#define MAXITEMS MAXCHUNKSIZE*MAXPROCESSORS
#define MAXITEMSPERLINE 10
```

```
// helper functions
//print data
void print_data(int *A)
{
      int i;
      printf("\nData A of %d items:\n", MAXITEMS);
      for(i=0; i<MAXITEMS; i++)
      {
            if(i%MAXITEMSPERLINE == 0) printf("\n");
            printf("%5d", A[i]);
      }
} // end of print_data

// compare the values and exchange if a is greater than b
void compare_exchange(int &a, int &b)
{
      int tmp;
      if (a > b)
      {
            tmp = a;
            a = b;
            b = tmp;
      }
}

// checks if data is sorted
void isSorted(int *A)
{
      int i;
      bool res;
      for(i=0; i<MAXITEMS-1; i++)
      {
            if(A[i] > A[i+1])
            {
                  res = false;
                  break;
            }
            if(i == MAXITEMS-1)
                  res = true;
      }
      if(res)
            printf("\nCongratulations!  Data A of %d items is sorted.",
MAXITEMS);
      else printf("\nData A of %d items is not sorted.", MAXITEMS);
} // end of isSorted

// checks if the max number of processors is a power of 2
bool ispowerof2(int numthreads)
{
      bool res;
      if(numthreads < 2) res = false;
      else
      {
            while(numthreads >= 2)
            {
                  if((numthreads%2) == 0)
                  {
                        if(numthreads == 2)
                        {
                              res = true;
                              break;
                        }
                        numthreads /= 2;
                  }
                  else
                  {
                        res = false;
                        break;
                  }
            }
      }
      return res;
} // end of ispowerof2

void sort(int *A,int chunksize, int num_procs, double seqdur)
{
      int i, j, k, m, tid, halfchunk = chunksize/2;
      //print_data(A);
      printf("\nThe number of processors used in parallel sort is %d",
num_procs);

      // variables to measure the computation time
      clock_t start, finish;
      double dur;

      // start the timer
      start = clock();

      for(i = 0; i < num_procs; i++)
      {
            #pragma omp parallel private(j,k,m,tid) shared(A, i,
num_procs, chunksize)
            {
                  tid = omp_get_thread_num();
                  #pragma omp for schedule(static,chunksize)
                  for(j = 1; j<MAXITEMS; j++) //processors sort their
assigned data
                  {
                        for(k = 0; k< MAXITEMS - j; k++)
                        {
                              m = tid*chunksize+k;
                              compare_exchange(*(A+m), *(A+m+1));
                        }
                  }//end of omp parallel for
            }// end of parallel region
      } // end for loop of i

      finish = clock();
      dur = (finish-start)/(double)CLOCKS_PER_SEC;
      printf("\nTime to parallel sort data of %d items is
%.4lf",MAXITEMS,dur);
```

```
      //printf("\nSequential duration is: %.4lf seconds", seqdur);
      if(seqdur > 0)
            printf("\nSpeed up ratio is: %.4lf per cent",(seqdur -
dur)/seqdur*100);
      //printf("\n");

      // check to see if data sorted
      printf("\n***** After parallel sort: *****\ncores = %d, chunksize
= %d & halfchunksize = %d",num_procs,chunksize,halfchunk);
      //print_data(A);
      isSorted(A);
} // end of sort

double bubble_sort(int *A)
{
      int i, j;
      // variables to measure the computation time
      clock_t start, finish;
      double dur;

      // start the timer
      start = clock();
      for(i = 1; i<MAXITEMS; i++)
            for(j = 0; j< MAXITEMS - i; j++)
                  compare_exchange(*(A+j), *(A+j+1));
      finish = clock();
      dur = (finish-start)/(double)CLOCKS_PER_SEC;
      printf("\nTime to bubble sort data of %d items is
%.4lf",MAXITEMS,dur);

      return dur;
} // end of bubble sort

void undosort(int *A, int *copyA)
{
      for(int i = 0; i<MAXITEMS; i++)
            A[i] = copyA[i];
} // undo sort data set

// main function
int main(int argc, char **argv)
{
      printf("\nOPENMP ODD-EVEN SORT:\n");

      bool sorted = false; // hold true if data is sorted
      double seqdur;
      int *A,*copyA; // the given set of data
      A = (int *)malloc(MAXITEMS * sizeof(int));
      copyA = (int *)malloc(MAXITEMS * sizeof(int));

      int i;
      // Assign random values to A
      for(i = 0; i<MAXITEMS; i++)
            A[i]= copyA[i] = rand()%1000;

      int num_procs; // number of maximum processors can be used
```

```
#pragma omp parallel
      {
            num_procs = omp_get_num_threads();
      } // end of parallel region
      bool numThrdspowerof2 = ispowerof2(num_procs);

      int more = 1;
      while(more == 1)
      {
            seqdur = bubble_sort(A);
            // check to see if data sorted
            //isSorted(A);
            undosort(A,copyA);

/*          if(numThrdspowerof2)
            {
                  i = 2;
                  while(i < num_procs)
                  {
                        sort(A,MAXITEMS/i,i,seqdur);
                        undosort(A,copyA);
                        i *= 2;
                  }
            }*/
            sort(A,MAXITEMS/num_procs,num_procs,seqdur);
            undosort(A,copyA);
            printf("\nwould you like more test? (0/1) ");
            scanf_s("%d",&more,1);
      }

      //delete [] A;
      return 0;
   }
```

6/ OpenMP – Graphics Rendering Simulation:

```
// Author: Thuy Nguyenphuc
// Date: 9/18/2009

// GraphicsSimulation.cpp : Simulate the graphic rendering.
// This program is the simplest OpenMP program and is used as
// the starting point

#include "stdafx.h"
#include <omp.h>
#include <stdlib.h>
#include <time.h>

#define MAXDATA 10000
#define CHUNK_SIZE 10

int _tmain(int argc, _TCHAR* argv[])
{
    int myRunTime = atoi(argv[1]);
    int myloop = 0;
    /*execution time measurement*/
    clock_t start, finish;
    double dur, total_rt = 0.0; // run time of program
    int tid; // processor's id
    int chunk = CHUNK_SIZE;

    while(myloop < myRunTime)
    {
        start = clock();
        #pragma omp parallel for private(tid) schedule(static,chunk)
        for(int i=0; i <MAXDATA; i++)
        {
            tid = omp_get_thread_num();
            if(tid%2 == 0)
                printf("\n\tAAAAAAAAAAAAAAAAAAAAAAAAA");
            if(tid%2 == 1)
                printf("\nBBBBBBBBBBBBBBBBBBBBBBBBB");
        }

        finish = clock();

        total_rt += (double)(finish - start) / CLOCKS_PER_SEC;
        myloop++;
    }
    dur = total_rt / myRunTime;
    printf("\n\nParallel execution time of one complete sort is %1.3f
           s.", dur);
    printf("\nParallel execution time of %d calculations is %f s.\n",
           myRunTime, total_rt);

    return 0;
}
```

7/ OpenMP – Sum of *N* integers:

```
/*
File name: ompSumN.cpp
Author: Thuy Nguyenphuc
OpenMP program to test multicore performance
The maximum number of cores is eight
*/
#include <stdio.h>
#include <stdlib.h>
// for clock_t type
#include <time.h>
// for OpenMP
#include <omp.h>

// constants
#define MAXITEMS 400000000
#define MAXLOOPS 10

// helper functions
double test(int *A, int chunksize, int numthreads, double seqdur)
{
      int i, j;
      // variables to measure the computation time
      clock_t start, finish;
      double sum, dur = 0.0;

      // Loops for the accuracy of the runtime
      for(j=0; j<MAXLOOPS; j++)
      {
            // start the timer
            start = clock();

            // the result sum is stored here
            sum = 0.0;

            // parallel computation start here
#pragma omp parallel for private(i) shared(chunksize,A)
schedule(static,chunksize) reduction(+:sum)
            for(i=0; i<MAXITEMS; i++)
            {
                  sum = sum +(double)A[i];
            } // end of parallel region

            finish = clock();
            dur = dur + ((finish-start)/(double)CLOCKS_PER_SEC);
      } // end of loops for accumulation of duration
      printf("\nThe sum is %lf", sum);
      if(numthreads > 1)
            printf("\nComputes the sum with %d cores and %d of chunk
size", numthreads, chunksize);
      else printf("\nComputes the sum with %d core and %d of chunk
size", numthreads, chunksize);
      printf("\nThe runtime is: %.8lf seconds", dur = dur/MAXLOOPS);
      if(seqdur > 0)
            printf("\nSpeed up ratio is: %.4lf per cent", (seqdur -
dur)/seqdur*100);
      printf("\n");
      return dur;
}

bool ispowerof2(int numthreads)
{
      bool res;
      if(numthreads < 2) res = false;
      else
      {
            while(numthreads >= 2)
            {
                  if((numthreads%2) == 0)
                  {
                        if(numthreads == 2)
                        {
                              res = true;
                              break;
                        }
                        numthreads /= 2;
                  }
                  else
                  {
                        res = false;
                        break;
                  }
            }
      }
      return res;
}

int main(int argc, char *argv)
{
      int i, numthreads = 1;
      bool numThrdspowerof2 = false;
      double seqdur = 0.0;

      printf("\n\nOPENMP:");

      //prepare array A
      int *A;
      A = (int *)malloc(MAXITEMS * sizeof(int));
      for(i=0; i<MAXITEMS; i++)
            A[i] = rand()%10;

      int more = 1;

#pragma omp parallel
      {
            numthreads = omp_get_num_threads();
      }

      numThrdspowerof2 = ispowerof2(numthreads);

      while(more == 1)
      {
            printf("\n*****Compute sum of %d arbitrary
numbers*****\n",MAXITEMS);
            if(numThrdspowerof2)
            {
                  i = 1;
                  while(i < numthreads)
                  {
                        if(i==1) seqdur = test(A,MAXITEMS/i,i,0);
                        else test(A,MAXITEMS/i,i,seqdur);
                        i *= 2;
                  }
            }
            test(A,MAXITEMS/numthreads,numthreads,seqdur);
            printf("\nwould you like more test? (0/1) ");
            scanf_s("%d",&more,1);
      }

      delete [] A;

      return 0;
   }
```

Appendix C - MPI source codes

1/ MPI – Maclaurin Series of e^x:

```
// Author: Thuy Nguyenphuc
// Date: 1/18/2010

// Maclaurin_MPI.cpp : Parallel Arithmetic computation using MPI.
//

#include "stdafx.h"
#include <mpi.h>
#include <time.h>
#include <math.h>

#define HALFTERMS 500
#define MAXTERMS HALFTERMS*2
#define E 2.71828

int _tmain(int argc, _TCHAR* argv[])
{
      /*MPI variables*/
      int num_procs;
      int ID;
      int tag1=1;
      MPI_Status stat;

      /*Maclaurin's variables*/
      float x = 0.5;
      double sum1 = 1, sum2 = 0,sum;
      double nfac_value1, nfac_value2;
      long nfac_base1, nfac_base2;

      /*execution time measurement*/
      clock_t start, finish;
      double dur;
      /*MPI execution time*/
      double MPIStart, MPIFinish;

      start = clock();
      MPI_Init(&argc,&argv);
      MPI_Comm_size(MPI_COMM_WORLD,&num_procs);
      MPI_Comm_rank(MPI_COMM_WORLD,&ID);

      MPIStart = MPI_Wtime();

      if(ID%2 == 0)
      {
            for (int i=1; i < HALFTERMS; i++)
            {
                  nfac_value1 = 1;
                  nfac_base1 = 0;
                  for(int j = 1; j <= i; j++)
                  {
                        nfac_value1 *= j;
                        if(nfac_value1 > 1000)
                        {
                              nfac_value1 /= 1000;
                              nfac_base1 += 3;
                        }
                  }
                  sum1 = sum1 + pow(x,i) / (nfac_value1 *
(float)pow(10.0,nfac_base1));
            }
            MPI_Send(&sum1,1,MPI_DOUBLE,1,tag1,MPI_COMM_WORLD);
      }
      if(ID%2 == 1)
      {
            for (int k=HALFTERMS; k < MAXTERMS; k++)
            {
                  nfac_value2 = 1;
                  nfac_base2 = 0;
                  for(int h = 1; h <= k; h++)
                  {
                        nfac_value2 *= h;
                        if(nfac_value2 > 1000)
                        {
                              nfac_value2 /= 1000;
                              nfac_base2 += 3;
                        }
                  }
            sum2 = sum2 + pow(x,k) / (nfac_value2 *
(float)pow(10.0,nfac_base2));
        }
        MPI_Recv(&sum,1,MPI_DOUBLE,0,tag1,MPI_COMM_WORLD,&stat);
//      printf("\nThe sum from P0 is %f",sum);
//      printf("\nThe sum from P1 is %f",sum2);
        sum +=sum2;
        printf("\nThe parallel estimation of Maclaurin of e^%1.2f
is: %f", x, sum);
    }

    MPIFinish = MPI_Wtime();
    printf("\nThe measured 1 second sleep time is %1.2f",MPIFinish -
MPIStart);

    MPI_Finalize();

    finish = clock();
    dur = (double)(finish - start) / CLOCKS_PER_SEC;
    printf("\n\nParallel execution time of one complete sort is %1.3f
s.", dur);

    return 0;
}
```

2/ MPI – Dot Product:

```
// Author: Thuy Nguyenphuc
// Date: 1/05/2010

// DotProduct.cpp : Parallel arithmetic computation using MPI.
// Master and slave model

#include "stdafx.h"
#include <mpi.h>
#include "windows.h"
#include <time.h> // for clock_t type

#define HALFMAX  2500000
#define MAXITEMS HALFMAX*2

int _tmain(int argc, _TCHAR* argv[])
{
    // time variables to measure the draw operation's execution time
    clock_t start, finish;
    double dur;
    /*MPI execution time*/
    double MPIStart, MPIFinish, mstart1,mstart2,mfinish1,mfinish2;

    int tag1 = 1, tag2 = 2;//message tags
    int num_procs;   // number of processes
    int ID; // a unique ID of a process
    MPI_Status stat; // MPI status parameter
```

```
      int *buf01;
      int *buf02;
      int *buf21;       //message3: p2 receives from p1
      int *buf22;       //message4: p2 receives from p1
      double res03;
      double res23;

      buf01 = (int *)malloc(MAXITEMS * sizeof(int)); //message1: from
p1 to p2
      buf02 = (int *)malloc(MAXITEMS * sizeof(int)); //message2: from
p1 to p2

      buf21 = (int *)malloc(MAXITEMS * sizeof(int)); //message3: p2
receives from p1
      buf22 = (int *)malloc(MAXITEMS * sizeof(int)); //message4: p2
receives from p1

      double acc_res;     //stores the scalar result of dot product
      int i,j;

      //initialize data to buffer 01 and 02
      for(i = 0; i < MAXITEMS; i++)
      {
            *(buf01+i) = rand()% 10;
            *(buf02+i) = rand()% 10;
      }

      //Start to count time here
      start = clock();

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &ID);
      MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

      //Sleep(15000);
      MPIStart = MPI_Wtime();
      if(ID==0)
      {
            mstart1 = MPI_Wtime();
            MPI_Send(buf01, HALFMAX, MPI_INT, 1, tag1, MPI_COMM_WORLD);
            MPI_Send(buf02, HALFMAX, MPI_INT, 1, tag1, MPI_COMM_WORLD);
            mfinish1 = MPI_Wtime();
            printf("\nThread %d: MPI_Wtime - P0 send time
%1.2f",ID,mfinish1 - mstart1);

            acc_res = 0.0;
            for(i = HALFMAX; i< MAXITEMS; i++)
                  acc_res += (double)(buf01[i]*buf02[i]);

            MPI_Recv(&res03, 1, MPI_DOUBLE, 1, tag2, MPI_COMM_WORLD,
&stat);
            acc_res += res03;

            printf("\nDot Product result is %f\n", acc_res);
      }
      else
      {
            mstart2 = MPI_Wtime();
            MPI_Recv(buf21, HALFMAX, MPI_INT, 0, tag1, MPI_COMM_WORLD, 
&stat);
            MPI_Recv(buf22, HALFMAX, MPI_INT, 0, tag1, MPI_COMM_WORLD, 
&stat);
            mfinish2 = MPI_Wtime();
            printf("\nThread %d: MPI_Wtime - P1 recv time 
%1.2f",ID,mfinish2 - mstart2);

            res23 = 0.0;
            for(j = 0; j<HALFMAX; j++)
                  res23 += (double)(buf21[j]*buf22[j]);

            MPI_Send(&res23, 1, MPI_DOUBLE, 0, tag2, MPI_COMM_WORLD);
      }

      MPIFinish = MPI_Wtime();
      printf("\nThread %d: Measured using MPI_Wtime is 
%1.2f",ID,MPIFinish - MPIStart);
      MPI_Finalize();

      /* Work's done. Get the elapsed wall time. */
//    if(ID==0)
//    {
            finish = clock();
            dur = (double)(finish - start) / CLOCKS_PER_SEC;
            printf("\n\nThread %d: Calculation time of vector size %d 
is %f s.",ID,MAXITEMS,dur);
//    }


      return 0;
}
```

3/ MPI – Dot Product (version 2):

```
#include <stdio.h>  // for printf and scanf_s
#include <stdlib.h> // for malloc and rand
#include <mpi.h>
#include <time.h> // for clock_t type
#include "windows.h"

// constants
#define MAXCORES 100
#define MAXITEMS 20000000

// helper functions
void master(int *A, int *B, int chunksize, int num_procs, int ID, 
double seqdur)
{
      // time variables to measure the draw operation's execution time
      clock_t start, finish;
      double dur;
      /*MPI execution time*/
      //double MPIStart, MPIFinish;
      MPI_Status stat;        // MPI status parameter
      double sumReturned;
      double res;             // result sums from all processors
      int i;

      /* Start to measure time */
      start = clock();
      //MPIStart = MPI_Wtime();

      for(i=1; i<num_procs; i++)
      {
            MPI_Send(A+(i-1)*chunksize, chunksize, MPI_INT, i, ID,
MPI_COMM_WORLD);
            MPI_Send(B+(i-1)*chunksize, chunksize, MPI_INT, i, ID,
MPI_COMM_WORLD);
      }

      res = 0.0;
      for(i = (num_procs - 1)*chunksize; i< MAXITEMS; i++)
            res = res + (double)((*(A+i)) * (*(B+i)));
//    printf("\nThe sum from P%d is %lf", ID,res);

      for(i=1; i<num_procs; i++)
      {
            MPI_Recv(&sumReturned, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
          res = res + sumReturned;
      }
      printf("\nP%d: The chunk size is %d.", ID,chunksize);
      printf("\nP%d: The sum result from all processors is %lf",
ID,res);

      //MPIFinish = MPI_Wtime();
      //printf("\nThread %d: Measured using MPI_Wtime is
%1.2f",ID,MPIFinish - MPIStart);

      /* Work's done. Get the elapsed wall time. */
      finish = clock();
      dur = (double)(finish - start) / CLOCKS_PER_SEC;
      printf("\n\n*****>Compute time of dot-product of vectors of size
%d is %f s.<*****\n",MAXITEMS,dur);

      if(seqdur >= dur)
            printf("\nP%d: Speed up ratio is: %.4lf per cent",
ID,(seqdur - dur)/seqdur*100);
      else printf("\nP%d: Slow down ratio is: %.4lf per cent\n",
ID,(dur - seqdur)/dur*100);
}

void slave(int chunksize, int ID)
{
      MPI_Status stat;        // MPI status parameter
      int *tmpBufA,*tmpBufB,j;
      tmpBufA = (int *)malloc(MAXITEMS * sizeof(int)); //memory for sub
set of A on thread ID != 0
```

```
      tmpBufB = (int *)malloc(MAXITEMS * sizeof(int)); //memory for sub
set of B on thread ID != 0

      MPI_Recv(tmpBufA, chunksize, MPI_INT, 0,  MPI_ANY_TAG,
MPI_COMM_WORLD, &stat);
      MPI_Recv(tmpBufB, chunksize, MPI_INT, 0,  MPI_ANY_TAG,
MPI_COMM_WORLD, &stat);

      double tmpRes = 0.0;
      for(j = 0; j<chunksize; j++)
            tmpRes = tmpRes + (double)((*(tmpBufA+j)) *
(*(tmpBufB+j)));
//    printf("\nThe sum from P%d is %lf\n",ID, tmpRes);

      MPI_Send(&tmpRes, 1, MPI_DOUBLE, 0, ID, MPI_COMM_WORLD);

      delete [] tmpBufA;
      delete [] tmpBufB;
}

int main(int argc, char **argv)
{
      int num_procs;                // number of processes
      int ID;                             // a unique ID of a process
      int chunksize;                // number of integers that are
assigned to processors

      // variables to measure the computation time
      clock_t start, finish;
      double seqscalar, seqdur = 0.0;

      int *A, *B;                         // set A of n numbers
      A = (int *)malloc(MAXITEMS * sizeof(int)); //allocate memory for
set A
      B = (int *)malloc(MAXITEMS * sizeof(int)); //allocate memory for
set B

      //initialize data to buffer A
      for(int i = 0; i < MAXITEMS; i++)
      {
            A[i] = rand()% 10;
            B[i] = rand()% 10;
      }

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &ID);
      MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

      chunksize = MAXITEMS/num_procs;

//    Sleep(10000);
      if(ID == 0)
      {
            printf("\n\nMPI:");
            start = clock();
            seqscalar = 0.0;
            for(int i = 0; i < MAXITEMS; i++)
            {
                  seqscalar = seqscalar + A[i]*B[i];
            }
            finish = clock();
            seqdur = (double)(finish - start)/CLOCKS_PER_SEC;
            if(seqdur == 0) seqdur = 0.0001;

            printf("\nSequential scalar dot-product value is: 
%lf",seqscalar);
            printf("\nSequential duration is: %lf",seqdur);

            master(A, B, chunksize, num_procs, ID, seqdur);
            delete [] A;
            delete [] B;
      }
      else
      {
            slave(chunksize, ID);
      }

//    MPIStart = MPI_Wtime();
//    Sleep(1000);
//    MPIFinish = MPI_Wtime();
//  printf("\nThread %d: Measured using MPI_Wtime is 
%1.2f",ID,MPIFinish - MPIStart);

      MPI_Finalize();

      return 0;
}
```

4/ MPI – Odd-Even Transposition:

```
// Author: Thuy Nguyenphuc
// Date: 1/20/2010

// ParaSort.cpp : Parallel sort using MPI. Master-slave model.
//

#include "stdafx.h"
#include <mpi.h>
#include <stdlib.h> //for malloc() and rand()
#include <time.h>

#define QUARTMAX  800
#define HALFMAX   QUARTMAX*2
#define MAXITEMS  HALFMAX*2

void sort(int *data);
void prnt(int *data);
void d_copy(int *dest, int *src, int count);
void test_result(int *data);

int _tmain(int argc, _TCHAR* argv[])
```

```
{
        int num_procs;
        int ID;
        int tag1=1, tag2=2;
        MPI_Status stat;

        /*execution time measurement*/
        clock_t start, finish;
        double dur;
        /*MPI execution time*/
        double MPIStart, MPIFinish, mstart1, mstart2, mfinish1, mfinish2;

        int *A, *B, *workA, *workB;
        int i, dummy1=1, dummy2=0;

        A = (int *)malloc(HALFMAX * sizeof(int));
        B = (int *)malloc(HALFMAX * sizeof(int));
        workA = (int *)malloc(HALFMAX * sizeof(int));
        workB = (int *)malloc(HALFMAX * sizeof(int));
        for(i = 0; i<HALFMAX; i++)
        {
                A[i] = rand();
                B[i] = rand();
        }

        /* Start time here */
        start = clock();

        /*initial MPI*/
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &ID);
        MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

        MPIStart = MPI_Wtime();
        for(i=0; i<num_procs; i++)
        {
                if(i%2 == 0)//even iterator
                {
                        if(ID%2 == 0)//even process
                        {
                                sort(A);
                                mstart1 = MPI_Wtime();

        MPI_Send(A+QUARTMAX,QUARTMAX,MPI_INT,1,tag1,MPI_COMM_WORLD);
                                mfinish1 = MPI_Wtime();
                                printf("\nP0 sent time is %1.2f",mfinish1 -
mstart1);
                                mstart1 = MPI_Wtime();

        MPI_Recv(A+QUARTMAX,QUARTMAX,MPI_INT,1,tag2,MPI_COMM_WORLD,
&stat);
                                mfinish1 = MPI_Wtime();
                                printf("\nP0 recv time is %1.2f",mfinish1 -
mstart1);
                        }
                        else //odd process
```

```
                  {
                        sort(B);
                        mstart2 = MPI_Wtime();

      MPI_Recv(workB,QUARTMAX,MPI_INT,0,tag1,MPI_COMM_WORLD, &stat);
                        mfinish2 = MPI_Wtime();
                        printf("\nP1 recv time is %1.2f",mfinish2 -
mstart2);
                        //memcpy(workB+QUARTMAX,B,QUARTMAX);
                        d_copy(workB+QUARTMAX,B,QUARTMAX);
                        sort(workB);
                        mstart2 = MPI_Wtime();

      MPI_Send(workB,QUARTMAX,MPI_INT,0,tag2,MPI_COMM_WORLD);
                        mfinish2 = MPI_Wtime();
                        printf("\nP1 sent time is %1.2f",mfinish2 -
mstart2);
                        //memcpy(B,workB+QUARTMAX,QUARTMAX);
                        d_copy(B,workB+QUARTMAX,QUARTMAX);
                  }
            }
            else // odd iterator
            {
                  if(ID%2 == 1) // odd process
                  {
                        sort(B);

      MPI_Send(B,QUARTMAX,MPI_INT,0,tag2,MPI_COMM_WORLD);

      MPI_Recv(B,QUARTMAX,MPI_INT,0,tag1,MPI_COMM_WORLD, &stat);
                  }
                  else // even process
                  {
                        sort(A);

      MPI_Recv(workA+QUARTMAX,QUARTMAX,MPI_INT,1,tag2,MPI_COMM_WORLD,
&stat);
                        //memcpy(workA,A+QUARTMAX,QUARTMAX);
                        d_copy(workA,A+QUARTMAX,QUARTMAX);
                        sort(workA);

      MPI_Send(workA+QUARTMAX,QUARTMAX,MPI_INT,1,tag1,MPI_COMM_WORLD);
                        //memcpy(A+QUARTMAX,workA,QUARTMAX);
                        d_copy(A+QUARTMAX,workA,QUARTMAX);
                  }
            }
      }

      if(ID==0)
      {
            test_result(A);
            if(A[HALFMAX-1] > B[0]) printf("\nSort program failed at
the middle point");
            delete [] A;
            delete [] workA;
      }
      else
      {
            test_result(B);
            delete [] B;
            delete [] workB;
      }


      MPIFinish = MPI_Wtime();
      printf("\nThread %d: The execution time using MPI_Wtime is 
%1.2f",ID,MPIFinish - MPIStart);

      MPI_Finalize();

      /* Finish MPI tasks */
      finish = clock();
      dur = (double)(finish - start) / CLOCKS_PER_SEC;
      printf("\nThread %d: The execution time of sorting data set of 
size %d is %1.3f s.",ID,MAXITEMS, dur);
      return 0;
}

void sort(int *data)
{
      int temp;

      for(int i = 1; i <HALFMAX; i++)
            for(int j = 0; j <HALFMAX - i; j++)
                  if(data[j] > data[j+1])
                        {
                              temp = data[j];
                              data[j] = data[j+1];
                              data[j+1] = temp;
                        }

}

void prnt(int *data)
{
      printf("Data:");
      for(int i = 0; i< HALFMAX; i++)
      {
            if((i%10)==0) printf("\n");
            printf("%5d ", data[i]);
      }

}

void d_copy(int *dest, int *src, int count)
{
      for(int i = 0; i<count; i++)
            *(dest+i) = *(src+i);
}

void test_result(int *data)
{
      for(int i = 1; i<HALFMAX; i++)
      {
            if(*(data+i) < *(data+i)-1)
            {
                  printf("\nSort program failed at position %d", i+1);
            }
      }
}
```

5/ MPI – Graphics Rendering (simulation):

```
// Author: Thuy Nguyenphuc
// Date: 1/10/2010

// Graphics_MPI.cpp : The simplest MPI program.
// It is treated as the Hello program in MPI

#include "stdafx.h"
#include <mpi.h>
#include <time.h>

#define HALFITEMS 60000
#define MAXITEMS HALFITEMS*2

int _tmain(int argc, _TCHAR* argv[])
{
      int num_procs;
      int ID;
      MPI_Status stat;

      /*execution time measurement*/
      clock_t start, finish;
      double dur;
      /*MPI execution time*/
      double MPIStart, MPIFinish;

      start = clock();
      MPI_Init(&argc,&argv);
      MPI_Comm_size(MPI_COMM_WORLD,&num_procs);
      MPI_Comm_rank(MPI_COMM_WORLD,&ID);
      MPIStart = MPI_Wtime();
      if(ID%2 == 0)
            for(int i = 0; i<HALFITEMS; i++)
                  printf("\n\tAAAAAAAAAAAAAAAAAAAAAAAAA");
      if(ID%2 == 1)
            for(int j = 0; j<HALFITEMS; j++)
                  printf("\nBBBBBBBBBBBBBBBBBBBBBBBBB");

      MPIFinish = MPI_Wtime();
      printf("\nThread %d: The execution time using MPI_Wtime()is 
%1.2f\n",ID,MPIFinish - MPIStart);

      MPI_Finalize();
      finish = clock();
      dur = (double)(finish - start) / CLOCKS_PER_SEC;
      printf("\nThread %d: Parallel display calls of %d lines of
strings is %1.3f s.\n",ID,MAXITEMS, dur);

      return 0;
}
```

6/ MPI – Sum of n integers:

```
/*
File name: mpiSumN.cpp
Author: Thuy Nguyenphuc
Date: 04/24/2010
Compute sum of n integers in mpi and measure computation time.
*/
#include <stdio.h>  // for printf and scanf_s
#include <stdlib.h> // for malloc and rand
#include <mpi.h>
#include <time.h> // for clock_t type
#include "windows.h"

// constants
#define MAXCORES 100
#define MAXITEMS 200000000

// helper functions
void master(int *A, int chunksize, int num_procs, int ID, double
seqdur)
{
      // time variables to measure the draw operation's execution time
      clock_t start, finish;
      double dur;
      /*MPI execution time*/
      //double MPIStart, MPIFinish;

      MPI_Status stat;        // MPI status parameter
      double sumReturned;
      double res;             // result sums from all processors
      int i;

      /* Start to measure time */
      start = clock();
      //MPIStart = MPI_Wtime();

      for(i=1; i<num_procs; i++)
            MPI_Send(A+(i-1)*chunksize, chunksize, MPI_INT, i, ID,
MPI_COMM_WORLD);

      res = 0.0;
      for(i = (num_procs - 1)*chunksize; i< MAXITEMS; i++)
            res = res + (double)(*(A+i));
//    printf("\nThe sum from P%d is %lf", ID,res);
```

```
      for(i=1; i<num_procs; i++)
      {
            MPI_Recv(&sumReturned, 1, MPI_DOUBLE, MPI_ANY_SOURCE,
MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
          res = res + sumReturned;
      }
      printf("\nP%d: The chunk size is %d.", ID,chunksize);
      printf("\nP%d: The sum result from all processors is %lf",
ID,res);

      //MPIFinish = MPI_Wtime();
      //printf("\nThread %d: Measured using MPI_Wtime is
%1.2f",ID,MPIFinish - MPIStart);

      /* Work's done. Get the elapsed wall time. */
      finish = clock();
      dur = (double)(finish - start) / CLOCKS_PER_SEC;
      printf("\n\n*****>P%d: Calculation time of sum of %d items is %f
s.<*****\n",ID,MAXITEMS,dur);

      if(seqdur >= dur)
            printf("\nP%d: Speed up ratio is: %.4lf per cent",
ID,(seqdur - dur)/seqdur*100);
      else printf("\nP%d: Slow down ratio is: %.4lf per cent\n",
ID,(dur - seqdur)/dur*100);
}

void slave(int chunksize, int ID)
{
      MPI_Status stat;        // MPI status parameter
      int *tmpBuf, j;
      tmpBuf = (int *)malloc(MAXITEMS * sizeof(int)); //memory for sub
set of A on thread ID != 0

      MPI_Recv(tmpBuf, chunksize, MPI_INT, 0,  MPI_ANY_TAG,
MPI_COMM_WORLD, &stat);

      double tmpRes = 0.0;
      for(j = 0; j<chunksize; j++)
            tmpRes = tmpRes + (double)(*(tmpBuf+j));
//    printf("\nThe sum from P%d is %lf\n",ID, tmpRes);

      MPI_Send(&tmpRes, 1, MPI_DOUBLE, 0, ID, MPI_COMM_WORLD);

      delete [] tmpBuf;
}

bool ispowerof2(int numthreads)
{
      bool res;
      if(numthreads < 2) res = false;
      else
      {
            while(numthreads >= 2)
            {
                  if((numthreads%2) == 0)
                  {
                        if(numthreads == 2)
                        {
                              res = true;
                              break;
                        }
                        numthreads = numthreads / 2;
                  }
                  else
                  {
                        res = false;
                        break;
                  }
            }
      }
      return res;
}

int main(int argc, char **argv)
{
      int num_procs;                // number of processes
      int ID;                             // a unique ID of a process
      int chunksize;                // number of integers that are
assigned to processors
      bool powof2;                  // check if the total processors
used is a power of 2

      // variables to measure the computation time
      clock_t start, finish;
      double seqsum, seqdur = 0.0;

      int *A;                             // set A of n numbers
      A = (int *)malloc(MAXITEMS * sizeof(int)); //allocate memory for
set A

      //initialize data to buffer A
      for(int i = 0; i < MAXITEMS; i++)
            A[i] = rand()% 10;

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &ID);
      MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

      chunksize = MAXITEMS/num_procs;

//    Sleep(10000);
      if(ID == 0)
      {
            printf("\n\nMPI:");
            start = clock();
            seqsum = 0.0;
            for(int i = 0; i < MAXITEMS; i++)
            {
                  seqsum = seqsum + A[i];
            }
            finish = clock();
            seqdur = (double)(finish - start)/CLOCKS_PER_SEC;
            if(seqdur == 0) seqdur = 0.0001;

            printf("\nSequential sum value is: %lf",seqsum);
            printf("\nSequential duration is: %lf",seqdur);

/*          powof2 = ispowerof2(num_procs);
            if(powof2)
            {
                  int i = 2;
                  while(i < num_procs)
                  {
                        master(A, MAXITEMS/i, i, ID, seqdur);
                        i = i*2;
                  }
            }*/
            master(A, chunksize, num_procs, ID, seqdur);
            delete [] A;
      }
      else
      {
            slave(chunksize, ID);
      }


      MPI_Finalize();

      return 0;
}
```

Appendix D - OpenCL source codes

1/ OpenCL – Dot Product:

```
// Author: Thuy Nguyenphuc
// Date: 2/07/2010

// thDotProduct.cpp: Parallel arithmetic computation in OpenCL

#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>
// for clock_t type
#include <time.h>

#define LOCALDATASIZE 1//256
// Number of elements in the vectors to be added
#define SIZE 10000000

// OpenCL source code
const char* OpenCLSource[] = {
      "__kernel void VectorAdd(__global int* c, __global int*
a,__global int* b)",
      "{",
      " // Index of the elements to add \n",
        " unsigned int n = get_global_id(0);",
        " // Sum the n’th element of vectors a and b and store in c \n",
        " c[n] = a[n] * b[n];",
        "}"
};

// Main function
//
**********************************************************************
int main(int argc, char **argv)
{
        // time variables to measure the draw operation's execution time
        clock_t start, finish;
        double dur; // for execution time calculation

        // Two integer source vectors in Host memory
        int *HostVector1, *HostVector2;
        HostVector1 = (int *)malloc(SIZE * sizeof(int));
        HostVector2 = (int *)malloc(SIZE * sizeof(int));
        // Initialize with some interesting repeating data
        for(int c = 0; c < SIZE; c++)
        {
                HostVector1[c] = rand()% 10;
                HostVector2[c] = rand()% 10;
        }
        // Create a context to run OpenCL on our CUDA-enabled NVIDIA GPU
        cl_int err_num;
        printf("\nCreate a GPU context");
        cl_context GPUContext = clCreateContextFromType(0,
CL_DEVICE_TYPE_GPU,
        NULL, NULL, &err_num);

        // Get the list of GPU devices associated with this context
        size_t ParmDataBytes;
        printf("\nclGetContextInfo\n...");
        clGetContextInfo(GPUContext, CL_CONTEXT_DEVICES, 0, NULL,
&ParmDataBytes);
        cl_device_id* GPUDevices = (cl_device_id*)malloc(ParmDataBytes);
        clGetContextInfo(GPUContext, CL_CONTEXT_DEVICES, ParmDataBytes,
GPUDevices, NULL);

        // Create a command-queue on the first GPU device
        printf("clCreateCommandQueue\n...");
        cl_command_queue GPUCommandQueue =
clCreateCommandQueue(GPUContext,

                        GPUDevices[0], 0, NULL);
        // Allocate GPU memory for source vectors AND initialize from CPU
memory
        printf("\nAllocate GPU memory for source vectors AND initialize
from CPU memory");
        cl_mem GPUVector1 = clCreateBuffer(GPUContext, CL_MEM_READ_ONLY,
                                                        sizeof(int) * SIZE,
HostVector1, &err_num);
        if(err_num != CL_SUCCESS)
                printf("Error in create buffer 1 - %d",err_num);
```

```
      cl_mem GPUVector2 = clCreateBuffer(GPUContext, CL_MEM_READ_ONLY,
                                                sizeof(int) * SIZE,
HostVector2, &err_num);
      if(err_num != CL_SUCCESS)
            printf("Error in create buffer 2 - %d",err_num);

      // Allocate output memory on GPU
      printf("\nAllocate output memory on GPU");
      cl_mem GPUOutputVector = clCreateBuffer(GPUContext,
CL_MEM_WRITE_ONLY,

      sizeof(int) * SIZE, NULL, &err_num);
      if(err_num != CL_SUCCESS)
            printf("Error in create output buffer - %d",err_num);

      // Create OpenCL program with source code
      printf("\nCreate OpenCL program with source code");
      cl_program OpenCLProgram = clCreateProgramWithSource(GPUContext,
7,

            OpenCLSource, NULL, NULL);
      // Build the program (OpenCL JIT compilation)
      // start to measure the time build the program
      start = clock();
      printf("\nBuild the program (OpenCL JIT compilation)");
      clBuildProgram(OpenCLProgram, 0, NULL, NULL, NULL, NULL);
      finish = clock();
      dur = (double)(finish - start) / CLOCKS_PER_SEC;
      printf("\nTime to build program is %f", dur);

      // Create a handle to the compiled OpenCL function (Kernel)
      printf("\nCreate a handle to the compiled OpenCL function
(Kernel)");
      start = clock();
      cl_kernel OpenCLVectorAdd = clCreateKernel(OpenCLProgram,
"VectorAdd", NULL);

      // In the next step we associate the GPU memory with the Kernel
arguments
      printf("\nAssociate the GPU memory with the Kernel arguments");
      clSetKernelArg(OpenCLVectorAdd, 0,
sizeof(cl_mem),(void*)&GPUOutputVector);
      clSetKernelArg(OpenCLVectorAdd, 1, sizeof(cl_mem),
(void*)&GPUVector1);
      clSetKernelArg(OpenCLVectorAdd, 2, sizeof(cl_mem),
(void*)&GPUVector2);

      // Launch the Kernel on the GPU
      printf("\nLaunch the Kernel on the GPU");
      size_t WorkSize[1] = {SIZE}; // one dimensional Range
      size_t LocalWorkSize[1] = {LOCALDATASIZE};
      err_num = clEnqueueWriteBuffer(GPUCommandQueue, GPUVector1,
CL_FALSE, 0,
                                       sizeof(cl_int) * WorkSize[0],
HostVector1, 0, NULL, NULL);
```

```
      err_num |= clEnqueueWriteBuffer(GPUCommandQueue, GPUVector2,
CL_FALSE, 0,
                                       sizeof(cl_int) * WorkSize[0],
HostVector2, 0, NULL, NULL);
      if(err_num != CL_SUCCESS)
            printf("Error in write to GPUbuffers - %d",err_num);

      err_num = clEnqueueNDRangeKernel(GPUCommandQueue,
OpenCLVectorAdd, 1, NULL,
      WorkSize, LocalWorkSize, 0, NULL, NULL);
      if(err_num != CL_SUCCESS)
            printf("\n\nError in clEnqueueNDRangeKernel() -
%d\n",err_num);

      finish = clock();
      dur = (double)(finish - start) / CLOCKS_PER_SEC;
      printf("\nTime to run kernels only with data size of %d is
%f",SIZE, dur);

      // Copy the output in GPU memory back to CPU memory
      printf("\nCopy the output in GPU memory back to CPU memory");
      int *HostOutputVector;
      HostOutputVector = (int *)malloc(SIZE * sizeof(int));
//    int HostOutputVector[SIZE];
      err_num = clEnqueueReadBuffer(GPUCommandQueue, GPUOutputVector,
CL_TRUE, 0,
                            WorkSize[0] * sizeof(int),
HostOutputVector, 0, NULL, NULL);
      if(err_num != CL_SUCCESS)
            printf("\n\nError in clEnqueueReadBuffer() -
%d\n",err_num);

      finish = clock();
      dur = (double)(finish - start) / CLOCKS_PER_SEC;
      printf("\nTime to run kernels and copy data between device and
host is %f", dur);

      int sum = 0;
      for (int c = 0; c < SIZE; c++)
            sum += *(HostOutputVector+c);
      printf("\nDot Product value is %d", sum);

      finish = clock();
      dur = (double)(finish - start) / CLOCKS_PER_SEC;
      printf("\nTime to complete dot product calculation is %f", dur);

      // Cleanup
      printf("\nCleanup...\n");
      free(GPUDevices);
      clReleaseKernel(OpenCLVectorAdd);
      clReleaseProgram(OpenCLProgram);
      clReleaseCommandQueue(GPUCommandQueue);
      clReleaseContext(GPUContext);
      clReleaseMemObject(GPUVector1);
      clReleaseMemObject(GPUVector2);
      clReleaseMemObject(GPUOutputVector);
      delete [] HostVector1;
      delete [] HostVector2;
      delete [] HostOutputVector;
      return 0;
}
```

2/ OpenCL – Odd-Even Transposition:

```
// Author: Thuy Nguyenphuc
// Date: 3/18/2010
// OddEvenTransposition program is written in OpenCL for 8 cores
// of GPU Nvidia GeForce 9200M GS
// this program was sucessfully sort a set of data of size 32.  The
algorithm assumes that the number of processes
// is as same as the number of elements in the data set.
//
// Notes: The CHUNKSIZE must be an even number for the logic to work
correctly and
// the maximum CHUNKSIZE of 500 is limited by the card GeForce 9200M
GS.
// The bottle neck of the card is the resources when returning data to
host.
#include <stdio.h>
#include <stdlib.h>
#include <CL/cl.h>
// for clock_t type
#include <time.h>

#define LOCALDATASIZE 1
// Number of elements in the vectors to be added
#define MAXPROCESSES 8
#define CHUNKSIZE 500
#define SIZE MAXPROCESSES*CHUNKSIZE

// OpenCL source code
const char* OpenCLSource[] = {
      "__kernel void OddEvenSort(__global int* c, __global int* a, int
elements, int totalkernels, int loopindex)",
      "{",
      "   int i,j,m,temp,half = elements/2;",
      " // Index of the current element \n",
      " unsigned int n = get_global_id(0);",
      "// bound check (equivalent to the limit on a 'for' loop for
standard/serial C code\n",
      "     if(n >= totalkernels){",
      "           return;",
      "     }",
      "//Sort local data sequential bubble sort\n",
      "     for(i = 1; i < elements; i++){",
      "           for(j = 0; j < elements - i; j++){",
      "                 m = n*elements+j;",
      "                 if(a[m] > a[m+1]){",
      "                       temp = a[m];",
      "                       a[m] = a[m+1];",
      "                       a[m+1] = temp;",
      "                 }",
      "           }",
      "     }",
      "   if(loopindex % 2 == 0){",
      "           if(n % 2 == 0){",
      "                 if(n!=0){",
      "                       for(i = 0; i < elements; i++){",
      "                             for(j = 0; j < elements - i;
j++){",
      "                                   m = n*elements-half+j;",
      "                                   if(a[m] < a[m-1]){",
      "                                         temp = a[m];",
      "                                         a[m] = a[m-1];",
      "                                         a[m-1] = temp;",
      "                                   }",
      "                             }",
      "                       }",
      "                 }",
      "           }",
      "           if(n%2 == 1){",
      "                 if(n < totalkernels -1){",
      "                       for(i = 0; i < elements; i++){",
      "                             for(j = 0; j < elements - i;
j++){",
      "                                   m = n*elements+half+j;",
      "                                   if(a[m] > a[m+1]){",
      "                                         temp = a[m];",
      "                                         a[m] = a[m+1];",
      "                                         a[m+1] = temp;",
      "                                   }",
      "                             }",
      "                       }",
      "                 }",
      "           }",
      "     }",
      "   else{",
      "           if(n % 2 == 1){",
      "                 if(n!=0){",
      "                       for(i = 0; i < elements; i++){",
      "                             for(j = 0; j < elements - i;
j++){",
      "                                   m = n*elements-half+j;",
      "                                   if(a[m] < a[m-1]){",
      "                                         temp = a[m];",
      "                                         a[m] = a[m-1];",
      "                                         a[m-1] = temp;",
      "                                   }",
      "                             }",
      "                       }",
      "                 }",
      "           }",
      "           if(n%2 == 0){",
      "                 if(n < totalkernels -1){",
      "                       for(i = 0; i < elements; i++){",
      "                             for(j = 0; j < elements - i;
j++){",
      "                                   m = n*elements+half+j;",
      "                                   if(a[m] > a[m+1]){",
      "                                         temp = a[m];",
      "                                         a[m] = a[m+1];",
      "                                         a[m+1] = temp;",
      "                                   }",
      "                             }",
      "                       }",
      "                 }",
      "           }",
      "     }",
      "     for(i = 0; i < elements; i++)",
      "           c[n*elements+i] = a[n*elements+i];",
      "}"
};

// Main function
//
*********************************************************************
int main(int argc, char **argv)
{
      // time variables to measure the draw operation's execution time
      clock_t start, finish;
      double dur; // for execution time calculation

      // Two integer source vectors in Host memory
      int *HostVector;
      HostVector = (int *)malloc(SIZE * sizeof(int));
      // Initialize with some interesting repeating data
      for(int c = 0; c < SIZE; c++)
      {
            HostVector[c] = rand()%1000;
      }
      // Create a context to run OpenCL on our CUDA-enabled NVIDIA GPU
      cl_int err_num;
      printf("\nCreate a GPU context");
      cl_context GPUContext = clCreateContextFromType(0,
CL_DEVICE_TYPE_GPU,
      NULL, NULL, &err_num);

      // Get the list of GPU devices associated with this context
      size_t ParmDataBytes;
      printf("\nclGetContextInfo\n...");
      clGetContextInfo(GPUContext, CL_CONTEXT_DEVICES, 0, NULL,
&ParmDataBytes);
      cl_device_id* GPUDevices = (cl_device_id*)malloc(ParmDataBytes);
      clGetContextInfo(GPUContext, CL_CONTEXT_DEVICES, ParmDataBytes,
GPUDevices, NULL);

      // Create a command-queue on the first GPU device
      printf("clCreateCommandQueue\n...");
      cl_command_queue GPUCommandQueue =
clCreateCommandQueue(GPUContext,
                  GPUDevices[0], 0, NULL);
      // Allocate GPU memory for source vectors AND initialize from CPU
memory
      printf("\nAllocate GPU memory for source vectors AND initialize
from CPU memory");

      cl_mem GPUVector = clCreateBuffer(GPUContext, CL_MEM_READ_ONLY,
                                                sizeof(int) * SIZE,
HostVector, &err_num);
      if(err_num != CL_SUCCESS)
            printf("Error in create buffer 1 - %d",err_num);
      int GPULoopIndex = 0;
      int GPUElements = CHUNKSIZE;
      int GPUTotalKernels = MAXPROCESSES;

      // Allocate output memory on GPU
      printf("\nAllocate output memory on GPU");
      cl_mem GPUOutputVector = clCreateBuffer(GPUContext,
CL_MEM_WRITE_ONLY,

      sizeof(int) * SIZE, NULL, &err_num);
      if(err_num != CL_SUCCESS)
            printf("Error in create output buffer - %d",err_num);

      // Create OpenCL program with source code
      printf("\nCreate OpenCL program with source code");
      cl_program OpenCLProgram = clCreateProgramWithSource(GPUContext,
83,

            OpenCLSource, NULL, NULL);
      // Build the program (OpenCL JIT compilation)
      // start to measure the time build the program
      start = clock();
      printf("\nBuild the program (OpenCL JIT compilation)");
      clBuildProgram(OpenCLProgram, 0, NULL, NULL, NULL, NULL);
      finish = clock();
      dur = (double)(finish - start) / CLOCKS_PER_SEC;
      printf("\nTime to build program is %f", dur);

      // Create a handle to the compiled OpenCL function (Kernel)
      printf("\nCreate a handle to the compiled OpenCL function
(Kernel)");
      start = clock();
      cl_kernel OpenCLOddEvenSort = clCreateKernel(OpenCLProgram,
"OddEvenSort", NULL);

      // In the next step we associate the GPU memory with the Kernel
arguments
      printf("\nAssociate the GPU memory with the Kernel arguments");
      clSetKernelArg(OpenCLOddEvenSort, 0,
sizeof(cl_mem),(void*)&GPUOutputVector);
      clSetKernelArg(OpenCLOddEvenSort, 1, sizeof(cl_mem),
(void*)&GPUVector);
      clSetKernelArg(OpenCLOddEvenSort, 2, sizeof(cl_int),
(void*)&GPUElements);
```

```
      clSetKernelArg(OpenCLOddEvenSort, 3, sizeof(cl_int),
(void*)&GPUTotalKernels);
      //the fourth argument is in the loop of sorting

      // Launch the Kernel on the GPU
      printf("\nLaunch the Kernel on the GPU");
      size_t WorkSize[1] = {SIZE}; // one dimensional Range
      size_t LocalWorkSize[1] = {LOCALDATASIZE};

      err_num = clEnqueueWriteBuffer(GPUCommandQueue, GPUVector,
CL_TRUE, 0,
                                                          sizeof(cl_int)
* SIZE, HostVector, 0, NULL, NULL);
      if(err_num != CL_SUCCESS)
            printf("Error in write to GPUbuffers - %d",err_num);

      for(GPULoopIndex = 0; GPULoopIndex < MAXPROCESSES +1;
GPULoopIndex++)
      {
            clSetKernelArg(OpenCLOddEvenSort, 4, sizeof(cl_int),
(void*)&GPULoopIndex);
            err_num = clEnqueueNDRangeKernel(GPUCommandQueue,
OpenCLOddEvenSort, 1, NULL,
            WorkSize, LocalWorkSize, 0, NULL, NULL);
            if(err_num != CL_SUCCESS)
                  printf("\n\nError in clEnqueueNDRangeKernel() -
%d\n",err_num);
      }

      finish = clock();
      dur = (double)(finish - start) / CLOCKS_PER_SEC;
      printf("\nTime to run kernels only is %f", dur);

      // Copy the output in GPU memory back to CPU memory
      printf("\nCopy the output in GPU memory back to CPU memory");
      int *HostOutputVector;
      HostOutputVector = (int *)malloc(SIZE * sizeof(int));
//    int HostOutputVector[SIZE];
      err_num = clEnqueueReadBuffer(GPUCommandQueue, GPUOutputVector,
CL_TRUE, 0,
                            WorkSize[0] * sizeof(int),
HostOutputVector, 0, NULL, NULL);
      if(err_num != CL_SUCCESS)
            printf("\n\nError in clEnqueueReadBuffer() -
%d\n",err_num);
      finish = clock();
      dur = (double)(finish - start) / CLOCKS_PER_SEC;
      printf("\nTime to run kernels and copy data between device and
host is %f", dur);

      // Cleanup
      printf("\nCleanup...\n");
      free(GPUDevices);
      clReleaseKernel(OpenCLOddEvenSort);
      clReleaseProgram(OpenCLProgram);
      clReleaseCommandQueue(GPUCommandQueue);
      clReleaseContext(GPUContext);
      clReleaseMemObject(GPUVector);
      clReleaseMemObject(GPUOutputVector);
      // Print out the results
      printf("\n");
      int c;
      for(c = 0; c < 10; c++){
            if(c%8==0) printf("\n");
            printf("%5d",*(HostOutputVector + c));
      }
      for(c = 1; c < SIZE; c++){
            int temp = *(HostOutputVector + c);
            if(temp < *(HostOutputVector + c -1))
            {
                  printf("\nSort program failed at position %d/%d",
c+1, SIZE);
                  break;
            }
      }
      if(c == SIZE) printf("\nThe program sorts correctly the data set
of size %d!", c);

      delete [] HostVector;
      delete [] HostOutputVector;
      return 0;
}
```

3/ OpenCL – Graphics Rendering:

```
/*
 * Graphics.cpp
 * Copyright 1993-2009 NVIDIA Corporation.  All rights reserved.
 *
 * NVIDIA Corporation and its licensors retain all intellectual
property and
 * proprietary rights in and to this software and related
documentation.
 * Any use, reproduction, disclosure, or distribution of this software
 * and related documentation without an express license agreement from
 * NVIDIA Corporation is strictly prohibited.
 *
 * Please refer to the applicable NVIDIA end user license agreement
(EULA)
 * associated with this source code for terms and conditions that
govern
 * your use of this NVIDIA software.
 *
 */

//
*********************************************************************
```

```
// Demo application for postprocessing of OpenGL renderings with OpenCL
// Based on the CUDA postprocessGL sample
//
*********************************************************************

// standard utility and system includes
#include <oclUtils.h>

// GLEW and GLUT includes
#include <GL/glew.h>

#if defined(__APPLE__) || defined(MACOSX)
    #include <GLUT/glut.h>
#else
    #include <GL/glut.h>
#endif

// CL/GL includes and defines
#include <CL/cl_gl.h>

// Uncomment this #define to enable CL/GL Interop
//#define GL_INTEROP

// constants / global variables
//*********************************************************************
********

// GL
int iGLUTWindowHandle;                      // handle to the GLUT
window
int iGLUTMenuHandle;                        // handle to the GLUT menu
int iGraphicsWinWidth = 512;                // GL Window width
int iGraphicsWinHeight = 512;               // GL Window height
cl_int image_width = iGraphicsWinWidth;     // teapot image width
cl_int image_height = iGraphicsWinHeight;   // teapot image height
GLuint tex_screen;                          // (offscreen) render
target
float rotate[3];                            // var for teapot view
rotation

// pbo variables
GLuint pbo_source;
GLuint pbo_dest;
unsigned int size_tex_data;
unsigned int num_texels;
unsigned int num_values;

// CL objects
cl_context cxGPUContext;
cl_command_queue cqCommandQue;
cl_device_id device;
cl_program cpProgram;
cl_kernel ckKernel;
cl_mem cl_pbos[2] = {0,0};
cl_int ciErrNum;
const char* clSourcefile = "postprocessGL.cl";
```

```
// Timer and fps vars
int iFrameCount = 0;                // FPS count for averaging
int iFrameTrigger = 90;             // FPS trigger for sampling
int iFramesPerSec = 0;              // frames per second
int iTestSets = 3;                  // # of loop set retriggers before
auto exit when bNoPrompt = shrTrue

// app configuration parms
const char* cProcessor [] = {"OpenCL GPU", "Host C++ CPU"};
int iProcFlag = 0;                  // 0 = GPU, 1 = CPU
shrBOOL bNoPrompt = shrFALSE;       // false = normal GL loop, true =
Finite period of GL loop (a few seconds)
shrBOOL bQATest = shrFALSE;               // false = normal GL loop,
true = run No-GL test sequence
bool bPostprocess = shrTRUE;        // true = run blur filter
processing on GPU or host, false = just do display of old data
bool bAnimate = true;               // true = continue incrementing
rotation of view with GL, false = stop rotation
int blur_radius = 8;                // radius of 2D convolution
performed in post processing step

// Forward Function declarations
//*********************************************************************
********
// OpenCL functionality
int initCL(int argc, const char** argv);
void renderScene();
void displayImage();
void processImage();
void postprocessHost(unsigned int* g_data, unsigned int* g_odata, int
imgw, int imgh, int tilew, int radius, float threshold, float
highlight);

// GL functionality
bool InitGL(int argc, const char** argv);
void createPBO(GLuint* pbo);
void deletePBO(GLuint* pbo);
void createTexture(GLuint* tex_name, unsigned int size_x, unsigned int
size_y);
void deleteTexture(GLuint* tex);
void dumpImage();
void DisplayGL();
void idle();
void KeyboardGL(unsigned char key, int x, int y);
void Reshape(int w, int h);
void mainMenu(int i);

// Helpers
void Cleanup(int iExitCode);
void (*pCleanup)(int) = &Cleanup;
void TestNoGL();
void TriggerFPSUpdate();

// Main Program
```

```
//**********************************************************************
********
int main(int argc, const char** argv)
{
    // start logs
    shrSetLogFileName ("oclPostProcessGL.txt");
    shrLog(LOGBOTH, 0.0, "%s Starting...\n\n", argv[0]);

    // process command line arguments
    if (argc > 1)
    {
        bQATest = shrCheckCmdLineFlag(argc, argv, "qatest");
        bNoPrompt = shrCheckCmdLineFlag(argc, argv, "noprompt");
    }

    // init GL
    if(!bQATest)
    {
        InitGL(argc, argv);
    }

    // init CL
    if( initCL(argc, argv) != 0 )
    {
        return -1;
    }

    // init fps timer
    shrDeltaT (1);

    // Create buffers and textures,
    // and then start main GLUT rendering loop for processing and
rendering,
      // or otherwise run No-GL Q/A test sequence
    if(!bQATest)
    {
        // create pbo
        createPBO(&pbo_source);
        createPBO(&pbo_dest);

        // create texture for blitting onto the screen
        createTexture(&tex_screen, image_width, image_height);

        glutMainLoop();
    }
    else
    {
        TestNoGL();
    }

    // Normally unused return path
    Cleanup(EXIT_FAILURE);
}

//**********************************************************************
********
```

```
void dumpImage()
{
    unsigned char* h_dump = (unsigned char*) malloc(sizeof(unsigned 
int) * image_height * image_width);

    clEnqueueReadBuffer(cqCommandQue, cl_pbos[1], CL_TRUE, 0, 
sizeof(unsigned int) * image_height * image_width, 
                        h_dump, 0, NULL, NULL);

    shrSavePPM4ub( "dump.ppm", h_dump, image_width, image_height);
    free(h_dump);
}

//**********************************************************************
********
void displayImage()
{
    // render a screen sized quad
    glDisable(GL_DEPTH_TEST);
    glDisable(GL_LIGHTING);
    glEnable(GL_TEXTURE_2D);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);

    glMatrixMode( GL_MODELVIEW);
    glLoadIdentity();

    glViewport(0, 0, iGraphicsWinWidth, iGraphicsWinHeight);

/*    glBegin(GL_POLYGON);

      glColor3d(1, 0, 0);

      float x = 0, y = 0;

      glVertex3d(-0.44+x, 0.6+y, 0);
      glVertex3d(-1.85+x, 0.6+y, 0);
      glVertex3d(-0.71+x, -0.24+y, 0);
      glVertex3d(-1.14+x, -1.58+y, 0);
      glVertex3d(0+x, -0.76+y, 0);
      glVertex3d(1.13+x, -1.58+y, 0);
      glVertex3d(0.70+x, -0.24+y, 0);
      glVertex3d(1.85+x, 0.6+y, 0);
      glVertex3d(0.43+x, 0.6+y, 0);
      glVertex3d(0+x, 1.94+y, 0);
*/

    glBegin(GL_QUADS);

    glTexCoord2f(0.0, 0.0);
    glVertex3f(-1.0, -1.0, 0.5);
    glTexCoord2f(1.0, 0.0);
    glVertex3f(1.0, -1.0, 0.5);

    glTexCoord2f(1.0, 1.0);
    glVertex3f(1.0, 1.0, 0.5);

    glTexCoord2f(0.0, 1.0);
    glVertex3f(-1.0, 1.0, 0.5);

    glEnd();

    glMatrixMode(GL_PROJECTION);
    glPopMatrix();

    glDisable(GL_TEXTURE_2D);
    glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB, 0);
    glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, 0);
}

// Display callback
//*********************************************************************
********
void DisplayGL()
{
    // Render the 3D teapot with GL
    renderScene();

    // start timer 0 if it's update time
    double dProcessingTime = 0.0;
    if (iFrameCount >= iFrameTrigger)
    {
        shrDeltaT(0);
    }

    // process
    processImage();

    // get processing time from timer 0, if it's update time
    if (iFrameCount >= iFrameTrigger)
    {
        dProcessingTime = shrDeltaT(0);
    }

    // flip backbuffer to screen
    displayImage();
    glutSwapBuffers();
    glutPostRedisplay();

    // Increment the frame counter, and do fps and Q/A stuff if it's
time
    if (iFrameCount++ > iFrameTrigger)
    {
#ifdef GPU_PROFILING
        // set GLUT Window Title
        char cFPS[256];
        iFramesPerSec = (int)((double)iFrameCount/shrDeltaT(1));
        if( bPostprocess )
        {
            #ifdef _WIN32
                sprintf_s(cFPS, 256, "%s Postprocessing ON | %u x %u | 
%i fps | Proc. t = %.4f s", 
                                      cProcessor[iProcFlag], 
iGraphicsWinWidth, iGraphicsWinHeight, iFramesPerSec, dProcessingTime);
            #else
                sprintf(cFPS, "%s Postprocessing ON | %u x %u | %i fps 
| Proc. t = %.4f s", 
                               cProcessor[iProcFlag], 
iGraphicsWinWidth, iGraphicsWinHeight, iFramesPerSec, dProcessingTime);
            #endif
            glutSetWindowTitle(cFPS);
        }
        else
        {
            #ifdef _WIN32
                sprintf_s(cFPS, 256, "Post Processing OFF | %u x %u | 
%i fps",  iGraphicsWinWidth, iGraphicsWinHeight, iFramesPerSec);
            #else
                sprintf(cFPS, "Post Processing OFF | %u x %u | %i fps", 
iGraphicsWinWidth, iGraphicsWinHeight, iFramesPerSec);
            #endif
            glutSetWindowTitle(cFPS);
        }

        // Log fps and processing info to console and file
        shrLog(LOGBOTH, 0.0, " %s\n", cFPS);
#endif

        // if doing quick test, exit
        if ((bNoPrompt) && (!--iTestSets))
        {
            // Cleanup up and quit
            Cleanup(EXIT_SUCCESS);
        }

        // reset framecount, trigger and timer
        iFrameCount = 0;
        iFrameTrigger = (iFramesPerSec > 1) ? iFramesPerSec * 2 : 1;
    }
}

//*********************************************************************
********
void idle()
{
    if (bAnimate) {
        rotate[0] += 0.2;
        rotate[1] += 0.6;
        rotate[2] += 1.0;
    }
    glutPostRedisplay();
}
```

```
// Keyboard events handler
//***********************************************************************
********
void KeyboardGL(unsigned char key, int x, int y)
{
    switch(key)
    {
        case 'P':   // P toggles Processing between CPU and GPU
        case 'p':   // p toggles Processing between CPU and GPU
            if (iProcFlag == 0)
            {
                iProcFlag = 1;
            }
            else
            {
                iProcFlag = 0;
            }
            shrLog(LOGBOTH, 0.0, "\n%s Processing...\n",
cProcessor[iProcFlag]);
            break;
        case ' ':   // space bar toggles processing on and off
            bPostprocess = !bPostprocess;
            shrLog(LOGBOTH, 0.0, "\nPostprocessing (Blur Filter)
Toggled %s...\n", bPostprocess ? "ON" : "OFF");
            break;
        case 'A':   // 'A' toggles animation (spinning of teacup)
on/off
        case 'a':   // 'a' toggles animation (spinning of teacup)
on/off
            bAnimate = !bAnimate;
            shrLog(LOGBOTH, 0.0, "\nGL Animation (Rotation) Toggled
%s...\n", bAnimate ? "ON" : "OFF");
            break;
        case '=':
        case '+':
            if (blur_radius < 16) blur_radius++;
            shrLog(LOGBOTH, 0.0, "\nBlur radius = %d\n", blur_radius);
            break;
        case '-':
        case '_':
            if (blur_radius > 1) blur_radius--;
            shrLog(LOGBOTH, 0.0, "\nBlur radius = %d\n", blur_radius);
            break;
        case '\033': // escape quits
        case '\015': // Enter quits
        case 'Q':    // Q quits
        case 'q':    // q (or escape) quits
            // Cleanup up and quit
            Cleanup(EXIT_SUCCESS);
    }

    // Trigger fps update and call for refresh
    TriggerFPSUpdate();
    glutPostRedisplay();
}
```

```
// Window resize handler callback
//**********************************************************************
********
void Reshape(int w, int h)
{
    iGraphicsWinWidth = w;
    iGraphicsWinHeight = h;

    glBindTexture(GL_TEXTURE_2D, tex_screen);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, w, h, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);

    image_width = w;
    image_height = h;

    shrLog(LOGBOTH, 0.0, " w = %d, h = %d\n", w,h);

    num_texels = image_width * image_height;
    num_values = num_texels * 4;
    size_tex_data = sizeof(GLubyte) * num_values;

    if( cl_pbos[0] != 0 ) {
      // update sizes of pixel buffer objects
      glBindBuffer(GL_ARRAY_BUFFER, pbo_source);
      glBufferData(GL_ARRAY_BUFFER, size_tex_data, NULL,
GL_DYNAMIC_DRAW);

      glBindBuffer(GL_ARRAY_BUFFER, pbo_dest);
      glBufferData(GL_ARRAY_BUFFER, size_tex_data, NULL,
GL_DYNAMIC_DRAW);

      glBindBuffer(GL_ARRAY_BUFFER,0);

      #ifndef GL_INTEROP
          // release current mem objects
          clReleaseMemObject(cl_pbos[0]);
          clReleaseMemObject(cl_pbos[1]);

          // create new objects for the current sizes
          cl_pbos[0] = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, 4
* image_width * image_height, NULL, 0);
          cl_pbos[1] = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY,
4 * image_width * image_height, NULL, 0);

          // update kernel arguments
          clSetKernelArg(ckKernel, 0, sizeof(cl_mem), (void *)
&(cl_pbos[0]));
          clSetKernelArg(ckKernel, 1, sizeof(cl_mem), (void *)
&(cl_pbos[1]));
      #endif

      clSetKernelArg(ckKernel, 2, sizeof(cl_int), &image_width);
      clSetKernelArg(ckKernel, 3, sizeof(cl_int), &image_height);
    }
}
```

```
//**********************************************************************
********
void mainMenu(int i)
{
    KeyboardGL((unsigned char) i, 0, 0);
}

//**********************************************************************
********
void deleteTexture(GLuint* tex)
{
    glDeleteTextures(1, tex);

    *tex = 0;
}

//**********************************************************************
********
void createTexture( GLuint* tex_name, unsigned int size_x, unsigned int
size_y)
{
    // create a texture
    glGenTextures(1, tex_name);
    glBindTexture(GL_TEXTURE_2D, *tex_name);

    // set basic parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

    // buffer data
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, size_x, size_y, 0, GL_RGBA,
GL_UNSIGNED_BYTE, NULL);
}

//**********************************************************************
********
void pboRegister()
{
    // Transfer ownership of buffer from GL to CL
    #ifdef GL_INTEROP
        clEnqueueAcquireGLObjects(cqCommandQue,2, cl_pbos, 0, NULL,
NULL);
    #else
        // Explicit Copy
        // map the PBO to copy data to the CL buffer via host
        glBindBufferARB(GL_PIXEL_PACK_BUFFER_ARB, pbo_source);

        GLubyte* ptr =
(GLubyte*)glMapBufferARB(GL_PIXEL_PACK_BUFFER_ARB,
                                             GL_READ_ONLY_ARB);
        clEnqueueWriteBuffer(cqCommandQue, cl_pbos[0], CL_TRUE, 0,
                            sizeof(unsigned int) * image_height *
image_width, ptr, 0, NULL, NULL);
        glUnmapBufferARB(GL_PIXEL_PACK_BUFFER_ARB);
        glBindBufferARB(GL_PIXEL_PACK_BUFFER_ARB, 0);
    #endif
}

//***************************************************************************
********
void pboUnregister()
{
        // Transfer ownership of buffer back from CL to GL
    #ifdef GL_INTEROP
        clEnqueueReleaseGLObjects(cqCommandQue,2, cl_pbos, 0, NULL,
NULL);
    #else
        // Explicit Copy
        // map the PBO to copy data from the CL buffer via host
        glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, pbo_dest);

        // map the buffer object into client's memory
        GLubyte* ptr =
(GLubyte*)glMapBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB,
                                            GL_WRITE_ONLY_ARB);
        clEnqueueReadBuffer(cqCommandQue, cl_pbos[1], CL_TRUE, 0,
                            sizeof(unsigned int) * image_height *
image_width, ptr, 0, NULL, NULL);
        glUnmapBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB);
        glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, 0);
    #endif
}

// Initialize GL
//***************************************************************************
********
bool InitGL(int argc, const char **argv )
{
    // init GLUT and GLUT window
    glutInit(&argc, (char**)argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_ALPHA | GLUT_DOUBLE |
GLUT_DEPTH);
    glutInitWindowPosition (glutGet(GLUT_SCREEN_WIDTH)/2 -
iGraphicsWinWidth/2,
                            glutGet(GLUT_SCREEN_HEIGHT)/2 -
iGraphicsWinHeight/2);
    glutInitWindowSize(iGraphicsWinWidth, iGraphicsWinHeight);
    iGLUTWindowHandle = glutCreateWindow("OpenCL/OpenGL post-
processing");

    // register GLUT callbacks
    glutDisplayFunc(DisplayGL);
    glutKeyboardFunc(KeyboardGL);
    glutReshapeFunc(Reshape);
    glutIdleFunc(idle);
    // create GLUT menu
    iGLUTMenuHandle = glutCreateMenu(mainMenu);
    glutAddMenuEntry("Toggle Post-processing (Blur filter) ON/OFF
<spacebar>", ' ');
    glutAddMenuEntry("Toggle Processor between GPU and CPU [p]", 'p');
    glutAddMenuEntry("Toggle GL animation (rotation) ON/OFF [a]", 'a');
    glutAddMenuEntry("Increment blur radius [+ or =]", '=');
    glutAddMenuEntry("Decrement blur radius [- or _]", '-');
    glutAddMenuEntry("Quit <esc>", '\033');
    glutAttachMenu(GLUT_RIGHT_BUTTON);

    // init GLEW
    glewInit();
    GLboolean bGLEW = glewIsSupported("GL_VERSION_2_0
GL_ARB_pixel_buffer_object");
    shrCheckErrorEX(bGLEW, shrTRUE, pCleanup);

    // default initialization
    glClearColor(0.5, 0.5, 0.5, 1.0);
    glDisable(GL_DEPTH_TEST);

    // viewport
    glViewport(0, 0, iGraphicsWinWidth, iGraphicsWinHeight);

    // projection
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, (GLfloat)iGraphicsWinWidth / (GLfloat)
iGraphicsWinHeight, 0.1, 10.0);
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    glEnable(GL_LIGHT0);
    float red[] = { 0.1, 1.0, 0.1, 1.0 };
    float white[] = { 1.0, 1.0, 1.0, 1.0 };
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, red);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, white);
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 60.0);

    return true;
}

// Create PBO
//*********************************************************************
********
void createPBO(GLuint* pbo)
{
    // set up data parameter
    num_texels = image_width * image_height;
    num_values = num_texels * 4;
    size_tex_data = sizeof(GLubyte) * num_values;

    // create buffer object
    glGenBuffers(1, pbo);
    glBindBuffer(GL_ARRAY_BUFFER, *pbo);

    // buffer data
    glBufferData(GL_ARRAY_BUFFER, size_tex_data, NULL,
GL_DYNAMIC_DRAW);

    glBindBuffer(GL_ARRAY_BUFFER, 0);
}

// Delete PBO
//**********************************************************************
********
void deletePBO(GLuint* pbo)
{
    glBindBuffer(GL_ARRAY_BUFFER, *pbo);
    glDeleteBuffers(1, pbo);

    *pbo = 0;
}

// render a simple 3D scene
//**********************************************************************
********
void renderScene()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, (GLfloat)iGraphicsWinWidth / (GLfloat)
iGraphicsWinHeight, 0.1, 10.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -3.0);
    glRotatef(rotate[0], 1.0, 0.0, 0.0);
    glRotatef(rotate[1], 0.0, 1.0, 0.0);
    glRotatef(rotate[2], 0.0, 0.0, 1.0);

    glViewport(0, 0, iGraphicsWinWidth, iGraphicsWinHeight);

    glEnable(GL_LIGHTING);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

      //glutSolidDodecahedron();
      //glutSolidOctahedron();
      //glutWireSphere(1.0,5,8);
      //glutSolidTorus(0.5, 1.0, 1000, 8);
      //glutSolidIcosahedron();
    glutSolidTeapot(1.0);
}

// Init OpenCL
//**********************************************************************
********
int initCL(int argc, const char** argv)
{
    // Create the OpenCL context on a GPU device
    cxGPUContext = clCreateContextFromType(0, CL_DEVICE_TYPE_GPU, NULL,
NULL, &ciErrNum);
    shrCheckErrorEX(ciErrNum, CL_SUCCESS, pCleanup);

    // get and log the device info
    if( shrCheckCmdLineFlag(argc, (const char**)argv, "device") ) {
      int device_nr = 0;
      shrGetCmdLineArgumenti(argc, (const char**)argv, "device",
&device_nr);
      device = oclGetDev(cxGPUContext, device_nr);
    } else {
      device = oclGetMaxFlopsDev(cxGPUContext);
    }
    oclPrintDevName(LOGBOTH, device);

    // create a command-queue
    cqCommandQue = clCreateCommandQueue(cxGPUContext, device, 0,
&ciErrNum);
    shrCheckErrorEX(ciErrNum, CL_SUCCESS, pCleanup);

    // Memory Setup
    #ifdef GL_INTEROP
        cl_pbos[0] = clCreateFromGLBuffer(cxGPUContext,
CL_MEM_READ_ONLY, pbo_source, &ciErrNum);
        cl_pbos[1]= clCreateFromGLBuffer(cxGPUContext,
CL_MEM_WRITE_ONLY, pbo_dest, &ciErrNum);
    #else
        cl_pbos[0] = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, 4 *
image_width * image_height, NULL, &ciErrNum);
        cl_pbos[1] = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY, 4
* image_width * image_height, NULL, &ciErrNum);
    #endif
    shrCheckErrorEX(ciErrNum, CL_SUCCESS, pCleanup);

    // Program Setup
    size_t program_length;
    const char* source_path = shrFindFilePath(clSourcefile, argv[0]);
    char *source = oclLoadProgSource(source_path, "", &program_length);
    shrCheckErrorEX(source != NULL, shrTRUE, pCleanup);

    // create the program
    cpProgram = clCreateProgramWithSource(cxGPUContext, 1,(const char
**) &source, &program_length, &ciErrNum);
    shrCheckErrorEX(ciErrNum, CL_SUCCESS, pCleanup);
    free(source);

    // build the program
    ciErrNum = clBuildProgram(cpProgram, 0, NULL, NULL, NULL, NULL);
    if (ciErrNum != CL_SUCCESS)
    {
        // write out standard error, Build Log and PTX, then cleanup
and exit
        shrLog(LOGBOTH | ERRORMSG, (double)ciErrNum, STDERROR);
        oclLogBuildInfo(cpProgram, oclGetFirstDev(cxGPUContext));
        oclLogPtx(cpProgram, oclGetFirstDev(cxGPUContext),
"oclPostProcessGL.ptx");
```

```
        Cleanup(EXIT_FAILURE);
    }

    // create the kernel
    ckKernel = clCreateKernel(cpProgram, "postprocess", &ciErrNum);

    // set the args values
    ciErrNum |= clSetKernelArg(ckKernel, 0, sizeof(cl_mem), (void *)
&(cl_pbos[0]));
    ciErrNum |= clSetKernelArg(ckKernel, 1, sizeof(cl_mem), (void *)
&(cl_pbos[1]));
    ciErrNum |= clSetKernelArg(ckKernel, 2, sizeof(image_width),
&image_width);
    ciErrNum |= clSetKernelArg(ckKernel, 3, sizeof(image_width),
&image_height);
    shrCheckErrorEX(ciErrNum, CL_SUCCESS, pCleanup);

    return 0;
}

// Kernel function
//**********************************************************************
********
int executeKernel(cl_int radius)
{

    // set global and local work item dimensions
    size_t szGlobalWorkSize[2];
    size_t szLocalWorkSize[2];
    szLocalWorkSize[0] = 16;
    szLocalWorkSize[1] = 16;
    szGlobalWorkSize[0] = shrRoundUp((int)szLocalWorkSize[0],
image_width);
    szGlobalWorkSize[1] = shrRoundUp((int)szLocalWorkSize[1],
image_height);

    // set the args values
    cl_int tilew =  (cl_int)szLocalWorkSize[0]+(2*radius);
    ciErrNum = clSetKernelArg(ckKernel, 4, sizeof(tilew), &tilew);
    ciErrNum |= clSetKernelArg(ckKernel, 5, sizeof(radius), &radius);
    cl_float threshold = 0.8f;
    ciErrNum |= clSetKernelArg(ckKernel, 6, sizeof(threshold),
&threshold);
    cl_float highlight = 4.0f;
    ciErrNum |= clSetKernelArg(ckKernel, 7, sizeof(highlight),
&highlight);

    // Local memory
    ciErrNum |= clSetKernelArg(ckKernel, 8,
(szLocalWorkSize[0]+(2*16))*(szLocalWorkSize[1]+(2*16))*sizeof(int),
NULL);

    // launch computation kernel
    ciErrNum |= clEnqueueNDRangeKernel(cqCommandQue, ckKernel, 2, NULL,
                                      szGlobalWorkSize,
szLocalWorkSize,
                                      0, NULL, NULL);
    shrCheckErrorEX(ciErrNum, CL_SUCCESS, pCleanup);

    return 0;
}

// copy image and process using OpenCL
//**********************************************************************
********
void processImage()
{
    // activate destination buffer
    glBindBuffer(GL_PIXEL_PACK_BUFFER_ARB, pbo_source);

    //// read data into pbo. note: use BGRA format for optimal
performance
    glReadPixels(0, 0, image_width, image_height, GL_BGRA,
GL_UNSIGNED_BYTE, NULL);

    if (bPostprocess)
    {
        if (iProcFlag == 0)
        {
            pboRegister();
            executeKernel(blur_radius);
            pboUnregister();
        }
        else
        {
            // map the PBOs
            glBindBufferARB(GL_PIXEL_PACK_BUFFER_ARB, pbo_source);
            glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, pbo_dest);

            unsigned int* source_ptr = (unsigned
int*)glMapBufferARB(GL_PIXEL_PACK_BUFFER_ARB,

GL_READ_ONLY_ARB);

            unsigned int* dest_ptr = (unsigned
int*)glMapBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB,

GL_WRITE_ONLY_ARB);

            // Postprocessing on the CPU
            postprocessHost(source_ptr, dest_ptr, image_width,
image_height, 0, blur_radius, 0.8f, 4.0f);

            // umap the PBOs
            glUnmapBufferARB(GL_PIXEL_PACK_BUFFER_ARB);
            glBindBufferARB(GL_PIXEL_PACK_BUFFER_ARB, 0);
            glUnmapBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB);
            glBindBufferARB(GL_PIXEL_UNPACK_BUFFER_ARB, 0);
        }

        // download texture from PBO
        glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, pbo_dest);
        glBindTexture(GL_TEXTURE_2D, tex_screen);
        glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0,
                        image_width, image_height,
                        GL_BGRA, GL_UNSIGNED_BYTE, NULL);

    }
    else
    {
        // download texture from PBO
        glBindBuffer(GL_PIXEL_UNPACK_BUFFER_ARB, pbo_source);
        glBindTexture(GL_TEXTURE_2D, tex_screen);
        glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0,
                        image_width, image_height,
                        GL_BGRA, GL_UNSIGNED_BYTE, NULL);

    }
}

// Helper to trigger reset of fps vars at transition
//*********************************************************************
********
void TriggerFPSUpdate()
{
    iFrameCount = 0;
    shrDeltaT(1);
    iFramesPerSec = 1;
    iFrameTrigger = 2;
}

// Run a test sequence without any GL
//*********************************************************************
********
void TestNoGL()
{
    // execute OpenCL kernel without GL interaction
    cl_pbos[0] = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, 4 *
image_width * image_height, NULL, &ciErrNum);
    cl_pbos[1] = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY, 4 *
image_width * image_height, NULL, &ciErrNum);

    // set the args values
    ciErrNum |= clSetKernelArg(ckKernel, 0, sizeof(cl_mem), (void *)
&(cl_pbos[0]));
    ciErrNum |= clSetKernelArg(ckKernel, 1, sizeof(cl_mem), (void *)
&(cl_pbos[1]));
    ciErrNum |= clSetKernelArg(ckKernel, 2, sizeof(image_width),
&image_width);
    ciErrNum |= clSetKernelArg(ckKernel, 3, sizeof(image_width),
&image_height);

    // warmup
    executeKernel(blur_radius);
    clFinish(cqCommandQue);

    // Measure Time
    shrDeltaT(0);
    executeKernel(blur_radius);

    clFinish(cqCommandQue);

    // Get elapsed time, and Log state and var data
    shrLog(LOGBOTH, 0.0, "\n%s Postprocess GL QA Test | %u x %u | Proc.
Time = %.4f s\n",
           cProcessor[iProcFlag], iGraphicsWinWidth,
iGraphicsWinHeight, shrDeltaT(0));

    // Cleanup and exit
    Cleanup(EXIT_SUCCESS);
}

// Function to clean up and exit
//*********************************************************************
********
void Cleanup(int iExitCode)
{
    // Cleanup allocated objects
    shrLog(LOGBOTH, 0.0, "\nStarting Cleanup...\n\n");
    if(pbo_source)deletePBO(&pbo_source);
    if(pbo_dest)deletePBO(&pbo_dest);
    if(tex_screen)deleteTexture(&tex_screen);
      if(ckKernel)clReleaseKernel(ckKernel);
    if(cpProgram)clReleaseProgram(cpProgram);
    if(cqCommandQue)clReleaseCommandQueue(cqCommandQue);
    if(cxGPUContext)clReleaseContext(cxGPUContext);
    if(cl_pbos[0])clReleaseMemObject(cl_pbos[0]);
    if(cl_pbos[1])clReleaseMemObject(cl_pbos[1]);
    if(iGLUTMenuHandle)glutDestroyMenu(iGLUTMenuHandle);
    if(iGLUTWindowHandle)glutDestroyWindow(iGLUTWindowHandle);
    shrLog(LOGBOTH, 0.0, "TEST %s...\n\n", iExitCode == 0 ? "PASSED" :
"FAILED !!!");

    // finalize logs and leave
    if ( bNoPrompt || bQATest)
    {
        shrLog(LOGBOTH | CLOSELOG, 0.0, "oclPostProcessGL.exe
Exiting...\n");
    }
    else
    {
        shrLog(LOGBOTH | CLOSELOG, 0.0, "oclPostProcessGL.exe
Exiting...\nPress <Enter> to Quit\n");
        #ifdef WIN32
            getchar();
        #endif
    }
    exit (iExitCode);
}
```

```
/*
 * Hostcode.cpp
 * Copyright 1993-2009 NVIDIA Corporation.  All rights reserved.
 *
 * NVIDIA Corporation and its licensors retain all intellectual
property and
 * proprietary rights in and to this software and related
documentation.
 * Any use, reproduction, disclosure, or distribution of this software
 * and related documentation without an express license agreement from
 * NVIDIA Corporation is strictly prohibited.
 *
 * Please refer to the applicable NVIDIA end user license agreement
(EULA)
 * associated with this source code for terms and conditions that
govern
 * your use of this NVIDIA software.
 *
 */

#include <math.h>
#include <oclUtils.h>

template<class T>
T clamp(T x, T a, T b)
{
    return MAX(a, MIN(b, x));
}


// convert floating point rgb color to 8-bit integer
unsigned int rgbToInt(float r, float g, float b)
{
  r = clamp(r, 0.0f, 255.0f);
  g = clamp(g, 0.0f, 255.0f);
  b = clamp(b, 0.0f, 255.0f);
  return (((unsigned int)b)<<16) + (((unsigned int)g)<<8) + ((unsigned
int)r);
}

// get pixel from 2D image, with clamping to border
unsigned int getPixel(unsigned int *data, int x, int y, int width, int
height)
{
    x = clamp(x, 0, width-1);
    y = clamp(y, 0, height-1);
    return data[y*width+x];
}

/*
    2D convolution
    - operates on 8-bit RGB data stored in 32-bit uint
    - assumes kernel radius is less than or equal to block size
    - not optimized for performance
     _____________
    |   :     :   |
    |_ _:_____:_ _|
    |   |     |   |
    |   |     |   |
    |_ _|_____|_ _|
  r |   :     :   |
    |___:_____:___|
      r    bw   r
    <----tilew---->
*/

void postprocessHost(unsigned int* g_data, unsigned int* g_odata, int
imgw, int imgh, int tilew, int radius, float threshold, float
highlight)
{

    for( int y=0; y<imgh; ++y ) {
        for( int x=0; x<imgw; ++x ) {

            // perform convolution
            float rsum = .0f;
            float gsum = 0.0f;
            float bsum = 0.0f;
            float samples = 0.0f;


            for(int iy=0; iy<=radius+radius+1; iy++) {
                for(int ix=0; ix<=radius+radius+1; ix++) {
                    int dx = ix - radius;
                    int dy = iy - radius;

                    unsigned int pixel = getPixel(g_data, x+dx, y+dy,
imgw, imgh);

                    // only sum pixels within disc-shaped kernel
                    float l = dx*dx + dy*dy;
                    if (l <= radius*radius) {
                        float r = (float)(pixel&0x0ff);
                        float g = (float)((pixel>>8)&0x0ff);
                        float b = (float)((pixel>>16)&0x0ff);

#if 1
                        // brighten highlights
                        float lum = (r + g + b) / (255*3);
                        if (lum > threshold) {
                            r *= highlight;
                            g *= highlight;
                            b *= highlight;
                        }
#endif
                        rsum += r;
                        gsum += g;
                        bsum += b;
                        samples += 1.0f;
                    }
                }
            }
            rsum /= samples;
            gsum /= samples;
            bsum /= samples;

            g_odata[y*imgw+x] = rgbToInt(rsum, gsum, bsum);
        }
    }
}




/*
 * postprocessGL.cl
 * Copyright 1993-2009 NVIDIA Corporation.  All rights reserved.
 *
 * NVIDIA Corporation and its licensors retain all intellectual
property and
 * proprietary rights in and to this software and related
documentation.
 * Any use, reproduction, disclosure, or distribution of this software
 * and related documentation without an express license agreement from
 * NVIDIA Corporation is strictly prohibited.
 *
 * Please refer to the applicable NVIDIA end user license agreement
(EULA)
 * associated with this source code for terms and conditions that
govern
 * your use of this NVIDIA software.
 *
 */

#define USE_LOCAL_MEM


// macros to make indexing shared memory easier
#define SMEM(X, Y) sdata[(Y)*tilew+(X)]

int iclamp(int x, int a, int b)
{
    return max(a, min(b, x));
}

// convert floating point rgb color to 8-bit integer
uint rgbToInt(float r, float g, float b)
{
  r = clamp(r, 0.0f, 255.0f);
  g = clamp(g, 0.0f, 255.0f);
  b = clamp(b, 0.0f, 255.0f);
  return (convert_uint(b)<<16) + (convert_uint(g)<<8) +
convert_uint(r);
}

// get pixel from 2D image, with clamping to border
uint getPixel(__global uint *data, int x, int y, int width, int height)
```

```
{
    x = iclamp(x, 0, width-1);
    y = iclamp(y, 0, height-1);
    return data[y*width+x];
}

/*
    2D convolution using local memory
    - operates on 8-bit RGB data stored in 32-bit uint
    - assumes kernel radius is less than or equal to block size
    - not optimized for performance
     _____________
    |   :     :   |
    |_ _:_____:_ _|
    |   |     |   |
    |   |     |   |
    |_ _|_____|_ _|
  r |   :     :   |
    |___:_____:___|
      r    bw   r
    <----tilew---->
*/

__kernel void postprocess(__global uint* g_data, __global uint*
g_odata, int imgw, int imgh, int tilew, int radius, float threshold,
float highlight, __local uint* sdata)
{
    const int tx = get_local_id(0);
    const int ty = get_local_id(1);
    const int bw = get_local_size(0);
    const int bh = get_local_size(1);
    const int x = get_global_id(0);
    const int y = get_global_id(1);

    if( x >= imgw || y >= imgh ) return;

#ifdef USE_LOCAL_MEM
    // copy tile to shared memory
    // center region
    SMEM(radius + tx, radius + ty) = getPixel(g_data, x, y, imgw,
imgh);

    // borders
    if (tx < radius) {
        // left
        SMEM(tx, radius + ty) = getPixel(g_data, x - radius, y, imgw,
imgh);
        // right
        SMEM(radius + bw + tx, radius + ty) = getPixel(g_data, x + bw,
y, imgw, imgh);
    }
    if (ty < radius) {
        // top
        SMEM(radius + tx, ty) = getPixel(g_data, x, y - radius, imgw,
imgh);
        // bottom
        SMEM(radius + tx, radius + bh + ty) = getPixel(g_data, x, y +
bh, imgw, imgh);
    }

    // load corners
    if ((tx < radius) && (ty < radius)) {
        // tl
        SMEM(tx, ty) = getPixel(g_data, x - radius, y - radius, imgw,
imgh);
        // bl
        SMEM(tx, radius + bh + ty) = getPixel(g_data, x - radius, y +
bh, imgw, imgh);
        // tr
        SMEM(radius + bw + tx, ty) = getPixel(g_data, x + bh, y -
radius, imgw, imgh);
        // br
        SMEM(radius + bw + tx, radius + bh + ty) = getPixel(g_data, x +
bw, y + bh, imgw, imgh);
    }

    // wait for loads to complete
    barrier(CLK_LOCAL_MEM_FENCE);
#endif

    // perform convolution
     float rsum = 0.0f;
     float gsum = 0.0f;
     float bsum = 0.0f;
     float samples = 0.0f;


     for(int iy=0; iy<=radius+radius+1; iy++) {
       for(int ix=0; ix<=radius+radius+1; ix++) {
       int dx = ix - radius;
       int dy = iy - radius;

#ifdef USE_LOCAL_MEM
       uint pixel = SMEM(radius+tx+dx, radius+ty+dy);
#else
       uint pixel = getPixel(g_data, x+dx, y+dy, imgw, imgh);
#endif

       // only sum pixels within disc-shaped kernel
       float l = dx*dx + dy*dy;
       if (l <= radius*radius) {
         float r = convert_float(pixel&0x0ff);
         float g = convert_float((pixel>>8)&0x0ff);
         float b = convert_float((pixel>>16)&0x0ff);
#if 1
         // brighten highlights
         float lum = (r + g + b) / (255*3);
         if (lum > threshold) {
           r *= highlight;
           g *= highlight;
           b *= highlight;
         }
#endif
         rsum += r;
         gsum += g;
         bsum += b;
         samples += 1.0f;
       }
       }
     }

     rsum /= samples;
     gsum /= samples;
     bsum /= samples;

    g_odata[y*imgw+x] = rgbToInt(rsum, gsum, bsum);
}
```