COMPUTATIONAL SPEED ON THE MULTIPROCESSOR ARCHITECTURE AND GPU

Peter Kvasnica[1], Igor Kvasnica[2]

[1]Alexander Dubcek University of Trencin, Faculty of Mechatronics,
Department of Informatics,
Študentská 2, 911 50 Trenčín, Slovak Republic
[2]Regional Department for Environment Issues of Trencin
Hviezdoslavova č. 3, 911 00 Trenčín, Slovak Republic

kvasnica@tnuni.sk (Peter Kvasnica)

Abstract

The article deals with the design of parallel computing in computer's systems of a simulator. The concept is based on computers that create a distributed computer system in the network and on the other side computer with support GPU. In both cases this information system is created by computers and the program applications of the mathematical models. The important part of this article describes time scheduling of simulation. The first time is scheduling time of the simulation processes running on the computer. The second time is interrupt latency for control signals in computer. The third time is latency over the network that means transfer n-bytes from one computer to the other one computer. It explains the benefits GPU computing with tasks, scheduling and parallel execution mathematical models of simulator. Mathematical modeling is the art that is able to transform a point from original application into theoretic area to mathematical formulations for numerical analysis. The significant part of this article describes the implementation of distributed mathematical model with computation implemented by single-processor architecture in network, the cluster computing and single-processor architecture with support GPU. Modeling processes of simulator computer with GPU in opposition to cluster's computers, create a time benefit.

Keywords: Computation resources, Parallel computing, Processor - CPU, Graphic processor unit – GPU, Mathematical model.

Presenting Author's biography

Peter Kvasnica. Is the deputy director of the Centre of Information Technologies at Alexander Dubček University of Trenčín. He has been involved in research on mathematical models and programming virtual reality applications. He graduated from University of Technology in Brno (VUT Brno), in the field of study: digital computers. He was awarded the degree of Doctor of Philosophy (PhD.) at M. R. Štefánik Military Academy of Aviation in Košice, in the specialization in computer science application. He has been involved in the development of special adapted mathematical models of objects from the point of view of programming and use of distributed computer system in flight simulators for real-time applications. He is interested in software tools for parallel programming MPI, Open MP and information about Open CL for creation GPU software application.

1 Introduction

These sophisticated models enable us to build systems that simulate complicated models. These systems require an enormous amount of computational resources. We typically satisfy these needs by parallelizing our computations across high performance computing. Definition parallel means that not only can they take advantage of the multiple cores on a CPU but that they can also take advantage of the tenths of cores on a GPU.
The GPU application runs in background and is able to start several threads with low priority to handle incoming requests. The user volunteering his computer is not bothered too much [1].
At its core parallel computing is using multiple computational resources in parallel to solve a computational problem. The goal of solving computational problems in parallel is to save time. Parallel computing is used in the areas such as mathematical simulation, graphics processing, and computation in the field of finance, data mining, seismology, mathematics and physics just to name a few [3].

A CPU (Central Processing Unit) functions via executing a sequence of instructions. These instructions reside in some sort of main memory. Typically go through four distinct phases during their CPU lifecycle: fetch, decode, execute, and writeback. During the *fetch* phase the instruction is retrieved from main memory and loaded onto the CPU. Once the instruction is fetched it is *decoded* or broken down into an opcode (operation to be performed). Once it is determined what operation needs to be performed the operation is *executed*. This may involve copying memory to locations specified in the instructions operands or having the ALU (arithmetic logic unit) perform a mathematical operation. The final phase is the *writeback* of the result to either main memory or a CPU register. When the writeback is completed, the entire process repeats. These four phases have some important impact on the computation speed.

To make their CPUs faster chip manufacturers started to create parallel execution paths in their CPUs by pipelining their instructions. Pipelining allows more than one step in the CPU lifecycle to be performed at any given time by breaking down the pathway into discrete stages. This separation can be compared to an assembly line, in which an instruction is made more complete at each stage until it exits the execution pipeline and it is returned [2].

The simulation of decomposited mathematical models can be created using parallel computer architecture. This parallel computer architecture can be based on the multiprocessors and nowadays especially on the multi core processors, resp. computing with support GPU and CUDA standard.

2 Distributed Mathematical Model

We get linear, manageable, dynamic system connected to the model [4]:

$$\begin{aligned} \dot{\mathbf{x}}(t) &= \mathbf{A}\mathbf{x}(t) + \mathbf{B}\mathbf{u}(t), \\ \mathbf{y}(t) &= \mathbf{C}\mathbf{x}(t), \\ \mathbf{x}(t) &= \mathbf{x}_0, \end{aligned} \tag{1}$$

where: **A** is an n x n matrix, **B** is an n x m matrix, **D** is an r x n matrix, **x** is an n x l matrix, **u** is an m x l matrix, **y** is an r x l matrix (columns).

When we try and make the task easier in the way that we will focus on the object of the control, the equation (1) can have a general shape:

$$\dot{\mathbf{x}}_i = \mathbf{A}\mathbf{x}_\mathbf{i} + \mathbf{B}u. \tag{2}$$

According to the given facts the equation (2) can be expressed [3]:

$$\begin{pmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} u. \tag{3}$$

Let us decompose the given system into four subsystems. The first one shall be the subsystem of state variable $\dot{x}_1$, the second $\dot{x}_2$ and the third $\dot{x}_3$ ones shall be a subsystem of performing items, noise and failures of the apparatus shall be measured by the fourth one $\dot{x}_4$.

The state space is divided into 4 parts :

$$\mathbf{x} = (x_1, x_2, x_3, x_4)^T \, \dot{\mathbf{x}} = (\dot{x}_{11}, \dot{x}_{22}, \dot{x}_{33}, \dot{x}_{44})^T, \tag{4}$$

where $\mathbf{x_{ij}}$ represent the items of a state vector derivation. The value *i* represents order in state vector, then *j* stands for sequential number of the item in the given subsystem. The architecture of the system matrix **A** in the state space after carrying out multiplication the following shape can be formed [5]:

$$\begin{aligned} \dot{x}_{11} &= a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + a_{14}x_4, \\ \dot{x}_{22} &= a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + a_{24}x_4, \\ &\vdots \\ \dot{x}_{44} &= a_{41}x_1 + a_{42}x_2 + a_{43}x_3 + a_{44}x_4. \end{aligned} \tag{5}$$

3 Computational speedup factor

To achieve speeds higher than scalar chip manufacturers started to embed multiple execution units in increasing their degree of parallelism. In a superscalar pipeline, multiple instructions are read and passed to a dispatcher, which decides whether or not the instructions can be executed in parallel. If they are dispatched to available execution units, resulting in the ability for several instructions to be executed simultaneously. In general, the more instructions a

superscalar CPU is able to dispatch simultaneously to waiting execution units, the more instructions will be completed in a given clock cycle. By using techniques like instruction pipelining and adding multiple execution units nowadays CPUs have significantly increased their degree of instruction parallelism, however, they still lag far behind GPUs as discussed below [2].

To compare heterogeneous CPU/GPU architectures in their speed, we need to use a different metric than their clock speed. The industry accepted measure is FLOPS (FLoating point Operations Per Second).

From the Fig 1 above we can see that Intel's Harpertown chips have about an 80 GFLOP rating while Nvidia's most powerful card shipping today offers just of a Terra FLOP worth of computational capacity. According to this chart the GPU is 10 times faster than the CPU [1].

Fig 1. Power response to Intel Core2 Duo and Harpertown versus Nvidia cards

A GPU is a special purpose processor, known as a stream processor, specifically designed to perform a very large number of floating point operations in parallel. These processors may be integrated on the motherboard or attached via a PCIExpress card. Modern GPUs typically contain several multi processors each containing many processing cores. Today's high end cards typically have gigabytes of dedicated memory and several hundred processors running thousands of threads all dedicated to performing floating point math [2].

As an example of power we compare the CPU model running on 2.4 GHz Intel Core2 Duo E6600 to a GPU model running on an Radeon X1950Pro card. As the chart clearly shows the GPU can perform this particular algorithm close to more 10x faster than the CPU. We're using a double precision algorithm for this comparison and the Radeon GPU is not very fast at double precision Math right now.

4 Potential Increased Computational Speed

With the advent of multi-core chips – from the traditional AMD and Intel multi-core to the more exotic hybrid multi-core of IBM Cell and many-core of AMD/ATi and NVIDIA graphics processing units (GPUs) – parallel computing across multiple cores on a single chip has become a necessity. However, parallel computing on a large-scale supercomputer is a challenging endeavor from the perspectives of ease of access, ease of programming, and cost.

A significantly more cost-effective solution is general-purpose computation on graphics processing units (GPU), also known as video cards. With the peak floating-point performance of a GPU now exceeding a teraflop (tera floating-point operations per second), the GPU delivers supercomputing in a small and economical package [6].

As mentioned in the section above, real-time Windows/Linux extensions are a convenient way to apply quick and deterministic response for interrupts on standard thread kernels. We decide for real-time application interface, which inserts an additional scheduler between the already existing one and the hardware layer.

General MPI program structure consists of three steps: initializing of MPI environment, scientific work with message passing calls and terminating MPI environment (see pseudo-code in Fig 2.).

```
MPI_Init()
...
MPI_Barrier()
//central node is 0
if (processor == 0){
  // data collect. + visualis.
} else {
   switch (processor)
   {
     case 1:
ModellingEquation1(); break;
     case 2: ModellingEquation2();
break;
     case 3: ModellingEquation3();
break;
     case 4: ModellingEquation4();
break;
   }
}
MPI_Finalize()
```

Fig 2. Parallel through network

The application program resides in the normal user space and calls *MPI_Init()* to start a message transfer over the synchronisation network. The application is then registered at real-time application interface and can further use Inter-Process-Communication (IPC) functionality. A signal is released for either a Sending-Thread (*SndThread*) in case of data transfer or a Trigger-Thread (*TrigThread*) in case of Trigger data. Receiving a message from the NIC works in the opposite way, whereby a signal is omitted by the Receiving Thread (*RcvThread*) each time a communication is completed [4].

4.1 Multi-core processor

The multi-core processor is a processing system composed of two or more independent cores. One can describe it as an integrated circuit to which two or more individual processors (called *cores* in this sense) have been attached. The cores are typically integrated into a single integrated circuit die (known as a chip multiprocessor or CMP), or they may be integrated onto multiple dies in a single chip package. A many-core processor is one in which the number of cores is large enough that traditional multi-processor techniques are no longer efficient — this threshold is somewhere in the range of several tens of cores — and probably requires a network on chip[10].

PC users are running multiple, intense software applications simultaneously and demanding more on hardware resources. In office, computer usage has changed from data entry and word processing to e-Commerce, online collaboration and an ever-increasing need for continual security and virus protection. In the home, interests have shifted from low-bandwidth photos and Internet surfing to downloading and viewing high definition videos as well as advanced photo and video editing. In science computers are used for parallel applications and simulation mathematical models.

4.2 Graphic processor unit

Our initial exploration of mapping Graphics Environment Manger (GEM) to the GPU was conducted through the high level abstraction of MS Visual 2008 SDK, also known as OpenCL, on a machine with a Radeon X1950Pro GPU running Windows XP with SP3. The OpenCL (Open Compute Language) is an open standard for parallel programming of heterogeneous systems, managed by the Khronos Group. OpenCL supports a wide range of applications, from embedded and consumer software to HPC solutions, through a low-level, high-performance, portable abstraction. By creating an efficient, close-to-the-metal programming interface, OpenCL will form the foundation layer of a parallel computing ecosystem of platform-independent tools, middleware and applications.
The calculation of mathematical models (e.g. aircraft) via an analytical approach implemented in source code described above has a very useful property of parallel across all the points of reference are calculated. In addition to this, each thread can be computed as a reduction, a sum to be specific, of a set of completely independent calculations on each model allowing for multiple dimensions of parallelism [6].
Such manner has efforts to exploit the thread for non-graphical applications such simulation of mathematical models. By using high-level shading languages such as DirectX, OpenGL, OpenCL and Brook+, various data parallel algorithms have been ported to such systems. Problems such as protein folding, stock options pricing, SQL queries, and MRI reconstruction achieved remarkable performance speedups on the computation system.

5 Speedup Factor in Concurent processes

An important influence on the simulation speed of mathematical models has interrupt and scheduling latency, and network latency in network simulaltion. All computers run a Windows operating system with a SystemResponsiveness extension that allows for smallest possible interrupt latencies. This key contains a REG_DWORD value named SystemResponsiveness that determines the percentage of CPU resources that should be guaranteed to low-priority tasks. For example, if this value is 20, then 20% of CPU resources are reserved for low-priority tasks. The framework designed provides functions for sending and receiving data packages as well as a user interface to program the trigger behaviour of the simulation.

Three different types of latencies can be distinguished in a computer simulation:

- **Network latency:** Time difference between sending a single byte and receiving it.
- **Interrupt latency:** Amount of time that elapses between the physical interrupt signal being asserted and the interrupt service routine running.
- **Scheduler latency:** Interval between a wakeup signaling that an event has occured and the operating system scheduler getting the opportunity to schedule the application that is waiting for the wakeup to occur.

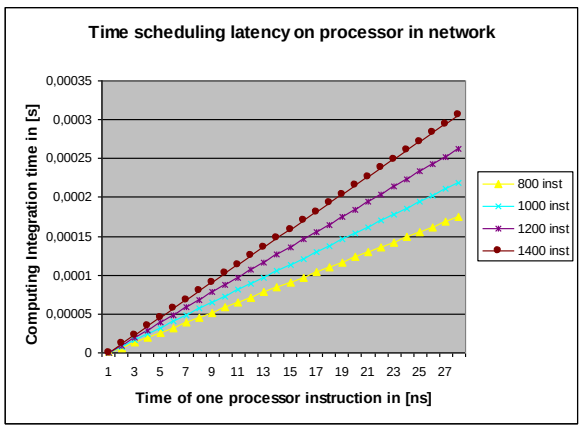

Fig 3. Time dependence scheduler latency, interrupt latency and network latency

5.1 Processes running at the Network (cluster)

Three different types of latencies can be distinguished in a computer simulation network:

- Network latency.
- Interrupt latency.

- Scheduler latency.

The interrupt latency component is much smaller than latencies caused by operating system schedulers. Therefore, the interrupt service routine, which reads data from the computer network and stores them into a buffer. Upon completion of a transmission of several bytes, the written buffer is read out by the user application. The amount of latency for a n-byte long message from one computer to another over the network is given by

$$T_S = T_{S,Sched} + T_{S,Int} + n * (T_{S,Netw}), \quad (6)$$

$T_{S,Sched}$ is the scheduler latency, $T_{S,Int}$ is the interrupt latency, and $T_{S,Netw}$ is the latency over the network for receiving n-byte [5].

The picture shows scheduling time of simulation process. The simulation process means scheduler latency consist from the different number of instruction, 800, 1000 and etc. These instructions represent numeric methods for solving differential equations. The equations describe the distributed mathematical model, rewriting in part 2.

5.2 Processes running at the multi-core processor

Two different types of latencies can be distinguished in a computer simulation:

- Interrupt latency.
- Scheduler latency.

The interrupt latency component is much smaller than latencies caused by operating system schedulers. Upon completion of a transmission of several bytes, the written buffer is read out by the user application. The amount of latency is given by

$$T_S = T_{S,Sched} + T_{S,Int} \quad (7)$$

$T_{S,Sched}$ is the scheduler latency, $T_{S,Int}$ is the interrupt latency, see Fig 4.

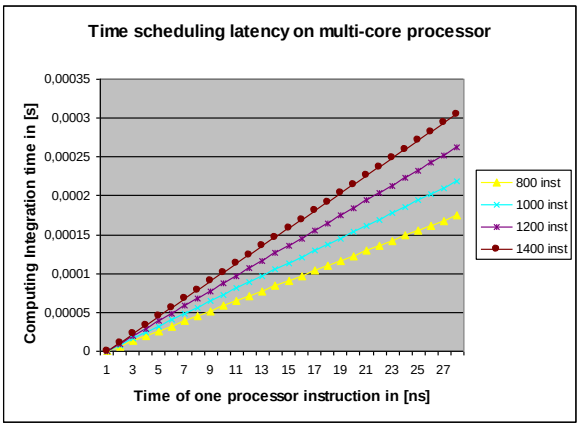

Fig 4. Time dependence scheduler and interrupt latency

The picture shows scheduling time of simulation process. The simulation process consists of the different number of instruction, 800, 1000 and etc., that define instruction of mathematical model. These instructions represent numeric methods of solving the distributed mathematical model.

5.3 Processes running at the GPU

The "processor" used in the X1950 Pro is based on an 8-vertex 36-pixel shader configuration. The pipeline configuration the X1950 Pro is based on the RV570 core is built with the X1950 Pro in mind. With the introduction of the X1950 Pro will be phased out. We would like to say that the X1950 Pro has the same core clock speed as the X1900 GT, but the issue is a little more complicated.

We are using the same Intel Core 2 E 6600 setup for this article that has been used for over past months. Driver revisions haven't changed since our X1950 Pro article, and we will be looking at the same resolution: 1280x1024 in our application. Here's the breakdown of the hardware used.

Tab. 1 Hardware parameters graphic cards [11]

	ATI X1900 GT	ATI X1950 Pro
Interface	PCI-E 16x	PCI-E 16x
RAMDAC	2 X.400 MHz	2 X.400 MHz
T&L	DirectX 9.0c	DirectX 9.0c
Pixels Pipelines	36	36
Vertex Pipelines	8	8
Embarked memory	256 Mo	256 Mo
Interface memory	256 bits	256 bits
Band-width	35,8 Go/S	44,1 Go/S
Frequency GPU	575 MHz	575 MHz
Frequency memory	600 MHz	690 MHz

Two different types of latencies can be distinguished in a computer simulation:

- Interrupt latency.
- Scheduler latency.

The interrupt latency component is much smaller than latencies caused by operating system schedulers. The simulation speed is 8x faster than the simulation on the CPU. Upon completion of a transmission of several bytes, the written buffer is read out by the user application. The amount of latency for a n-byte message from one machine to another over the network is given by

$$T_S = T_{S,Sched} + T_{S,In} + T_{S,Out} \quad (8)$$

$T_{L,Sched}$ is the scheduler latency, $T_{L,In}$ is the date interrupt latency for writing data to the GPU, and $T_{L,Out}$ is the date interrupt latency for reading data from the GPU. Time dependences are in next picture.

The picture shows scheduling time of simulation process. The simulation process consists of the different number of instruction, 800, 1000 and etc., that define instruction of mathematical model.

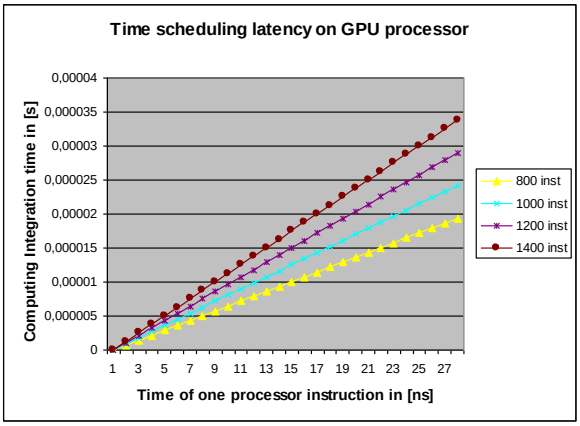

Fig 5. Time dependence scheduler and interrupt latency

Our results prove that there is no difference between simulation mathematical models by computers in network and simulation by multi-core processor. It's due to the fact that small data range are transferred between computers in network. The amount of data reaches the number of more than hundred bytes and time need for transfer in network 100 Mb/s is neglectable. Speed-up in simulation mathematical models by graphics processing unit is approximately by 8x higher than the one in two previous methods. It is due to the architecture of the scalar computing.

We will also be testing the same three simulation methods that have been employed for the past few projects. In future we will see an exciting change in our benchmarking lineup as we are preparing new applications or computing to be benchmarked. Expect to see the latest self on the art and their way into our test suite in the near future. For now, sit back, relax, and enjoy the soothing experience that is benchmark analysis.

Acknowledgement: **This work has been supported by the grant VEGA 1/0330/09**.

6 References

[1] Mengotti, T. : GPU, a framework for distributed computing over Gnutella, Master Thesis in Computer Science, ETH Zűrich, Switzerland, March 29, 2004.

[2] CUDA / OpenCL Computing: [online] http://www.openclcomputing.com/GPGPU101.mht.

[3] Confessions of a Speed Junkie [online] http://openclcomputing.com/index.php/component/content/article/5-general/2-confessions-of-a-speed-junkie.

[4] Clark, R., N.: *Control System Dynamics*, First Published, Cambirdge University Press, New York, USA, 1996.

[5] Driels, M.: *Linear Control Systems Engineering*, McGraw-Hill Inc., San Francisco, USA, 1996.

[6] Anandakrishnan, R.: Accelerating Electrostatic Surface Potential Calculation with Multiscale Approximation on Graphics Processing Units. Department of Computer Science, Virginia Tech 2050 Torgersen Hall (0106), Blacksburg, VA 24061 USA.

[7] Griesser, A, Luc Van Gool1: RTSyncNet - A _exible Real-Time Synchronisation Network for Cluster based Vision- and Graphics-Architectures. ftp://ftp.vision.ee.ethz.ch/publications/proceedings/eth_biwi_00336.pdf.

[8] NVIDIA's Next Generation CUDATM Compute Architecture:[online] http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[9] Huges, C., Huges, T.: *Parallel and Distributed Programming Using* C++. The Safari Press: Addison-Wesley Professional, 2003, ISBN 978-0-13-101376-6.

[10] Programming Strategies for Multicore Processing: Data Parallelism : [online] http://zone.ni.com/devzone/cda/tut/p/id/6421.

[11] ATI Radeon X1950 Pro full specification : [online]. http://xtreview.com/review154.htm.