PARALLEL COMPUTING IN ECONOMICS - AN OVERVIEW OF THE SOFTWARE FRAMEWORKS

Bogdan OANCEA*

Abstract

This paper discusses problems related to parallel computing applied in economics. It introduces the paradigms of parallel computing and emphasizes the new trends in this field - a combination between GPU computing, multicore computing and distributed computing. It also gives examples of problems arising from economics where these parallel methods can be used to speed up the computations.

Keywords: *Parallel Computing, GPU Computing, MPI, OpenMP, CUDA, computational economics.*

1. Introduction

Although parallel computers have existed for over 40 years and parallel computing offer a great advantage in terms of performance for a wide range of applications in different areas like engineering, physics, chemistry, biology, computer vision, in the economic research field they were very rarely used until recent years. Economists have been accustomed to only use desktop computers that have enough computing power for most of the problems encountered in economics because nowadays off-the-shelves desktops have FLOP rates greater than a supercomputer in late 80's.

The nature of parallel computing has changed since the first supercomputers in early '70s and new opportunities and challenges have appeared over the time including for economists. While 35 years ago computer scientists used massively parallel processors like the Goodyear MPP (Batcher, 1980), Connection Machine (Tucker & Robertson, 1988), Ultracomputer (Gottlieb et al., 1982), machines using Transputers (Baron, 1978) or dedicated parallel vector computers, like Cray computer series, now we are used to work on cluster of workstations with multicore processors and GPU cards that allows general programming. If Cray 1 had a peak performance of 80 Megaflops and CRAY X-MP had a peak performance of 200 MFLOPS a multicore Intel processor like I7-990X has more than 90 GFLOPS, Intel i7-3960X has a theoretical peak performance of 158 GFLOPs and Intel Xeon Phi reaches 1200 GFLOP for double precision operations (Intel, 2014).

One way of improving the processors speed and complexity was to increase the clock frequency and the number of transistors. The clock frequency has increased by almost four orders of magnitudes between the first 8086/8088 Intel processor used in a PC and actual Intel processors. The number of transistors also rose from 29.000 in Intel 8086 to approximately 730 million for an Intel Core i7-920 processor or 5000 million for a 62-Core Xeon Phi processor (INTEL, 2014). More transistors on a chip means that an operation can be executed in fewer cycles and a higher clock frequency means that more operations can be executed in a time unit. The increased clock frequency has an important side effect, namely the increased heat dissipated by processors. To overcome this problem, instead of increasing the clock

* Professor, PhD, Faculty of Economic Studies, Nicolae Titulescu University of Bucharest (e-mail: bogdanoancea@univnt.ro).

frequency, the processor designers come with a new model – multicore processors. Major microprocessors' producers like INTEL and AMD both offer multicore processors that are now common for desktop computers. Multicore processors turn normal desktops into truly parallel computers. These facts emphasize a major shift in processors and computer design: further improvements in performance will add multiple cores on the same chip, and many processors that share the same memory. Having high performance multicore processors rise a new problem: how to write the software to fully exploit the capabilities of the processors in a manner that the complexity of the underlying software not to be exposed to the programmer.

As graphical cards has become more and more complicated to support the requirements of the entertainment industry, they were developed as many-core multiprocessors that can perform identical operations on different data (implementing the SIMD model of parallel computing). In 2003, Mark Harris (Harris, 2003) recognized the potential of using graphical processing units (GPU) for general purpose applications. GPUs are high performance many-core processors that can achieve very high FLOP rates. Since the first time GPU was used for general purpose computing, GPU programming models have evolved and there are several approaches to GPU programming now: CUDA (Compute Unified Device Architecture) from NVIDIA and APP (Stream) from AMD. A great number of applications were ported to use the GPU and they obtain speedups of few orders of magnitude comparing to optimized multicore CPU implementations. We can mention here molecular dynamics applications (Stone, 2007) or bioinformatics (Charalambous, 2005).

GPGPU (general-purpose computing on graphics processing units) is used nowadays to speed up parts of applications that require intensive numerical computations. Traditionally, these parts of applications are handled by the CPUs but GPUs have now MFLOPs rates much better than CPUs (NVIDIA, 2014). The reason why GPUs have floating point operations rates much better even than multicore CPUs is that the GPUs are specialized for highly parallel intensive computations and they are designed with much more transistors allocated to data processing rather than flow control or data caching as is the case for multicore processors (NVIDIA, 2014).

As network technology advanced, the ability to connect computers together has become easier. This leads to another development in parallel computing - the appearance of grids and clusters of computers. In such environments, processors access the local memory and send messages (data) between them making a number of connected desktops a truly shared memory computer.

2. Software frameworks for developing parallel applications

The software tools for developing numerical intensive high performance applications depend on the parallelism that the users want to exploit. We will present four approaches to obtain high performance:

- Optimizing the serial programs, without taking into account multiple cores/processors;
- Using multiple cores in a shared memory environment;
- Using grids/clusters or clusters of computers, i.e. programming a distributed memory computer;
- Using GPU for general purpose computations;

A fifth approach combines distributed processing in a grid/cluster of computers with local GPU processing.

The first approach tries to obtain the maximum performance without using more than one core or CPU. Carefully writing the code can bring huge benefits yielding generous

returns. Code optimizations should consider using the cache memory as much as it can or avoid unnecessary branches in programs. (Oancea, 2011b) shows few techniques that can greatly improve the performance of numerical intensive programs:

Loop Unrolling: replace the body of a loop with several copies, such that the body of the new loop executes the exactly the same instructions as the initial loop. This will reduce the proportion of execution time spent on evaluating the control instruction.

Loop Unfolding: remove a number of the first iterations of a loop and place them before the main body. This will allow the earlier iterations of the loop to execute without requiring the processor to follow jump instructions back to the beginning of the loop, improving the ability of the code to be pipelined.

Loop Invariant Code Motion: move out of the loop the code within the loop that does not change on each iteration of the loop.

Other Optimizations: inline expansion of subroutines, strength reduction, factoring out of invariants.

The hot spots of a program could be identified using performance analyzer tools (profilers) and the effort of optimizing the code should be focused on these parts of the program. This could greatly improve the performance of a numerically intensive program (Oancea, 2011b).

The shared memory environment supposes that many processors can access the same memory on an equal basis. Nowadays every PC is a shared memory parallel computer since the CPU has multiple cores. One of the most used software framework to develop parallel programs for such architecture is the **OpenMP** library available for C/C++ or Fortran programming languages (www.openmp.org).

The OpenMP facilities can be accessed in three ways: by compiler directives (that appears as comments for compilers that do not support OpenMP), by library routines or by environment variables (Creel, 2008). OpenMP uses a "fork-join" model where a master thread starts and executes serially until reaches a directive that branches execution into many parallel threads (fork) that eventually are collapsed back (joined) into the master thread (Kiessling, 2009).

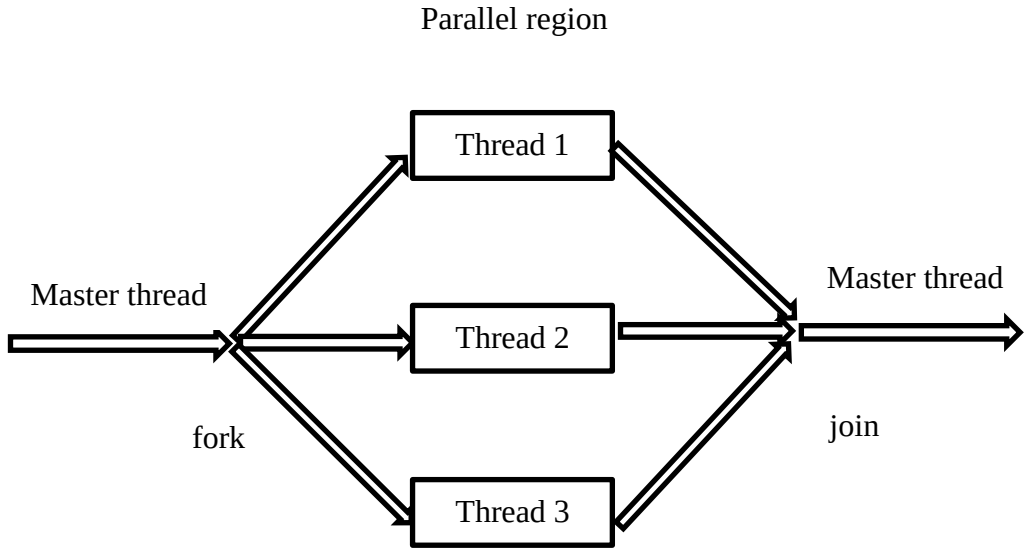

Figure 1. The execution path of OpenMP programs

A basic C program that uses OpenMP looks like this:

```
#include <stdio.h>
#include <omp.h>
int main(void) {
  #pragma omp parallel
    printf("Hello world!\n");

  return 0;
}
```

Figure 2. A basic C code that uses OpenMP

#pragma omp parallel specifies that the following code should be executed in parallel. On a 4-core processor the program output could look like:

```
Hello world!
Hello world!
Hello world!
Hello world!
```

Figure 3. The output of the C code

The general structure of a simple OpenMP program looks like this:

```
#include <omp.h>
main () {
        int nthreads, thread_id;
        printf("Here the code is executed serially");

      /* Fork a number of threads, each thread having a private thread_id variable */

        #pragma omp parallel private(thread_id){
        /* Get and print the thread ID. This portion is executed in parallel by each thread */
                thread_id = omp_get_thread_num();
                printf("Hello from thread = %d\n", thread_id);
                /* Only master thread enter here */
                if (thread_id == 0) {
                        nthreads = omp_get_num_threads();
                        printf("Number of threads = %d\n", nthreads);
                }
        }  /*All threads join the master thread */
        printf("Here the code is executed serially");
}
```

Figure 3. The general structure of an OpenMP code

The program starts in a serial manner and declares a variable thread_id.
#pragma omp parallel private(thread_id)forks a number of threads, each having a private thread_id variable. Inside the parallel portion on code, each thread receives its id by calling omp_get_thread_num()and prints it on the standard output. The master thread (having id=0) calls omp_get_num_threads()which returns the number of threads. After printing the number of threads, all threads are joined and the execution continues serially to the end of the

program. In this code snippet we exemplified how to use private variables that are available within each thread but OpenMP allows declaring shared variables too, that are available for all threads. When shared variables are used the same memory location is accessed by all threads. This poses some problems when many threads try to modify the same shared variable and special care must be taken by programmers.

Below is a list of most used OpenMP C functions (OpenMP ARB, 2011):

int omp_get_num_threads(void);
- Returns the number of threads in the parallel region when it is called;

void omp_set_num_threads(int num_threads);
- Sets the number of threds for the parallel region that will follow;

int omp_get_max_threads(void);
- Returns the maximum number of threads;

int omp_get_thread_num(void);
- Returns the current thread id;

int omp_get_num_procs(void);
- Returns the number of available processors;

int omp_in_parallel(void);
- Returns true if the call to the routine is enclosed by an active parallel region, false otherwise;

Programming distributed memory computer takes another approach – the message passing paradigm. In this model a processor can access its own local memory but when it want to communicate with other processors it send a message. The most widely used library for message passing programming is MPI. MPI provides portability, flexibility and efficiency. In fact MPI is a standard that has many implementations: MPICH (Gropp, 1996), OpenMPI (Open MPI: Open Source High Performance Computing, 2014), LAM/MPI (LAM/MPI Parallel Computing, 2014). In is available as a library for C or Fortran but many software packages like R, Matlab, Octave or Ox have extensions that allows use of MPI.

A number of processes that communicates between them are called a communicator and each process is identified by a “rank” (or ID).

The general structure of an MPI program is depicted in figure 4.

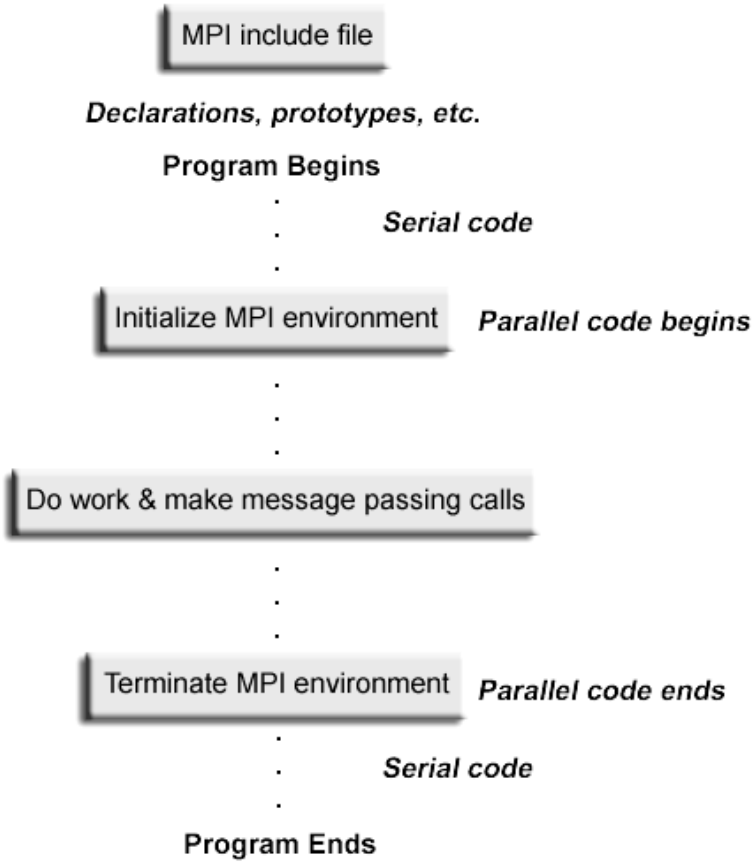

Figure 4. The structure of a MPI program

Image source: https://computing.llnl.gov/tutorials/mpi/

The program begins with a serial code, and then the MPI environment is initialized. After initialization follows the parallel code and in the end the MPI environment is terminated and some serial code could follow. Below are some of the most used MPI functions:

MPI_Init (&argc,&argv)
- Initializes the MPI environment.

MPI_Comm_size (comm,&size)
- This function determines how many processes are in a communicator.

MPI_Comm_rank (comm,&rank)
- Gives the rank of the process that called the function.

MPI_Send (&buf,count,datatype,dest,tag,comm)
- Sends a message to a given rank (process).

MPI_Recv (&buf,count,datatype,source,tag,comm,&status)
- Receives a message from a given rank (process).

MPI_Bcast (&buffer,count,datatype,root,comm)
- Sends a message from the root process (the one with rank=0) to all the other ranks (processes) in the communicator.

MPI_Finalize ()

- Terminates the MPI environment.

A simple MPI program that follows a master-slave paradigm is shown in figure 5.

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[]) {
        int numtasks, rank, r;
        r = MPI_Init(&argc,&argv);
        if (r != MPI_SUCCESS) {
                printf ("Error in MPI_Init()!\n");
                MPI_Abort(MPI_COMM_WORLD, r);
        }
        MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        printf ("Number of tasks= %d rank= %d\n", numtasks,rank);
        if( rank == 0){
                // master code here
        } else {
                // slave code here
        }
MPI_Finalize();
// serial code
}
```

Figure 5. An MPI code snippet

The program starts by initialization of the MPI environment by calling MPI_Init, checks if the initialization succeeded, determines the number of MPI processes started, retrieves the process rank, and writes these information to the standard output, then executes some code in the master process (the one with rank=0) and other code in the slave process(es). Finally, the MPI environment is terminated and some serial code could follow.

Building a cluster of computers for MPI parallel computing is could not be easy to achieve. For high performance computing we usually need a specialized network topology like Infiniband or Myrinet but for most of the economics problems an Ethernet network should be enough. There are easy to install packages the builds up a cluster in few minutes. One of them is described in (Creel, 2008) and it consists in a bootable CD (called Knopix) that allows building a cluster of computers running Linux operating system. It only uses the RAM memory of the computers in the cluster leaving the hard disks unmodified, such that, when the computers are restarted they are in the initial state with the operating system that had been install on them. This is a limitation of the Knopix approach because saving the work done between two sessions requires some special attention.

The fourth approach mention previously is the use GPU for speeding up computations. GPGPU (general-purpose computing on graphics processing units) is used widely nowadays to speed up numerical intensive applications. Traditionally, these parts of applications are handled by the CPUs but now GPUs have FLOP rates much better than CPUs (NVIDIA, 2014) because GPUs are specialized for highly parallel intensive computations and they are designed with much more transistors allocated to data processing rather than flow control or data caching.

Figures 6 and 7 (NVIDIA, 2014) shows the advances of the current GPUs that become highly parallel, many-core processors with an enormous computational power and very high memory bandwidth. Figure 6 shows the FLOP rates of NVIDIA GPUs in comparison with Intel multicore processors while figure 7 shows the memory bandwidth of the NVIDIA GPUs.

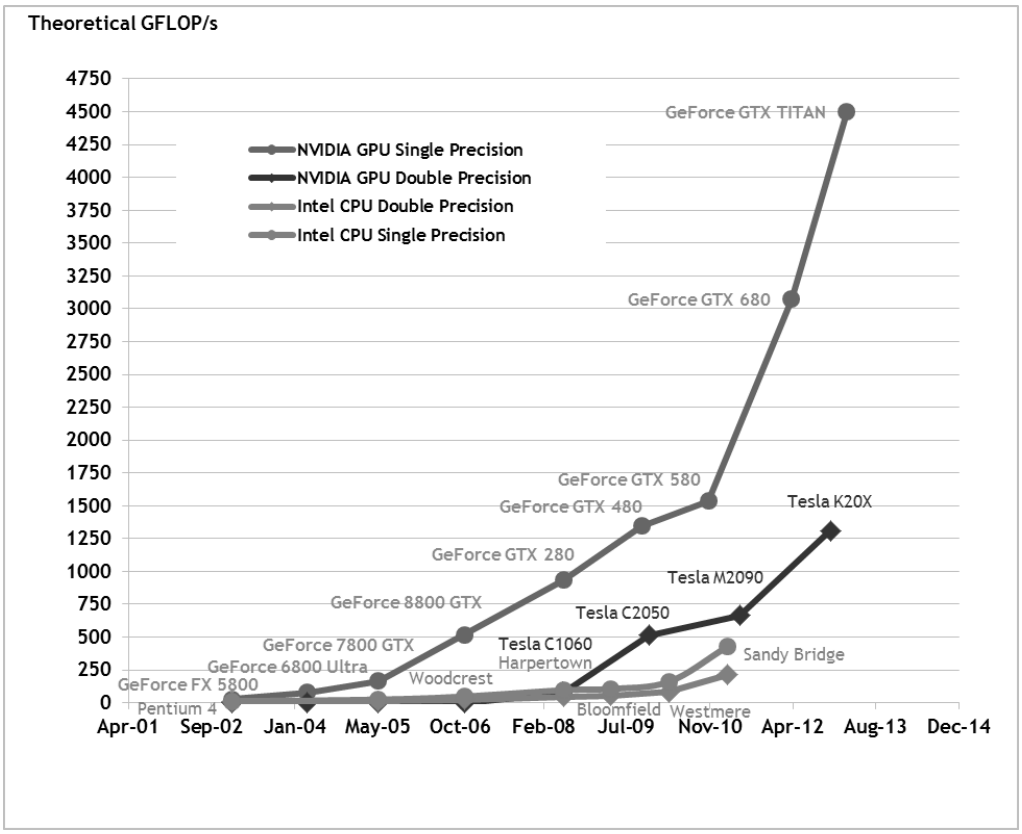

Figure 6. Theoretical FLOP rates of NVIDIA GPU and INTEL CPU

Source: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

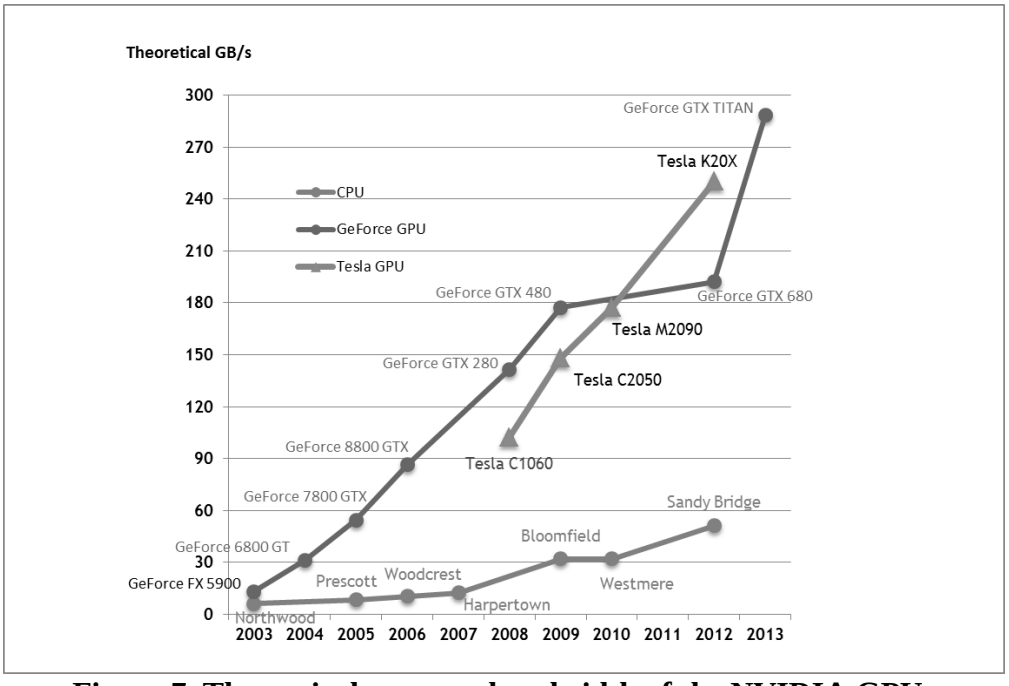

Figure 7. Theoretical memory bandwidth of the NVIDIA GPUs

Source: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

GPUs obtain high FLOP rates because they are specialized for highly parallel computations and they have more transistors dedicated to data processing rather than flow

control or data caching as in the case of a CPU. Figure 8 (NVIDIA, 2014) shows this design paradigm for GPU compared with CPU.

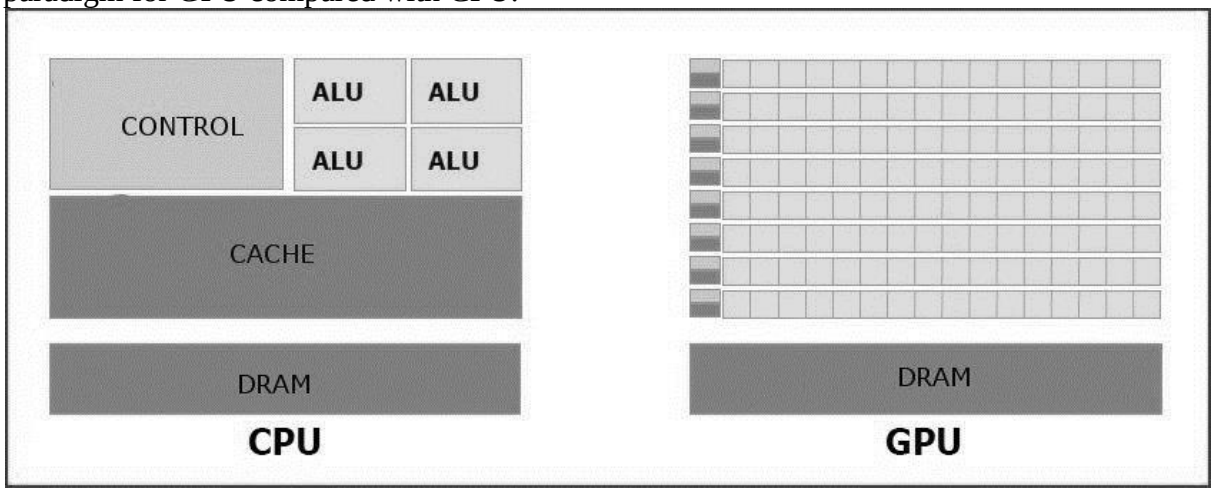

Figure 8. Design differences between GPU and CPU

Source: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

GPUs are designed to solve problems that can be formulated as data-parallel computations – the same instructions are executed in parallel on many data elements with a high ratio between arithmetic operations and memory accesses similar with the SIMD approach of the parallel computers taxonomy.

CUDA (Compute Unified Device Architecture) was introduced in 2006 by NVIDIA and is a general purpose parallel programming architecture that uses the parallel compute engine in NVIDIA GPUs to solve numerical intensive problems in a more efficient way than a CPU does. It supports programming languages like FORTRAN, C/C++, Java, Python, etc.

The CUDA parallel programming framework works with three important abstractions: a hierarchy of thread groups, shared memories, and barrier synchronization exposed to the programmer as language extensions. The CUDA parallel programming paradigm requires programmers to partition the problem into coarse tasks that can be run in parallel by blocks of threads and to divide each task into finer groups of instructions that can be executed cooperatively in parallel by the threads within on block. Figure 9 (NVIDIA, 2014) shows how a program is partitioned into blocks of threads each block being executed independently from each other.

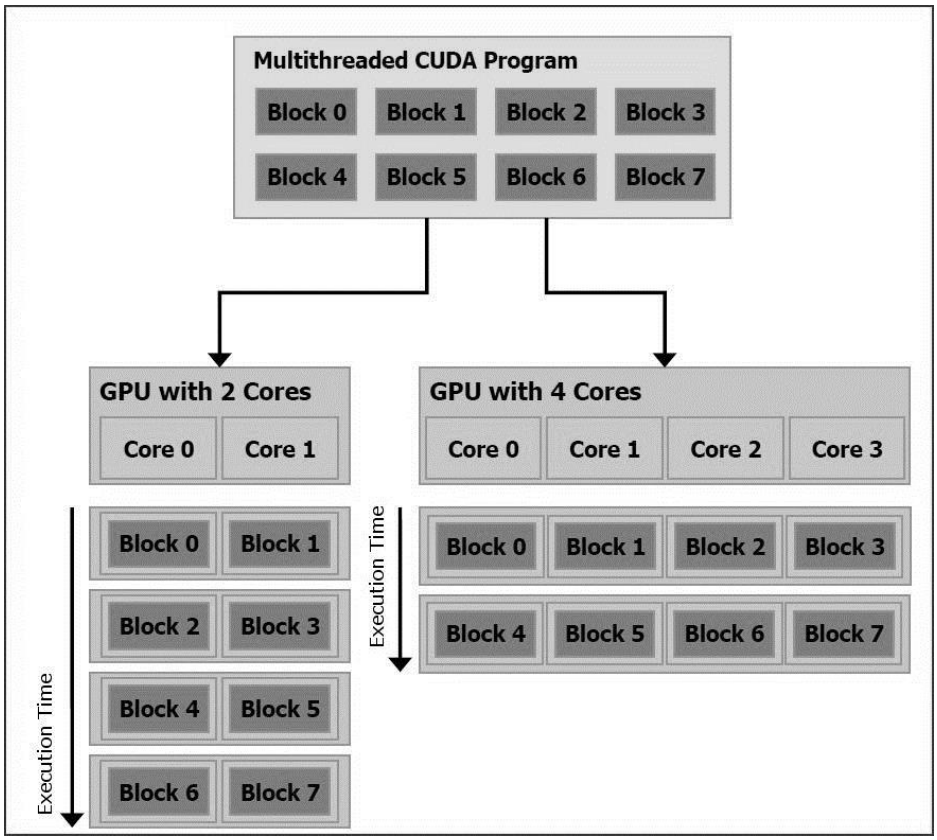

Figure 9. A multithreaded program splitted into blocks that are allocated on 2 or 4 cores of the GPU

Source: http://docs.nvidia.com/cuda/cuda-c-programming-guide/

The C language extension of the CUDA programming framework allows the programmer to define special C functions, called kernels, that are executed in parallel by different CUDA threads. Each thread is identified by a unique thread ID. The IDs of the threads are accessible within the kernel through the built-in threadIdx variable. ThreadIdx is a 3-component vector, so that each thread can be identified using a one-dimensional, two-dimensional, or three-dimensional index.

CUDA threads are executed on a physically separate *device* that operates like a coprocessor to the *host* processor running the C program. The device is located on the GPU while the host is the CPU. The CUDA programming model presumes that the host and the device maintain their own separate memory spaces respectively *host memory* and *device memory*. Using CUDA programming model a matrix multiplication code $C = A \times B$ can be structured like in the following example (Oancea, 2012):

```
/* allocate memory for the matrices A, B, C in the host memory, A,B,C, being N x N matrices*/
host_matrix_A = (float*)malloc(N * N * sizeof(host_matrix_A[0]));
host_matrix_B = (float*)malloc(N * N * sizeof(host_matrix_B[0]));
host_matrix_C = (float*)malloc(N * N * sizeof(host_matrix_C[0]));
/* generate random test data */
randomTestData(host_matrix_A, host_matrix_B, host_matrix_C)

/* allocate memory for the matrices in the device memory space*/
cudaAlloc(N*N, sizeof(device_matrix_A[0]), (void**)&device_matrix_A);
cudaAlloc(N*N, sizeof(device_matrix_B[0]), (void**)&device_matrix_B);
cudaAlloc(N*N, sizeof(device_matrix_C[0]), (void**)&device_matrix_C);

/* copy the values from the host matrices to the device matrices */
```

```
cudaSetVector(N*N, sizeof(host_matrix_A[0]), host_matrix_A, 1, device_matrix_A, 1);
cudaSetVector(N*N, sizeof(host_matrix_B[0]), host_matrix_B, 1, device_matrix_B, 1);
cudaSetVector(N*N, sizeof(host_matrix_C[0]), host_matrix_C, 1, device_matrix_C, 1);

/* call the matrix multiplication routine*/
cudaMatMul(device_matrix_A, N, device_matrix_B, N, beta, device_matrix_C, N);

/* Copy the result from the device memory back to the host memory */
cudaGetVector(N*N, sizeof(host_matrix_C[0]), device_matrix_C, 1, host_matrix_C, 1)
```

Figure 10. CUDA C code for matrix multiplication.

Besides CUDA there are other frameworks that allow GPU programming. AMD Accelerated Parallel Processing (former ATI Stream technology) is a set that enable AMD graphics processors, working in together with the central processor (CPU) to accelerate applications (AMD, 2013). AMD Accelerated Parallel Processing implements the OpenCL (Open Computing Language) standard (Khronos OpenCL Working Group, 2009) which is the first open standard for general-purpose parallel programming of heterogeneous systems. It tries to provide a unique programming environment for software developers to write portable code for servers, laptops, desktop computer systems and handheld devices using a both multi-core CPUs and GPUs. OpenCL programs are divided in two parts: one that executes on the device (the GPU) and other that executes on the host (the CPU). OpenCL uses a SIMT (SINGLE INSTRUCTION MULTIPLE THREAD) model of execution that reflects how instructions are executed in the host. This means that the same code is executed in parallel by a different thread, and each thread executes the code with different data.

The fifth approach combines GPU computing with a distributed cluster of computers. The simplified structure of the computing architecture is presented in Figure 7.

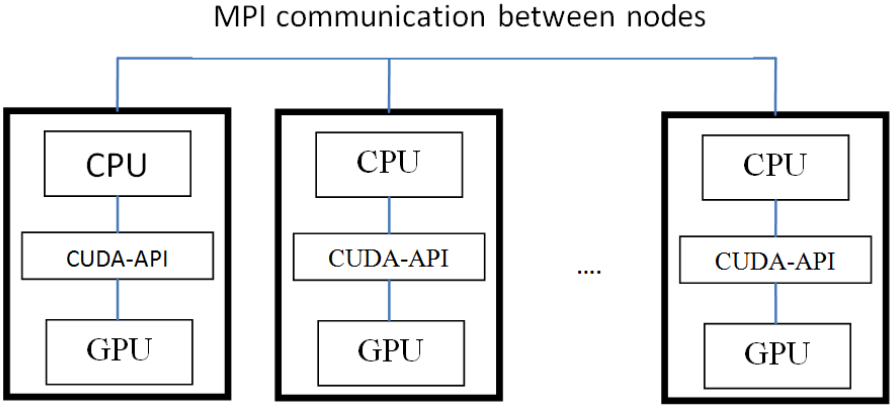

Figure 11. MPI – CUDA hybrid architecture

MPI is used to facilitate the communication between nodes and exploit coarse grained parallelism of the applications and CUDA accelerates local computations on each node exploiting the fine grained parallelism.

3. Parallel computing in economics

Parallel computing has been employed in solving economics problems in the last two decades.

One of the fields that require high computing power is macroeconometric modelling with forward-looking variables because it involves very large systems of equations, macroeconometric models with rational expectations described in (Fischer, 1992) being one

of them. They contain variables that forecast the economic system state for the future periods t + 1, t + 2, ... , t + T, where T is the forecast time horizon and could result in systems with tens or hundreds of thousands of equations. Such models are described in literature (Oancea, 2011a): MULTIMOD, QPM (Quarterly Projection Model), FRB/US. Solving these models involves solving systems of equations with or hundreds of thousands of equation that can be done using parallel processing (Oancea, 2011a).

(Fragniere, 1998) presents a procedure to solve big stochastic financial models with 1,111,112 constraints and 2,555,556 variables. Using a cluster of PCs with MPI this problem was solved in less than 3 hours.

(Gondzio, 2000) presents asset liability management problem solved with a stochastic model that resulted in 4,826,809 scenarios, the corresponding stochastic linear program having 12,469,250 constrains and 24,938,502 variables. At that time, this was the largest linear optimization problem in economics. The problem was solved on a PARSYTEC machine with 13 using 13 processors in less than 5 hours.

(Aldrich, 2011a, 2011b, 2014) shows how GPU computing can be used to solve economic problems. He presents a canonical real business cycle (RBC) model solved with value function iteration using a single threaded C++ program, an OpenMP program executed on 4 cores and CUDA on a Tesla C2075 device. He obtains a speed-up of 2500 between C++ serial implementation and CUDA version. In (Aldrich, 2011a) the author investigates an asset exchange within a general equilibrium model with heterogeneous beliefs about the evolution of aggregate uncertainty, reporting the computational benefits of GPU parallelism.

(Creel, 2005, 2007, 2012a, 2012b) describes a GPU algorithm for an estimator based on an indirect likelihood inference method. The estimation application arises in various domains such as econometrics and finance, when the model is fully specified, but too complex for estimation by maximum likelihood. The author compare the GPU algorithm run on a four NVIDIA M2090 GPU devices with a serial implementation executed on two 2.67GHz Intel Xeon X5650 processor showing a speed up factor of 242.

(Dziubinski, 2014) describes the same RBC model as (Aldrich, 2011a) but solved with C++ AMP technology introduced by Microsoft that allows exploiting the parallelism of the GPU in a manner transparent to the programmer.

4. Conclusions

This paper shows how the hardware and software environments have changed over the last decades and how parallel computing could be used even one can have only commodity hardware at his/her disposal. Multicore processors, GPU computing, MPI clusters have become available to economists for solving large problems that require high performance computing. In the future, parallel processing tools will become more and more accessible to programmers, allowing writing programs for multicore processors or GPU without knowing much details about the underlying hardware.

References

- AMD, (2013), AMD Accelerated Parallel Processing. OpenCL Programming Guide.
- Aldrich, E. M. (2011a), “Trading Volume in General Equilibrium with Complete Markets," Working Paper.
- Aldrich, E. M., Fernandez-Villaverde, J., Gallant, A. R., and Rubio-Ramirez, J. F. (2011b), “Tapping the supercomputer under your desk: Solving dynamic equilibrium models with graphics processors," Journal of Economic Dynamics and Control, 35, pp. 386-393.
- Aldrich, Eric M. (2104), "GPU Computing in Economics", Handbook of Computational Economics, Vol. 3, eds. Judd, Kenneth L. and Schmedders, Karl, Elsevier, chap. 10.
- Barron, I. M. (1978), The Microprocessor and its Application: an Advanced Course, in D. Aspinall. ed. "The Transputer", Cambridge University Press.

- Batcher, K. E. (1980), Design of a Massively Parallel Processor, IEEE Transactions on Computers, Vol. C29, September, pp. 836–840.
- Creel, M., and Zubair, M., (2012a), High Performance Implementation of an Econometrics and Financial Application on GPUs, Proceeding SCC '12 Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, pp. 1147-1153.
- Creel, M., Mandal, S., and Zubair, M., (2012b), Econometrics on GPUs, working paper No 669, Barcelona Graduate School of Economics, 2012.
- Creel, M., Goffe, W. (2008), Multi-core CPUs, Clusters, and Grid Computing: A Tutorial, Computational Economics, 32, (4), pp. 353-382.
- Creel, M., (2005), User-Friendly Parallel Computations with Econometric Examples, Computational Economics, 26, (2), pp. 107-128.
- Creel, M., (2007), I ran four million probits last night. HPC Clustering with ParallelKnoppix, Journal of Applied Econometrics, 22(1), pp. 215-223.
- Charalambous, M., Trancoso, P., and Stamatakis, A. (2005), "Initial Experiences Porting a Bioinformatics Application to a Graphics Processor," in Vol. 3746 of Lecture Notes in Computer Science, eds. Bozanis, P. and Houstis, E. N., New York, USA: Springer-Verlag, pp. 415-425.
- Dziubinski, M. P., and Grassi, S. (2014),"Heterogeneous Computing in Economics: A Simplied Approach," Computational Economics, 43, pp. 485:495.
- Fisher, P., (1992): *Rational Expectations in Macroeconomic Models*. Kluwer Academic Publishers, Dordrecht.
- Fragniere, E., Gondzio, J., Vial, J.P., (1998), "Building and Solving Large-Scale Stochastic Programs on an Affordable Distributed Computing System", Logiclab Technical Report 11, June, 1998.
- Kiessling, A., (2009), An Introduction to Parallel Programming with OpenMP, The University of Ediburgh.
- Gondzio, J., Kouwenberg, R., (2000), "High Performance Computing for Asset Liability Management",Technical Report MS-99-004,University of Edinburgh, March, 2000.
- Gropp, W., Lusk, E., Doss, N., and Skjellum, A., (1996), "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard", Parallel Computing, 22, pp. 789–828.
- Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K. P., Rudolph, L., Snir, M., (1982), The NYU Ultracomputer - designing a MIMD, shared-memory parallel machine, ISCA '82 Proceedings of the 9th annual symposium on Computer Architecture, pp. 27 – 42.
- Harris, M. J., Baxter III, W. V., Scheuermann, T., and Lastra, A., (2003), Simulation of Cloud Dynamics on Graphics Hardware. In Proceedings of the IGGRAPH/Eurographics Workshop on Graphics Hardware, pp. 92-101.
- Khronos OpenCL Working Group (2009), The OpenCL Specification - Version 1.0. The Khronos Group, Tech. Rep.
- Intel, (2014), Intel® Xeon Phi™ Product Family Performance Rev 1.412/30/13.
- LAM/MPI Parallel Computing, <http://www.lam-mpi.org> 2014.
- NVIDIA, (2014), NVIDIA CUDA C Programming Guide, version 5.5
- OpenMP ARB (2011), OpenMP API 3.1 C/C++ Directives.
- Open MPI: Open Source High Performance Computing, <http://www.open-mpi.org/>, 2014.
- Oancea, B, Andrei, T., Rosca, I. Gh., Iacob, A. I. (2011a), "Parallel algorithms for large scale econometric models", Procedia Computer Science, Vol. 3, pp. 479-483.
- Oancea, B., Rosca, I. G., Andrei, T., Iacob, A. I., (2011b), Evaluating Java performance for linear algebra numerical computations , Procedia Computer Science, vol. 3, pp. 474-478.
- Oancea, B., Andrei, T., Dragoescu, R. M., (2012), GPGPU COMPUTING, Proceedings of the "Challenges for the Knowledge Society", pp. 2026-2035.
- Tucker, L. W., Robertson, G., (1988), Architecture and Applications of the Connection Machine, Computer, vol. 21, no. 8, pp. 26–38.