Maciej Panczyk[1]

IMPROVING COMPUTATION EFFICIENCY BY PARALLEL PROGRAMMING

All modern computers are equipped with multi-core processors, and the majority of them also have graphics cards for carrying out vector calculations. Every aspiring programmer should know the techniques of parallel and distributed programming. The paper presents recent trends in programming and applications using central and graphics processing units.

Keywords: *parallel and distributed programming, GNU Parallel, OpenMP, MPI.*

Мачєй Паньчик

ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ РОЗРАХУНКІВ ШЛЯХОМ ПАРАЛЕЛЬНОГО ПРОГРАМУВАННЯ

У статті обговорено проблему того, що всі сучасні персональні комп'ютери вже обладнані багатоядерними процесорами, більшість із них - і відеокартами для векторних розрахунків, а кожен програміст має бути знайомий із методами паралельного і розподіленого програмування. Представлено сучасні тренди в програмуванні і додатках, що використовують центральний і графічний процесори для обробки даних.

Ключові слова: *паралельне і розподілене програмування, утиліта GNU parallel, стандарт OpenMP, багатопоточність.*

Мачей Паньчик

ПОВЫШЕНИЕ ЭФФЕКТИВНОСТИ РАСЧЕТОВ ПУТЕМ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ

В статье обсуждена проблема того, что все современные персональные компьютеры уже оборудованы многоядерными процессорами, большинство из них - и видеокартами для векторных расчетов, и каждый программист должен быть знаком с методами параллельного и распределенного программирования. Представлены современные тренды в программировании и приложениях, использующих центральный и графический процессоры для обработки данных.

Ключевые слова: *параллельное и распределенное программирование, утилита GNU parallel, стандарт OpenMP, многопоточность.*

Probably the less complicated way of programming using modern equipment is to use the OpenMP standard for multiprocessor computers with common memory. Today even the home network with a throughput of 1 Gbit allow using multiple computers for calculations within a local network using the MPI (message passing interface) standard. The use of even inexpensive graphics cards for calculations can easily reach the performance of 0.5 TFlop. It should be remembered that few years ago to achieve 1TFlop calculation speed required the construction of entire data centers. CUDA (compute unifed device architecture), introduced by NVIDIA enabled the use of advanced multiprocessor units such as graphics cards today. OpenCL (open computing language) is a similar solution based on royalty free open standard license.

[1] Lecturer, Faculty of Electrical Engineering and Computer Science, Institute of Computer Science, Lublin University of Technology, Poland.

To demonstrate these technologies, we will use simple examples that do not require special complicated numerical treatment, starting from an interesting and simple modification of shell programming called GNU Parallel.

GNU Parallel. GNU parallel [PAR12] is an extension to shell tool dedicated to executing jobs in parallel using one multiprocessor computer or more networking computers. Its main task is to run jobs for each of the lines in the input. It has many common options with xargs command.

Possibly the easiest usage is to group a few commands into one script. Here we have a simple script which do three time (CPU) consuming actions - calculate md5sum for a big file, converts few images into one PDF file and compress another file using bzip2 algorithm:

```
$ cat my_script
md5sum bigfile.iso
convert -density 300x300 `ls pic*.jpg` holidays.pdf
bzip2 poster.pdf
```

That job can be spread into several processors simply by sending it to parallel via pipe and done simultaneously:

```
$ cat my_script | parallel
```

Another example - let's execute a simple script that runs 6 time sleep command on a 6 core machine. File parallelsleep10.sh contains:

```
sleep 10
sleep 10
sleep 10
sleep 10
sleep 10
sleep 10
```

time command allows us to compare sequential and parallel execution:

```
$ time ./parallelsleep10.sh
real1m0.021 s
$ time cat parallelsleep10.sh | parallel
real0m10.360 s
```

Compression of 6 files using bzip2 algorithm can be performed as before taking input from stdin (standard input):

```
time seq 6 | parallel tar cvf BigFile{}.avi.tar BigFile{}.avi
BigFile1.avi
BigFile3.avi
BigFile4.avi
BigFile2.avi
BigFile6.avi
BigFile5.avi
real 0m21.851 s
```

It is also possible to read them as a command line arguments:

```
time parallel tar xvjf {} ::: BigFile?.avi.tar.bz2
BigFile5.avi
BigFile3.avi
BigFile6.avi
```

```
BigFile4.avi
BigFile2.avi
BigFile1.avi
real   1m36.823 s
```

An important feature of GNU Parallel is the ability to run parallel jobs on multiple machines across a network. By using appropriate options it is possible send data to a remote (fast) machines, to process it and to receive results in one line. To read more about this useful and easy to use tools, it is recommended to visit the home of the GNU Parallel, read articles [TAN11], [MAR10] or find many interesting examples on www.youtube.com.

OpenMP. OpenMP (open multiprocessing) [OMP12, KAR09, PAP06, CHA01, CHA08] is an application programming interface for parallel programming on shared memory multiprocessors. It runs on most operating systems and consists of a set of compiler directives, a library of support functions, and environment variables that influence run-time behavior. OpenMP works in conjunction with C, C++, and Fortran. The core elements of OpenMP are the constructs for thread creation, workload distribution (work sharing), data-environment management, thread synchronization, user-level runtime routines and environment variables.

Frequently used first program type "Hello World!" in OpenMP looks very familiar. The only difference is compiler directives #pragma (used in C/C++).

```
#include <stdio.h>
int main(void)   {
    #pragma omp parallel
    printf("Hello, world!\n");
    return 0;
}
```

Listing. 1. Parallel program "hello_world" using OpenMP

To compile it using GCC it is necessary to add   -fopenmp   flag:

```
$gcc   -fopenmp   hello_world.c   -o   hello_world
```

After defining number of threads equal to 2:

```
export OMP NUM THREADS=2
```

the result on a computer with 2 Cores and 2 threads is the following:

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

Next mid-level example (listing 2: program maximum) [SPI11] except previously used "#pragma omp parallel" defines a range of variables and uses two nice constructions: "#pragma omp for" and a barrier "#pragma omp critical" within parallel section of code limited by { } braces. Parallel loop "for" divides its simultaneous execution between the declared number of threads, and then waits for the completion of all threads which is forced by the critical section. The result of the program is to determine the maximum of function, as shown in Figure 1.

```
#include <stdio.h>
#include <math.h>
#include <omp.h>
#define N 10000001
float a[N];
```

```
main(){
int i,n=10000000;   float max=0.0,tmp_max=0.0;   double time;
#pragma omp parallel firstprivate(tmp_max) shared(max)
{ // beginning of parallel section
#pragma omp for
for (i=0;i<n;i++) a[i]=1.0*sin(i*2.0*M_PI/n)+0.5*sin(9.0*i*2.0*M_PI/n);
time=omp_get_wtime();
#pragma omp for nowait
for(i=0;i<n;i++) if(a[i]<tmp_max) tmp_max=a[i];
#pragma omp critical
if(tmp_max<max) max=tmp_max;
} // end of parallel section
time=omp_get_wtime()-time;
printf("time=%lf\nmax = %f\n",time,max);
}
```

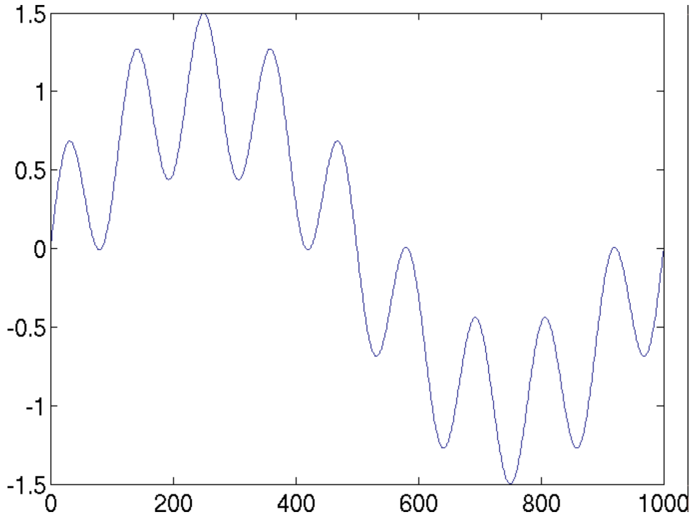

Figure 1. **Data set for program "maximum"**

Listing 2. Program "maximum" which calculates in parallel maximum of a given dataset using OpenMP [SPI11]

After compiling the program:

```
$ g++ -o maximum -fopenmp maximum.cc
```

and declaring number of threads to 1:

```
$ export OMP_NUM_THREADS=1
```

program runs sequentially on one processor within 0.034221 seconds.

```
$ ./maximum
time=0.034221
max = -1.500000
```

Comparable execution for 6 threads on 6 core machine takes only 0.009229 seconds:

```
$ export OMP_NUM_THREADS=6
$ ./maximum
time=0.009229
max = -1.500000
```

The purpose of the above OpenMP programming examples was only to notice that the transition from pure C/C++ programming into C/C + in OpenMP is not so complicated. It relies upon fork/join parallelism. A master thread is executed as sequential code until it reaches a parallel code segment. Then it forks other threads which communicate each other via shared variables. In the end of parallel part these threads synchronize, rejoining the master one.

MPI. MPI (message passing interface) [SPI11, KAR09, PAP06, MOR09, GRO99, GLT99] is a library specification for message-passing, communication protocol which is standard for sending messages between processes running parallel programs on one or more computers. The standard defines the syntax and semantics of a core of library routines for programs written C/C++ or Fortran programming language. Sample implementations [QUI03] for those routines are MPICH [MCH12] and OpenMPI [MPI12, HPC12].

First example - a simple program like "Hello World!", shows that MPI is based on a group of functions which allows for communication between a set of processes due to communicator objects which connect groups of processes in the MPI session.

```
#include <stdio.h>
#include <mpi.h>
#include <unistd.h>
int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);
    printf("Hello, world! - from process %d out of %d on host %s\n", rank,
numprocs, processor_name);
  MPI_Finalize();
}
```

Listing. 3. MPI version of parallel program "hello_world"

To compile the above program (from listing 3.) written in C we need to execute:

```
mpicc  -o  mpi_hello  mpi_hello.c
```

To launch the parallel calculation we also need a special command:

```
mpirun  -np  4  ./mpi_hello
```

where "-np 4" option tells the task to be executed on 4 processors.

The result of the "MPI Hello world" program is similar to OpenMP version ie. each process prints its message on the screen:

```
Hello World! from process 0 out of 4 on server1
Hello World! from process 1 out of 4 on server1
Hello World! from process 2 out of 4 on server2
Hello World! from process 3 out of 4 on server2
```

Unlike OpenMP programs, these which use MPI functions can be executed not only on a single multiprocessor machine but also simultaneously on many computers which take part in the calculations.

The example presented below is similar to OpenMP "maximum" program (listing 2) but MPI set contains reduction operation MPI_MAX which can choose the maximum value from single thread and within multiple threads using MPI functions MPI_Reduce and MPI_Reduce_scatter. That simplifies program code.

```
#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>
int main( int argc, char **argv ) {
   int err = 0;
   int *sendbuf, recvbuf[8], *recvcounts;
   int size, rank, i, sumval;
   MPI_Comm comm;
   MPI_Init( &argc, &argv );
   comm = MPI_COMM_WORLD;
   MPI_Comm_size( comm, &size );
   MPI_Comm_rank( comm, &rank );
   sendbuf = (int *) malloc( size * sizeof(int) );
   for (i=0; i<size; i++)
      sendbuf[i] = rank * (i+1);
                                    fprintf(stdout,        "[Rank        %d]
sendbuf[0..7]=%2d,%2d,%2d,%2d,%2d,%2d,%2d,%2d\n", rank, sendbuf[0], send-
buf[1], sendbuf[2], sendbuf[3], sendbuf[4], sendbuf[5], sendbuf[6], sendbuf[7]);
   fflush(stdout);
   recvcounts = (int *)malloc( size * sizeof(int) );
   for (i=0; i<size; i++)
      recvcounts[i] = 1;
   recvbuf[0] = -1;
   MPI_Barrier (comm);
   MPI_Reduce_scatter( sendbuf, recvbuf, recvcounts, MPI_INT, MPI_MAX,
comm );
   fprintf(stdout, "[Rank %d] recvbuf[0]=%2d\n", rank, recvbuf[0]);
   fflush(stdout);
   int send1[1] = {recvbuf[0]};
   int reduce1 = -1;
   MPI_Barrier (comm);
   MPI_Reduce (send1, &reduce1, 1, MPI_INT, MPI_MAX, 0, comm);
   fprintf(stdout, "[Rank %d] reduce1=%2d\n", rank, reduce1);
   fflush(stdout);
   MPI_Finalize( );
   return 0;
}
```

Listing. 4. MPI version of parallel program "maximum" which generates and calculates maximum of a given dataset

The result of MPI program "maximum" is presented below:

```
[Rank 5] sendbuf[0..7]= 5,10,15,20,25,30,35,40
[Rank 1] sendbuf[0..7]= 1, 2, 3, 4, 5, 6, 7, 8
[Rank 2] sendbuf[0..7]= 2, 4, 6, 8,10,12,14,16
[Rank 4] sendbuf[0..7]= 4, 8,12,16,20,24,28,32
[Rank 6] sendbuf[0..7]= 6,12,18,24,30,36,42,48
[Rank 0] sendbuf[0..7]= 0, 0, 0, 0, 0, 0, 0, 0
[Rank 7] sendbuf[0..7]= 7,14,21,28,35,42,49,56
[Rank 3] sendbuf[0..7]= 3, 6, 9,12,15,18,21,24
[Rank 1] recvbuf[0]=14
[Rank 7] recvbuf[0]=56
[Rank 4] recvbuf[0]=35
[Rank 5] recvbuf[0]=42
[Rank 3] recvbuf[0]=28
[Rank 2] recvbuf[0]=21
[Rank 0] recvbuf[0]= 7
[Rank 6] recvbuf[0]=49
[Rank 7] reduce1=-1
[Rank 1] reduce1=-1
[Rank 3] reduce1=-1
[Rank 2] reduce1=-1
[Rank 6] reduce1=-1
[Rank 5] reduce1=-1
[Rank 0] reduce1=56
[Rank 4] reduce1=-1
```

Listing. 5. MPI version of parallel program "maximum" results

The last example is based on the MPICH documentation. It calculates the value of π using the trapezoidal integration method:

$$\pi = \int_0^1 \frac{4}{1+x^2}dx$$

The integration interval is split between the processors, each responsible for their part of the integral. After the calculation, partial results are aggregated using the MPI_Reduce function. The source of the program is presented below:

```
#include <stdio.h>
#include <math.h>
#include <mpi.h>
int main(int argc, char **argv) {
    int my_rank,size;
    int i,intervals;
    double my_pi,pi,h,sum,x;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    if (my_rank==0)
    {
        printf("Enter the number of intervals: ");
```

```
            scanf("%d",&intervals);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        h=1.0/(double)intervals;
        sum=0.0;
        for (i=my_rank; i<intervals; i+=size)
        {     x=h*((double)i+0.5);
              sum+=4.0/(1.0+x*x);
        }
        my_pi=h*sum;
              MPI_Reduce(&my_pi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
        if (my_rank == 0)
              printf("pi=%.16f\n",pi);
        MPI_Finalize();
    }
```

Listing. 6. MPI program calculating π

The result for 1000000 intervals is 3.1415926535898899 .

Second MPI program (listing 6.) calculates the value of π uses collective communication. Collective functions involve communication among all the processes at a process group. Function MPI_Bcast takes data from one node and sends it to all the processes in the process group. MPI_Reduce function allows for reverse operation, it takes data from all the processes in a group, performs an operation (such as summing in our case or finding global maximum/minmum), and stores the results on one node.

Programming with MPI is more difficult than using GNU Palallel or OpenMP standard nevertheless it is worth to understand its basics. For many common tasks it is enough to use 6-25 functions.

Conclusion. In a brief article it is not possible to present all the possibility of multiprocessor computers usage or programming computers in a network which works at a common task. Nevertheless, OpenMP and MPI are standards for parallel and distributed programming. In the era of multicore computers and computer networks, these standards are essential for teaching programming. Typical course usually begins with POSIX threads in C and some popular C++ threads libraries like Boost. That is usually enough for one teaching subject. Next step is to use multi-threads in other popular languages like Java. Nowadays architectures tend to integrate shared memory machines into clusters. That means that clusters use heterogeneous computing mixing OpenMP and MPI. These techniques are used not only for sophisticated scientific computing but in usual auction and shopping websites like eBay or Allegro mentioned in the list of top 500 supercomputing centers.

Moreover, modern computers (not only clusters) use graphic cards for GPGPU. GPGPU is general-purpose computing on graphics processing units, which handles computation not only for computer graphics, but to perform computation in applications traditionally handled by the central processing unit. Second subject is to teach GPGPU techniques like CUDA and OpenCL. Unfortunately, it violates the principle often treated as a canon of good programming saying that the algorithm should be independent of the used equipment.

Efficient calculation of PGPU requires the knowledge of processor construction. From this point of view, innovations such as the GNU Parallel is a very nice, incentive, easy for use element to learn programming and general usage of modern computers, or modern technical solutions at all as even cell phones already have multi-core processors.

References:

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R. (2001). Parallel Programming in OpenMP, Academic Press, San Diego, USA.

Chapman, B., Jost, G., van der Pas, R. (2008). Using OpenMP portable shared memory parallel programming, MIT Press, Massachusets, USA.

GNU Parallel Homepage http://www.gnu.org/software/parallel/, accessed June 2012.

Gropp, W., Lusk, E., Skjellum, A. (1999). Using MPI: Portable Parallel Programming With the Message-passing Interface Scientific and Engineering Computation, MIT Press, USA.

Gropp, W., Lusk, E., Thakur, R. (1999). Using MPI-2: Advanced Features of the Message-Passing Interface, MIT Press, USA.

Karbowski, A., Niewiadomska-Szynkiewicz, E. (2009). Programowanie rownolegle i rozproszone, Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa.

Martin B. (2010). Leb w leb, LINUX Magazine, December, p.32-35.

http://en.wikipedia.org/wiki/OpenMP, accessed Jun 2012.

MPI Documents, http://www.mpi-forum.org/docs/docs.html, accessed Jun 2012.

MPICH2 home page, http://www.mcs.anl.gov/research/projects/mpich2, accessed Jun 2012.

Mordechai, B.-A. (2009). Podstawy programowania wspolbieznego i rozproszonego, WNT, Warszawa.

Open Source High Performance Computing, http://www.open-mpi.org, accessed Jun 2012.

Paprzycki M., Stpiczynski P. (2006). A Brief Introduction to Parallel Computing, the first chapter of the book: Handbook of Parallel Computing and Statistics, E.J.Kontoghiorghes Ed.,Taylor & Francis.

Spiczynski P., Brzuszek M. (2011). Podstawy programowania obliczen rownoleglych, UMCS Lublin.

Tange O. (2011). GNU Parallel - The Command-Line Power Tool, ;login: The USENIX Magazine, February, p.42-47.

Quinn, M. J. (2003). Parallel programming In C with MPI and OpenMP, McGraw Hill, Singapore.

Стаття надійшла до редакції 02.08.2012.