© University Press 2024
ISSN 2719-6550
ISSN 2719-7417 online
"Journal of Education, Technology and Computer Science"
No. 5(35)/2024
www.eti.ur.edu.pl

Received: 3.09.2024
Accepted for printing: 11.12.2024
Published: 20.12.2024
DOI: 10.15584/jetacomps.2024.5.5
License: CC BY-NC-ND 4.0

ROBERT PĘKALA [1], **JAKUB SZUMILAK** [2], **ADRIAN MUCHA** [3]

Parallel Programming in PC and Computer Cluster Environment – Selected Computational Problems

[1] ORCID: 0000-0003-0530-0005, Ph.D., Department of Computer Science, State University of Applied Sciences in Jarosław, ul. Czarnieckiego 16, 37-500 Jarosław, Poland
[2] ORCID: 0009-0006-7798-0897, student, State University of Applied Sciences in Jarosław, ul. Czarnieckiego 16, 37-500 Jarosław, Poland
[3] ORCID: 0009-0004-5491-7647, student, State University of Applied Sciences in Jarosław, ul. Czarnieckiego 16, 37-500 Jarosław, Poland

Abstract

Parallel programming is a skill that requires the use of technologies and techniques that allow applications to use multiple threads/processes simultaneously. In some cases, this is a condition for their launch or correct operation. This article presents selected aspects of such a programming using the example of two proposed applications: for PC computers with the .NET platform and application designed for a computer cluster operating in the GNU/Linux system. These are two applications with different purposes – the first uses image processing mechanisms, while the second – is the implementation of precise numerical calculations. The proposed applications were designed in the context of using multithreading and multiprocessing technologies. The obtained results indicate that implementing appropriate programming techniques is an important aspect of programming various types of applications, ensuring their correct operation and acceleration of long-term calculations.

Keywords: thread, process, synchronization, OpenMPI, Amdahl's law

Introduction

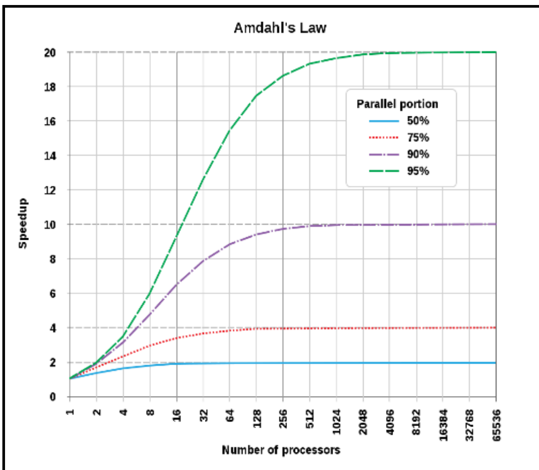
Sequential calculations, by default are performed serially, this means that one thread performs calculations one by one as they are placed in the application code. This method works well for small workloads or when tasks have strong data dependencies.

It should be noted that there are computational problems that require the use of large computing power and a large amount of memory. Without these condi-

tions solving such tasks is problematic and sometimes even impossible. The answer to this is parallel programming whose idea is to break the computation into subtasks and assign each of them to a different thread or process that can run independently of each other (Robey, Zamora, 2021; https://www.open-mpi.org 2024). Moreover, each thread can run on a different processor core within the same physical system or on the computer cluster which usually – especially in the second case – significantly translates into increased speed of operation (Rainders, 2007; Robey, Zamora, 2021).

Parallel programming is a relatively narrow specialization in the broadly understood field of programming. There are several internationally published scientific journals dedicated only to this programming technique. Solutions presented there are concerned to many fields of science and technology (e.q.: Collange, Defour, Graillat, Iakymchuk, 2015; Herrmann, Kuchen, 2023; Birath, Ernstsson, Tinnerholm, Kessler, 2024). However, this topic is also presented in other journals, (e.q.: Szyszko, Smołka, 2018; Hielscher, Bartel, 2024), if only the problem under consideration requires parallelization of computations.

What kind of increase in computational speedup we can expect depends on how much we can split a program code which can be assigned to separate threads or processes. The theoretical relationship between their number and application performance is described by Amdahl's law (Rainders, 2007). It indicates that the maximum acceleration of the execution time is limited by the largest indivisible part of the program. The specific courses and well known analytical formula, which should be treated rather as a classical approach, are shown below:



$$S = \frac{1}{(1-P)+\frac{P}{N}} \quad (1)$$

where: S – maximum speed up of the program,
N – amount of threads/processes,
P – is percentage of program code, that can be parallelized (parallel portion).

Figure 1. Amdahl's law of speeding up program execution depending on the number of processors (https://www.researchgate.net 2016)

Figure 1 shows the saturation of the curve representing the acceleration of the program. This means that with the number of processing units, theoretically approaching infinity, the time to solve a given task is established because the acceleration cannot exceed the value determined by the sequential part of the program code – which of course is a feature of even the most modern computing systems, including computer clusters.

One of the important aspects in parallel programming is the problem of synchronizing threads/processes so that each works efficiently on their task and maintains data integrity. Two or more threads can use and change the same variable. If the processes are not synchronized, the variable will not reach the proper value after the calculation is completed. This applies to both solutions implemented using cluster technologies, as well as applications created for PC computers. In our article, using the example of created applications, we show how implementing parallelized code has a positive impact not only on the speed of operation, but also on their responsiveness. We also pay attention to selected aspects of synchronization, which in turn ensures the correctness of the calculation of shared variables. Presented applications were implemented in the .NET programming environment and the C# language, as well as in the C++ language environment and the OpenMPI library of the didactic computing cluster, which is equipped with the Department of Computer Science at the State University of Applied Sciences in Jarosław (PANS Jarosław).

Technology of multithreading

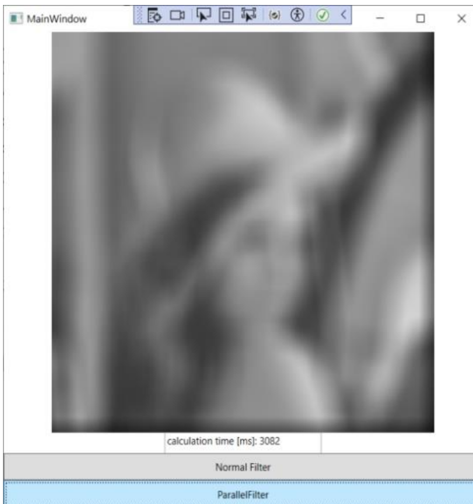
As the name suggests, a program that implements a multithreading feature can have the code run on multiple threads at the same time. We can treat threads as workers and the program as the supervisor of the workers. Multithreading is the ability of the "supervisor" to split the work among his "workers" instead of having just one worker do all the work on their own. Each thread is given a unique ID. This way the flow remains controllable and the threads can be managed through dedicated libraries to avoid leaks and errors. In our case we consider *Task Parallel Library* (TPL) for .NET Framework (https://learn.microsoft.com/pl-pl/dotnet/standard/parallel-programming 2022)

The main reason to use, and the biggest advantage of multithreading is the application execution speed. If implemented correctly, multithreading can cause the code to finish much faster than it would in a single thread. Moreover, running complex code on only one thread can sometimes create application security issues or other deficiencies in its functioning. On the other hand though, keeping multiple threads in check, starting them, making sure they don't interrupt each other pose a serious challenge, therefore writing programs that make use of multithreading is much more difficult and takes much more time. What is more, errors that originate from using multithreading are much more difficult to spot.

It should be noted that not all cases of implementing multithreaded technology collide with the problem of synchronization related to the sharing of variables. There exist applications in which threads can be executed asynchronously, concurrently, and even in parallel, while the synchronization mechanism comes down only to the "supervisor" collecting the results from the "workers". As an illustration of this problem, an application from the field of image processing was implemented. The figures below show the performance results of an averaging filter for a bitmap image (https://eeweb.engineering.nyu.edu) with a resolution of 512x512 pixels. The filter calculates new values of each pixel as an average of the other pixel values from the area of the sliding 40-pixel wide window, which provides an intense blurring of the image. The filter works in two versions: a serial version and a parallel version. The output under the image shows the running time of both versions of the filter.

a)

b)



c)

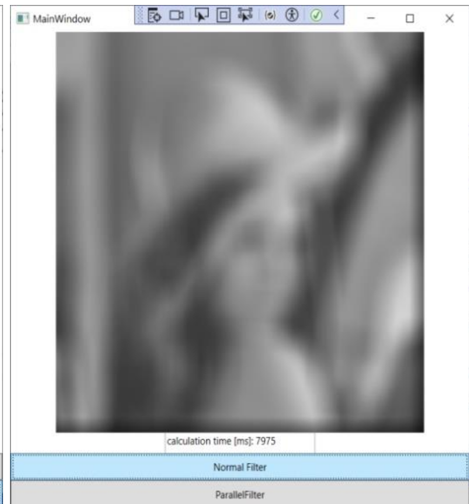


Figure 2. Original image (a) as the input to the averaging filter and calculation times in milliseconds: for serial calculations (b) and for parallel calculations (c)

The application was implemented in the C# language environment and the .NET platform. For the version of the parallel filter, the total image area was divided into 4 blocks of pixels, which is determined by the variable declared in the first line of the application code (Figure 3). Each block of parallel filter is processed by a separate thread, while the serial calculations run in the context of the main thread only. Additional threads of the parallel filter were run in the context of the task factory (*Task.Run*() method) using the default queuing mechanism. The *Filter*() method – used as a parameter – calculates the average value for each pixel in a 40-pixel window. Synchronization between additional threads and the main thread is provided by the *WaitAll*() method (the last red code line). It causes the main thread to wait until the additional threads complete their calculations. Running the application on a PC with Intel Core i7 2.2 GHz processor indicates that filtering the image in 4 additional threads speeds up the calculations several times.

```
const int number_of_blocks=4;
var tasks = new Task[number_of_blocks];
for (int i = 0; i < square_block; i++)
{
  if  (i == square_block - 1)  wEnd = (int)bitmap_k.Width;
  var hStart = 0; var hEnd = th;
  for (int j = 0; j < square_block; j++)
   {
     if (j == square_block - 1) hEnd = (int)bitmap_k.Height;
     var fWstart = wStart; var fWEnd = wEnd; var fHstart = hStart; var fHend = hEnd;
     tasks[block++] = Task.Run(() => Filter(img, imgCpy, w, h, WindowSz, fWstart, fWEnd, fHstart,
fHend));
     hStart = hEnd; hEnd += th;
   }
     wStart = wEnd; wEnd += tw;
 }
Task.WaitAll(tasks);
```

Figure 3. Application code snippet responsible for running the filter algorithm in 4 tasks-threads

Taking into account the formula (1) for the proposed application, in which $P = 0.82$ and $N = 4$, it can be stated that the estimated value of the maximum speed up obtained from Amdahl's law and the value obtained in real conditions are similar.

The second important aspect is that implementing filter calculations in additional threads makes the main application window responsive — unlike the serial filter version, where the graphical interface and the filter run in the same, main application thread.

Multiprocessing programming with using MPI – solution of a selected numerical problem

The application presented in the previous chapter concerns solutions that can be used on PC computers. However, demanding computational processes, e.g. complex engineering calculations, may require the construction of applications implemented on high-performance computing systems – usually computer clusters that implement computational algorithms in a distributed multiprocessor environment.

Applications built for computing clusters can use MPI *(Message Passing Interface)* technology. It is a set of libraries used to enable communication in parallel computing architectures. It provides useful functions in C, C++ and Fortran that allow control over multiple processes even if they are run on different machines (https://www.open-mpi.org 2024). The interface also supports synchronization and communication functionality between a set of processes. This communication involves sending messages between processes, hence the name – message passing interface. These messages may be, for example, variables of different types, representing data for various computational problems. In our research we used a cluster located in PANS Jarosław running under the GNU/Linux operating system with the *OpenMPI* library in the form of a loadable module. Simplified diagrams and general parameters of the cluster are shown on Figure 4.

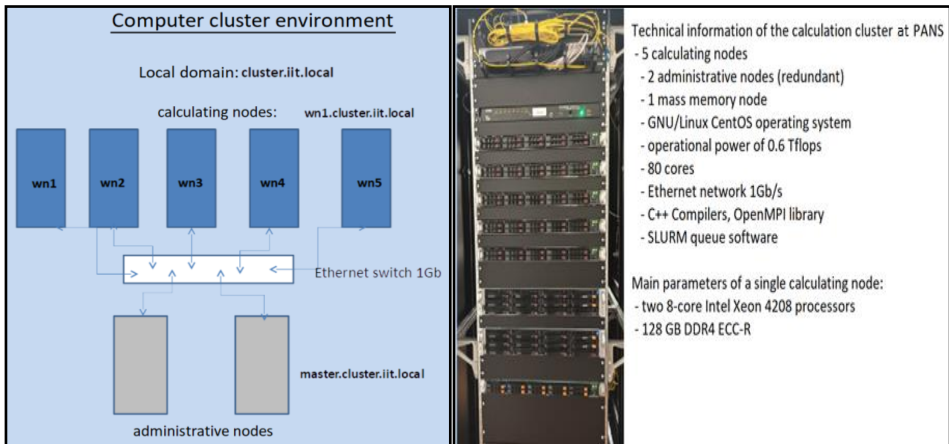


Figure 4. Computer cluster's connections and general specification of the nodes

Calculating nodes *wn1÷wn5* as well as the administrative node *master* (in redundant solution) are connected via an Ethernet switch. The built applications must be placed on the one of the master nodes that has the tools to compile and run them on calculating nodes.

Some problems from various fields of technology require calculation of definite integrals of a function of one variable. Calculated values of the integrals may represent different physical quantities describing a given phenomenon or object. Appropriate calculations can be performed for data in the form of analytically given integrands or when discrete values of an unknown function are given only.

In numerical computations, one of the key problems is to ensure the best possible accuracy. It is usually associated with the need to adopt discretization steps in time or space that can require resources significantly exceeding the capabilities of PC computers. Implementing calculations on a computer cluster with the OpenMPI library requires programmers to develop a different concept of calculations than in the case of the serial approach - which is not always easy. This of course also leads to the need to rebuild the serial application code into a new form. To demonstrate this problem in integral calculations, Simpson's quadrature was used in the form given by formula 2 (Flowers, 2000):

$$F = \frac{1}{3}\Delta x[f(a) + 4f(a + \Delta x) + 2f(a + 2\Delta x) + 4f(a + 3\Delta x) + 2f(a + 4\Delta x) + + \cdots + 4f(b - \Delta x) + f(b)] \qquad (2)$$

where: F denotes value of integral, a, b – b oundaries of calculations, Δx – discretization step of the independent variable x.

In C++ the code of in sequential version might look like on the Figure 5:

```
long double f (long double x)
{
   return sin(sqrt(x));
}
long double F (long double a, long double b, long int n)
{
   long double dx = (b - a) / (double)(n - 1);
   long double output = f(a) + f(b); int multiplier;
   for (long int i = 1; i <= n; i++)
```

Figure 5. Program code that allows to calculate sequentially the integral of the function $f(x) = sin\, sin\, (\sqrt{x}\,)$ in the interval [a, b] and with a number of steps n

Used declaration of the variable *n* allows the number of steps to be entered into the calculation area depending on the upper range of the *unisgned int* type, which should be sufficient in most practical applications. It should be noted, however, that PC calculations in the upper range of this type are time-consuming and may lead to errors.

As mentioned earlier, the implementation of the above algorithm in a computing cluster environment with the OpenMPI library requires the development of a different concept than the serial one for calculating the integral according to the formula (2). MPI is a standard for sending so-called messages between processes. This means that it is possible to run applications within multiple processes distributed across computing nodes, with the ability to transfer data between these processes. In our research, this idea was used so that individual processes performed calculations for a finite number of Simpson's formula components. Figure 6 shows the essence of the adopted solution.

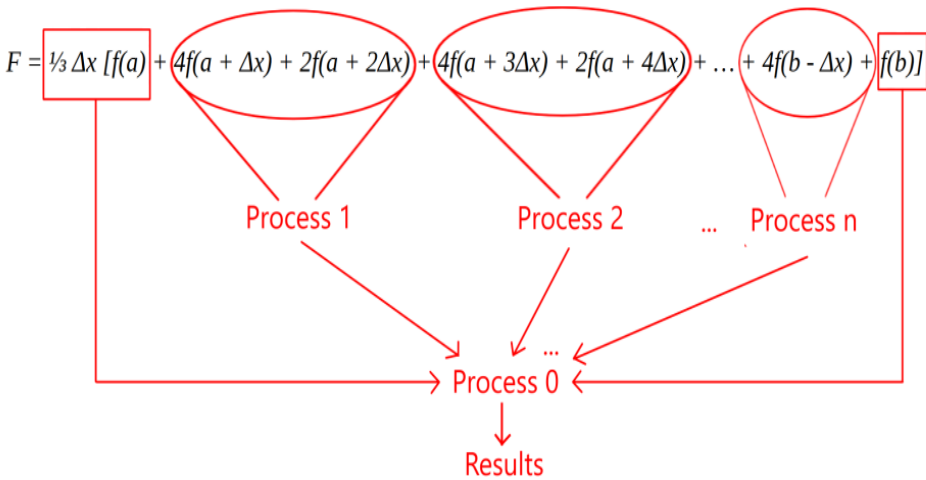


Figure 6. Visual explanation of idea the Simpson formula calculations by processes of OpenMPI library

Each library process has its own unique identifier expressed as a natural number. In our case, process no. 0 (called as root) was elected for storing the first and last elements of the Simpson formula (which do not repeat). Moreover, the middle parts of the formula are divided into other available processes. Once the calculations are done, the outcome of each of them is collected in root process, which sums partial totals and yields the final result of calculations. The most important part of the code of application is shown in the figure below.

```
// Calculate the function results at the first and the last index
if (processRank == 0) outcome = f(a) + f(b);
// Getting the equal range to perform calculations in
    unsigned int start = processRank * npart + 1; unsigned int end = (processRank + 1) * npart;
// Calculate the rest of the formula
    int multiplier; double addition = 0.0;
    for (unsigned int i = start; i <= end; i++) {
        // determining the multiplier
        if (i % 2 == 0) { multiplier = 2; } else { multiplier = 4; }
        addition += multiplier * f(a + (double)i * dx);
    }
    // Gathering the results
    if (processRank == 0) {
        outcome += addition;
        for (int j = 1; j < processNumber; j++) {
            MPI_Recv(&addition, 1, MPI_DOUBLE, j, MPI_ANY_TAG, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            outcome += addition;
        }
    } else { MPI_Send(&addition, 1, MPI_DOUBLE, 0, processRank, MPI_COMM_WORLD); }
```

Figure 7. Main part of OpenMPI application code

Sending partial sums to the root process is implemented using two OpenMPI library routines: MPI_SEND and MPI_RECEIVE ensure correct calculations – and very importantly – synchronization between processes. This means that the shared variable *outcome*, representing each partial sum, is correctly incremented in the root process. The figure below presents the final results of the integral calculations of the function $f(x) = sin\, sin\, (\sqrt{x})$. For comparison, the results for sequential calculations according to the code from Figure 5 are also included.

The calculations were performed for the same conditions, i.e. the integration interval [a, b] and the discretization step Δx. The SLURM computational task queuing system was used to run the application on the cluster. It is a convenient tool that allows running the application with a specified number of processes declared by the parameter n on N computational nodes. According to the technical data (Figure 4), the application can be run on a maximum of 5 computing nodes (N=5). Each of them has 16 processor cores, so the maximum number of available cores is 80. Of course, the declared number of processes, using the n parameter can be greater than the number of cores, but our results include a number of processes equal to the maximum number of physical cores of the cluster. The computation times for different variants are given on Figure 8.

Obtained results show that even sequential calculations on a PC and on a selected cluster node differ significantly due to the more efficient processor and larger RAM resources of the node. Using the full computing power of the cluster means that the computation time is almost 100 times shorter than on a PC. Therefore, the adopted parallel computation algorithm can be considered highly effective.

```
Computer Cluster

[szumilakj@master Programy]$ srun -N 1 -n 1 SimpsonIntegral.exe

Total steps:                4294967295
Processes run:              1
Time elapsed:               152.364s
Calculated integral:        4.46340952338899743523

PC Computer

Total steps:                4294967295
Time elapsed:               290.687s
Calculated integral:        4.46340952338899921159
```

```
[szumilakj@master Programy]$ srun -N 1 -n 1 SimpsonIntegral.exe

Total steps:                       4294967295
Processes run:                     1
Time elapsed:                      152.364s
Calculated integral:               4.46340952338899743523

[szumilakj@master Programy]$ srun -N 1 -n 4 SimpsonIntegral.exe

Total steps:                       4294967295
Steps per process:                 1073741823
Processes run:                     4
Time elapsed:                      45.410s
Calculated integral:               4.46340952252771305808

[szumilakj@master Programy]$ srun -N 2 -n 20 SimpsonIntegral.exe

Total steps:                       4294967295
Steps per process:                 214748364
Processes run:                     20
Time elapsed:                      10.891s
Calculated integral:               4.46340951736052371501

[szumilakj@master Programy]$ srun -N 5 -n 80 SimpsonIntegral.exe

Total steps:                       4294967295
Steps per process:                 53687091
Processes run:                     80
Time elapsed:                      2.731s
Calculated integral:               4.46340951735901558806
```

Figure 8. Comparison of calculation results obtained on the computer cluster and PC

Conclusions

This article discusses some aspects of the parallel programming used in applications for PCs and computer clusters. The general idea of programming for both technologies is similar – it is about fully utilizing the capabilities of modern computer systems, and in particular their multi-core/multi-threaded processors. Both technologies offer dedicated libraries with appropriate classes and their methods. One common feature is also the need to ensure synchronization between threads/processes – this is one of the key aspects defining the correctness of the solutions – as indicated in the part describing the presented applications. Of course, there are many more of these aspects and problems and it is impossible to mention all of them in such a short study.

The examples presented clearly indicate that implementing parallelism features requires programmers to have a specific approach already at the application design stage, in order to create the possibility of using the aforementioned parallel programming libraries. In the first application, an approach was used in which the image was divided into sections of pixels, and each of them was processed independently by dedicated threads. It should be noted that the data from individual sections of the image are independent of each other, so the processing can actually be carried out in a parallel manner. This not only increases the speed of calculations, but it also ensures the responsiveness of the main application thread, implementing the GUI interface. A similar parallelization scheme applies to the OpenMPI application, in which dedicated processes calculate independent sets of components of a numerical formula. The presented simulation results for different conditions of code distribution between cluster nodes show

the high effectiveness of the approach used in accelerating calculations. Of course, there are other additional aspects of numerical calculations – which could not be discussed due to the volume of the article – such as the problem of integration accuracy, development of the problem for integrals of functions of two variables or the problem of applications of the presented solution in specific engineering tasks. It is also worth pointing out the enormous scientific and educational benefits resulting from the access to a computer cluster. This allows planning dedicated classes for students, giving the opportunity to familiarize themselves with the physical structure of the cluster, management of its operating system, the system for managing computational tasks and finally, the implementation and launch of OpenMPI applications. It seems that this allows students to be equipped with specific knowledge and skills that can additionally enrich their possibilities of adapting to the requirements of the IT industry.

References

Birath, B., Ernstsson, A., Tinnerholm, J., Kessler, Ch. (2024). High-level programming of FPGA – accelerated systems with parallel patterns. *International Journal of Parallel Programming*, *52*, 253–273.

Collange, C., Defour, D., Graillat, S., Iakymchuk, R. (2015). Numerical reproducibility for the parallel reduction on multi – and many – core architectures. *Parallel Computing*, *49*, 83–97. https://doi.org/10.1016/j.parco.2015.09.001.

Flowers, B.H. (2000). *An introduction to numerical methods in C++*. Oxford University Press.

Herrmann, N., Kuchen, H., (2023). Distributed calculations with algorithmic skeletons for heterogeneous computing environments. *International Journal of Parallel Programming*, *51*, 172–185.

Hielscher, A., Bartel, S. (2024). Parallel programming of gradient-based iterative image reconstruction schemes for optical tomography. *Computer Methods and Programs in Biomedicine*, *73*(2), 101–113.

https://learn.microsoft.com/pl-pl/dotnet/standard/parallel-programming (2022).

https://www.open-mpi.org (2024).

Madhuri, A.J. (2020). *Digital image processing. An algorithmic approach*. PHI Learning, India.

Rainders, J. (2007). *Intel threading building blocks: outfitting C++ for multicore processor parallelism*. O'Reilly Media Inc. USA.

Robey, R., Zamora, Y. (2021). *Parallel and high performance computing*. Manning Shelter Island Publications Co., USA.

Szyszko, P., Smołka, J. (2018). Five Ways to Introduce Concurrency into Your C# Program (in Polish). *Journal of Computer Science Institute*, *6*, 62–67.