Krzysztof Bartyzel

Department of Computer Image Analysis,
John Paul II Catholic University of Lublin
Al. Racławickie 14, 20-950 Lublin, Poland
e-mail: kbartyzel@kul.lublin.pl

Algorithms optimization for the image processing and analysis by constructing parallel solutions

Abstract: This paper presents a concept of parallel programming in the context of image analysis and processing algorithms. It demonstrates an exact implementation of the issue of image filtration using the Microsoft .NET framework and the C# language. All technical aspects were subject to analysis. Presented are both theoretical considerations and nuances of implementation.
An experiment was also conducted which consisted in the creation of an appropriate program to demonstrate an example noise filter and the recording of performance time in the case of synchronous and parallel execution. The solution analysis was tested on a typical, average laptop and a server with high computing power. The results unanimously show that applying parallel algorithms can significantly improve the effectiveness of the hardware used.

Keywords: Parallel computing, image processing and analysis, Microsoft .NET Framework, multithread applications

1. Introduction

With the development of hardware capabilities comes the constantly increasing importance of perfecting the algorithms used. It might seem that, having increasingly powerful computers at one's disposal, the quality of constructed algorithms is no longer as significant. However we are dealing with the exact opposite phenomenon. Because the quantity of aggregated and analysed data is steadily increasing, there is an unending need for the optimisation of existing solutions. On the other hand we have been observing for decades a steady increase in computing power, however for more than ten years the primary method of increasing computing power has been the placement of increasingly numerous cores within a single central processing unit (CPU), enabling simultaneous computation. Even now the standard is computers capable of performing computations in 8 or 16 threads. Even popular game consoles (Xbox, PS) enable the use of multithread algorithms. The subject matter of parallel programming grows in significance, however the lack of appropriate algorithms is noticeable. The vast majority of scientific papers deals with theoretical matters, omitting aspects of implementation. Very infrequently can flowcharts be found, let alone pieces of source code. In the case of asynchronous algorithms the situation is even worse and left entirely to the audience.

The field of image processing and analysis develops at a very quick pace. A significant factor affecting it is certainly the development of technology enabling the acquisition of multiple high quality images over a short period of time. Also the applications of image analysis and processing are very wide: from facial and smile recognition in digital cameras, controlling a console or computer with the Kinect device to life-saving and military applications in unmanned aircraft assisting searches of difficult to reach areas.

Only a few years ago the mindset of „Power is free, but transistors are expensive" was very common, but now it seems to be closer to the truth that „Power is expensive, but transistors are »free«. That is, we can put more transistors on a chip than we have the power to turn on"[1].

Combining the above observations one can easily reach the conclusion that one of the best ways of shortening the time of image processing will be modifying existing algorithms so that they use as efficiently as possible the computing power of the computers available.

In this paper I would like to present the notion of parallel programming in the context of image analysis and processing algorithms. As an example I would like to demonstrate the implementation of a parallelised image filtration done using the Microsoft .NET framework and the C# language. In the end I will present the results of an experiment demonstrating the expected increase in efficiency.

2. Problem outline

Before jumping into details one should consider the main issues and problems. Realising any task in a multithread fashion requires answering a few standard questions, e.g.

- How to divide the main problem into multiple smaller subtasks.
- How numerous the subtasks should be.
- How will the subtasks be performed, both in the technological context (what implementation provided by the environment or operating system should be used) as well as the algorithmic context (how to divide the main tasks into subtasks so that the division is not computationally too complex and how to combine the results).
- What technical synchronisation mechanisms should be used to prevent negative effects of competition and to guarantee data consistency.

The above issues are universal enough that they are analysed in various forms in most papers and courses [2-7]. In the subsequent part of this paper an attempt is made to answer the above questions, and the best practices in the context of the solutions offered by the .NET framework and C# language are presented. The topic of parallel image processing has been subject to scientific consideration for many years. Papers worth recommending include [8-9]. However, in this paper I wish to present one specific implementation of the problem, along with detailed experiment results.

3. Detailed description of the solution

In the realisation of any multithread problem five elements can be specified:

a) Image processing variable initialisation block,
b) Multithread programming variable initialisation block,

c) Dividing the task into separate subtasks and assigning control to separate threads,
d) Awaiting completion of processing by all threads and analysis of returned results,
e) Disposing resources.

In the simultaneous concept elements b) and d) were absent, whereas c) was heavily modified. Taking into account the above considerations it is possible to create a prototype of a method which would realise all five elements. This method might assume as one of the input parameters a delegate (function pointer) realising a specific implementation of the appropriate effect realised within a subtask. At this stage the specificity of the realised problem does not yet matter. The exact same steps need to be taken in image analysis as well as analysis of other problems.

Listing 1 presents an example implementation of a method realising any image filter in a multithread fashion. As input parameters a bitmap of the original image, a function pointer realising a single instruction block and an additional parameter controlling the realised effect are provided. Obviously, when creating specific solutions one may opt out of providing a function pointer or specific execution parameters, however, then we lose the generic nature and versatility of the solution.

```
private static Bitmap AnyMultiThreadsFilter(
        Bitmap bitmapIn,
        WaitCallback filterProcedure,
        int maskSize)
{
    1. Image processing variable initialisation block

    2. Multithread programming variable initialisation block

    3. Dividing the task into separate subtasks and assigning control to separate threads

    4. Awaiting completion of processing by all threads and analysis of returned results

    5. Disposing resources

}
```

Listing 1. Elements of multithread image processing realisation

When reading the presented listing it is worth noting that variable initialisation has been divided into two areas. In the case of multithread programming we need to prepare mechanisms of code synchronisation [10-11]. In the case of both multithread and distributed programming the ability to return to synchronous (single-thread) processing is a vitally important element. One of the solutions that can be used at this point is a Blockade [12-13]. It is a construction which enables the halting of the main thread functioning until all side threads are confirmed ready for further work (or reach a defined point).

As stated earlier, the initial task is to declare and initialise the variables related to both image processing and subtask synchronisation. For the covered example this is shown on listing 2.

```
1.  Image processing variable initialisation block

Bitmap bitmapOut = new Bitmap(bitmapIn.Width, bitmapIn.Height,
PixelFormat.Format24bppRgb);
BitmapData bmInData = bitmapIn.LockBits(
new Rectangle(0, 0, bitmapIn.Width, bitmapIn.Height),
ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb);
BitmapData bmOutData = bitmapOut.LockBits(
new Rectangle(0, 0, bitmapOut.Width, bitmapOut.Height),
ImageLockMode.ReadWrite, PixelFormat.Format24bppRgb);

int stride = bmWeData.Stride;
IntPtr scanIn = bmWeData.Scan0;
IntPtr scanOut = bmWyData.Scan0;

int width = bitmapIn.Width;
int height = bitmapIn.Height;

int threadsCount = Environment.ProcessorCount;

2.  Multithread programming variable initialisation block

_waitHandles = new WaitHandle[threadsCount];
for (int i = 0; i<threadsCount; i++)
_waitHandles[i] = new AutoResetEvent(false);
```

Listing 2. Variable initialisation

Applying the .NET framework we have access to implementing a thread queue (ThreadPool class [14]). This is an exceptionally convenient solution, as it enables simple management of a group of threads. Simultaneously, any burdens resulting from the need to secure memory for new system objects have been limited to the absolute minimum. This is shown on listing 3.

```
3.  Dividing the task into separate subtasks and assigning control to separate threads

for (int i = 0; i<threadsCount; i++)
{
ThreadPool.QueueUserWorkItem(filterProcedure,
new{
yStart = i * height / threadsCount,
yStop = (i + 1) * height / threadsCount,
    width, stride, maskSize, scanIn, scanOut, waitHandle = _waitHandles[i]
  });
}
```

Listing 3: Task division and assigning control to separate threads

In the case of filters the creation of parallel algorithms is quite a charming task. The answer to the question of how many threads should be used and how to divide the image is usually obvious. To maximally utilise the computing power of the central processing unit (CPU) one should divide the image into as many parts as there are available cores. This means that each core will be responsible for the execution of one subtask. Figure 1 (The original "Lena" image is available as part of the USC SIPI Image Database) shows an example of dividing the analysed image. The image has been divided into four strips of similar size. It has been empirically tried and tested by author and it is easily shown through theoretical consideration that any other division is less optimal. Of course these are just theoretical considerations, in the real world we deal with significant factors of uncertainty and indeterminism. The covered situation assumes that the operating system aims to balance the load of each core and a single core being overloaded is a rare occurrence. Furthermore, an analysis of the individual cores' workload is of little help. The situation may change so dynamically that any estimates made before initiating the main computations can become irrelevant afterwards. Using the ThreadPool class we can rest assured that Microsoft .NET Framework takes care of proper subtask distribution. As per documentation, upon adding to the thread pool, the tasks are initiated with no further delay [14]. As an alternative to the ThreadPool class, use of the Task class can also be considered, as it offers very similar capabilities [15].

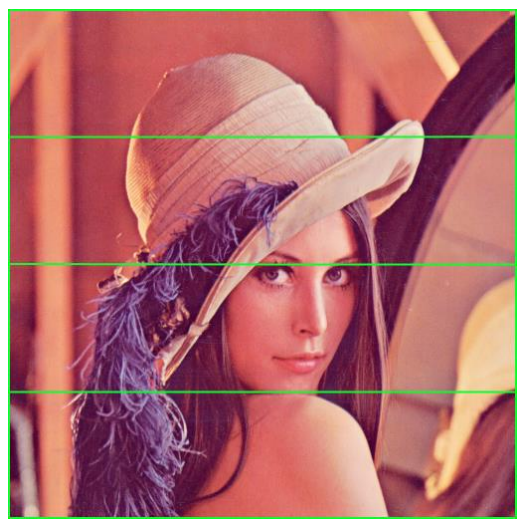

Figure 1. Dividing the object into four areas

Figure 2 shows a screenshot of the Task Manager during the execution of example computations. It is a distinctive element that for a short moment in time all cores were working at maximum load. The screenshot comes from a machine with the following CPU specification: Intel Xeon E5-2667 @2.9 GHz (2 processors), RAM: 32GB, OS: Windows Server 2008R2 Standard.

The final elements left to perform are the awaiting of the main thread for the realisation of all side tasks (listing 4) and the freeing of resources and returning of results (listing 5).

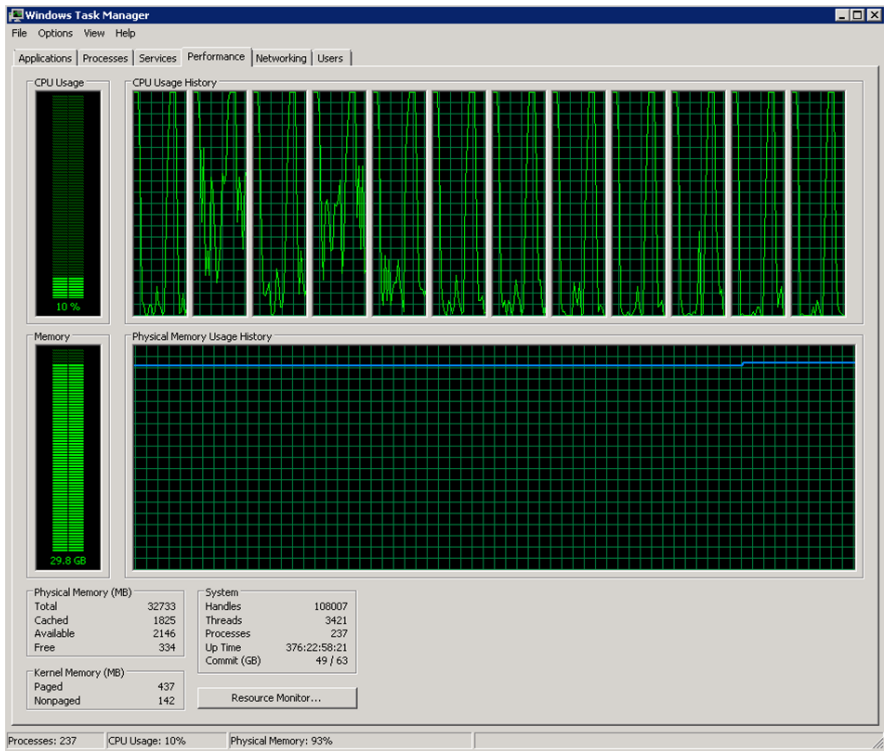

Figure 2: Simultaneous use of all available cores

```
4.  Awaiting completion of processing by all threads and analysis of returned results

foreach (WaitHandle waitHandle in _waitHandles)
{
waitHandle.WaitOne();
}
```

Listing 4. Blockade awaiting the completion of work by all threads

```
5.  Disposing resources

bitmapOut.UnlockBits(bmOutData);
bitmapIn.UnlockBits(bmInData);

return bitmapOut;
```

Listing 5. Disposing resources and returning results

Furthermore, to get a complete view of the problem and to simplify future implementations, listing 6 shows a skeleton of the function responsible for the execution of a subtask. The last line of the method is very important. It contains an instruction which enables notifying the main thread of work completion in the subtask.

```
private static voidKuwaharaFilterThreating(Object obj)
{
Type anonymousType = obj.GetType();

int yStart = (int)anonymousType.GetProperty("yStart").GetValue(obj, null);
int yStop = (int)anonymousType.GetProperty("yStop").GetValue(obj, null);
int width = (int)anonymousType.GetProperty("width").GetValue(obj, null);
int stride = (int)anonymousType.GetProperty("stride").GetValue(obj, null);
int maskSize = (int)anonymousType.GetProperty("maskSize").GetValue(obj, null);
IntPtr scanIn = (IntPtr)anonymousType.GetProperty("scanIn").GetValue(obj, null);
IntPtr scanOut = (IntPtr)anonymousType.GetProperty("scanOut").GetValue(obj, null);
AutoResetEvent waitHandle
        = (AutoResetEvent)anonymousType.GetProperty("waitHandle").GetValue(obj, null);

 Subtask implementation

waitHandle.Set();
}
```

Listing 6. Skeleton of a subtask processing method

4. Experiment

For the purposes of this paper an experiment was conducted, consisting in the creation of a program illustrating an example noise reduction filter (KuwaharaFilter [16]) and recording of runtimes for synchronous execution (single thread) and parallel execution (thread count depending on technical hardware capability).

To achieve even more interesting results, the demonstrated solutions were tested on both an average, ordinary laptop (CPU: Intel Core i5-3210M @2.50GHz, RAM: 6GB) and a server (CPU: Intel Xeon ES-2667 @ 2.90GHz, 2 processors, RAM: 32GB). In the experiment the filter window size was set to 7x7 and 23x23. Quite large images were used, at a resolution of 4256x2820 (12MP). Using less demanding settings meant that some of the computation times balanced on the edge of statistical error and the constant load of the CPU could impact the results. The experiment results are shown in Table 1.In 1-10 rows were placed particular results of the experiment while the two last rows provide cumulated results: the mean and standard deviation.It was carried out only 10 repetitions, but obtained concentration of results (based on standard deviations) leads to the conclusion that the received results are reliable and representative.For time measurement was usedthe built in .NET Framework System.Diagnostics.Stopwatchclass which allows to performaccurate executionelapsed times.

Table 1. Runtime comparison of synchronous and parallel executions (in milliseconds)

	Server				Laptop			
Mask size	23x23		7x7		23x23		7x7	
Threads count	12 hreads	1 thread	12 hreads	1 thread	4 threads	1 thread	4 threads	1 thread
1	7919	59789	701	7683	27840	58165	3379	7528
2	7940	59815	703	7663	26499	57937	3396	7422
3	6481	59841	706	7715	27958	57351	3316	7407
4	5658	59738	682	7653	28511	57528	3340	7313
5	7118	59764	889	7656	26348	57478	3291	7541
6	5669	59801	704	7672	27773	58093	3227	7324
7	5358	59804	695	7690	26222	58384	3286	7401
8	7785	59837	698	7669	26927	57784	3311	7346
9	6365	59841	696	7661	26507	57370	3298	7423
10	6666	59794	712	7660	26679	58404	3244	7571
AVG	**6696**	**59802**	**719**	**7672**	**27126**	**57849**	**3309**	**7428**
STD	**881,31**	**33,39**	**60,12**	**18,71**	**779,45**	**392,16**	**47,70**	**85,07**

5. Conclusions

It is easy to see that the increase in efficiency is significant. The case of the server solution resulted in a nearly 9-fold increase in efficiency (in the case of a 23x23 filter window) or over 10-fold (in the case of a 7x7 filter window). The weaker machine gave a less spectacular result of merely double efficiency. This leads to the obvious conclusion that with an increase of the core count the total solution efficiency also increases.In calculating the hypothetical efficiency growth Amdahl's law [17] can be used. It clearly shows that when creating a parallel version of the standard algorithms one cannot expect a linear increase of the performance. It is worth noting at this point the runtime in the case of the single core. One can conclude that if the software used is not adjusted to use multiple cores, then a purchase of a powerful hardware solution is not economically sound.

In the situation where we have the hardware capability, we should always consider transforming computationally complex portions of code so as to optimally utilise the processor architecture. One should keep in mind that not for all algorithms a parallel version can be developed. It should also be considered if it would bring any sizeable gain. The problem of subtask synchronisation can significantly impact the final appraisal of the improved solution.

It should also be noted that there exists the possibility of creating even more

efficient software (especially in the field of image processing or mathematical modelling). Using a graphics processing unit (GPU) and graphics libraries such as OpenGL or DirectX one can obtain much better results. However, this is a completely different technology, designed to solve entirely different issues, but which requires analysing similar problems. The most difficult problem is still the transformation of a synchronous algorithm into its asynchronous version.

Bibliography

1. Asanovic K., Bodik R. and others (2006) *The Landscape of Parallel Computing Research: A View from Berkeley*, Electrical Engineering and Computer Sciences University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html
2. Kumar V., Grama A., Gupta A., Karypis G. (1994) *Introduction to Parallel Computing, Design and Analysis of Algorithms* Benjamin-Cummings Publishing Co.
3. Intel® Developer Zone, *Courseware - Parallel Programming Basics* https://software.intel.com/en-us/courseware-parallel-programming-basics
4. Raghavendra P. (2011) *Multi core challenges and strategies* Department of Information Technology National Institute of Technology Karnataka Surathkal, https://software.intel.com/en-us/courseware/249625
5. Yang U.M., *A Parallel Computing Tutorial*, http://www.cise.ufl.edu/class/cis6930fa14pro/doc/IMA-PPtTutorial.pdf
6. Barney B., *Introduction to Parallel Computing* https://computing.llnl.gov/tutorials/parallel_comp
7. Jones J., *A Parallel Multigrid Tutorial* https://computing.llnl.gov/casc/linear_solvers/present.html
8. Merigot A., Petrosino A. (2008) *Parallel processing for image and video processing: Issues and challenges* Parallel Computing Volume: 34, Issue: 12, pp. 694-699.
9. Nicolescu C., Jonker P. (2002) *A data and task parallel image processing environment*, Parallel Computing Volume: 28, Issue: 7-8, pp. 945-965.
10. Doroshenko A.E. (1995) *Programming abstracts for synchronization and communication in parallel programs*, Parallel Computing Technologies Lecture Notes in Computer Science Volume 964, pp. 157-162.
11. Rauber T., Rünger G. (2013) *Parallel Programming Models for Multicore and Cluster Systems,* Springer Berlin Heidelberg.
12. Lubachevsky B.D. (1990) *Synchronization barrier and related tools for shared memory parallel programming*, International Journal of Parallel Programming, Volume 19, Issue 3, pp 225-250.
13. Jung I., Hyun J., Lee J., Ma J. (2001) *Two-Phase Barrier: A Synchronization Primitive for Improving the Processor Utilization*, International Journal of Parallel Programming Volume 29, Issue 6, pp. 607-627, Kluwer Academic Publishers-Plenum Publishers.
14. Micosoft Developer Network *Thread Pooling*, http://msdn.microsoft.com/en-us/library/h4732ks0.aspx
15. Micosoft Developer Network *Task Parallelism*, http://msdn.microsoft.com/en-us/library/dd537609(v=vs.110).aspx).

16. Kuwahara M., Hachimura K., Eiho S., and Kinoshita M. (1976) *Processing of RI-angiocardiographic images*, Digital Processing of Biomedical Images, K. Preston Jr. and M. Onoe, Editors. pp.187-202, New York: Plenum.
17. Amdahl G.M. *Validity of the single processor approach to achieving large scale computing capabilities*, AFIPS '67 (Spring) Proceedings of the April 18-20, 1967, spring joint computer conference, p. 483-485.