IMPLICATIONS OF PARALLEL COMPUTING IN THE OPTIMIZATION OF DIGITAL IMAGE PROCESSING ALGORITHMS

Bogdan Vasilescu*, PhD Eng. Gabriel Rădulescu

Petroleum-Gas University of Ploieşti, Romania
** e-mail: vasilescu@gmail.com*

Abstract

Digital image processing is one of the fastest growing fields of research nowadays, with applications ranging from satellite imagery, medical imaging and videophone to character recognition, motion detection and photo enhancement. Digital image processing allows the use of much more complex algorithms for image processing, and hence can offer both more sophisticated performance at simple tasks, and the implementation of methods which would be impossible by analog means.

The current paper offers a possible solution to the optimization problems that arise from working with digital image processing algorithms. It focuses on using parallel computing architectures to increase the performance of the algorithms, especially by using the OpenMP standard.

Keywords: digital image processing, optimization, parallel architectures, OpenMP

1. Introduction

The software dimension associated with digital image processing has known a spectacular development along with computing technology. The algorithms are now solidly based on mathematical grounds, giving birth to new approaches which are faster, more precise and can solve formerly unsolvable problems. Most of the times the increase in performance due to using a more efficient algorithm is considerable, reaching up to several sizes in scale. Therefore, faster

algorithms render many new digital image processing techniques applicable and significantly reduce the costs of the systems.

Processing digital images typically involves several filtering steps, some of which are time-consuming [1]. However, there is an interesting method to improve program performance through parallel computing. Lately, since there are many computers with multi processors or multi-core CPUs, parallel processing becomes widely available. Nevertheless, having multiple processing units on the hardware does not make existing programs necessarily run faster. Programmers must take the initiative to implement parallel processing capabilities in their programs to fully make the most of the hardware available. OpenMP is a set programming APIs which include several compiler directives and a library of support functions. It was first developed for use with FORTRAN and now it is available for C and C++ as well.

The current paper focuses on using OpenMP-based tools to show how can multithreading be implemented to improve filter performance on multiprocessor systems and/or processors that support Hyper-Threading Technology.

2. Types of Parallel Programming

Before beginning with OpenMP, it is important to know why parallel processing is needed. In a typical case, a sequential code will execute in a thread which runs on a single processing unit. Thus, if a computer has 2 processors or more (either two cores or one processor with HyperThreading technology), only a single processor will be used for execution, therefore wasting the other's processing power. Rather than letting the other processor sit idle (or process other threads from other programs) it can be used to speed up the algorithm.

Parallel processing can be divided into two groups, task based and data based [2].

• Task based: Divide different tasks to different CPUs to be

executed in parallel. For example, a Printing thread and a Spell Checking thread running simultaneously in a word processor. Each thread is a separate task.

• Data based: Execute the same task, but divide the workload on the data over several CPUs (for example, converting a color image to grayscale). We can convert the top half of the image on the first CPU, while the lower half is converted on the second CPU (or as many CPUs you have), thus processing in half the time.

There are several methods to do parallel processing

• Using MPI: Message Passing Interface - MPI is most suited for a system with multiple processors and multiple memory modules (for example, a cluster of computers with their own local memory). MPI can be used to divide the workload across this cluster and merge the results when it is finished.

• Using OpenMP: OpenMP is suited for shared memory systems like desktop computers. Shared memory systems are systems with multiple processors but each are sharing a single memory subsystem. Using OpenMP is like writing smaller threads and letting the compiler manage them.

• Using SIMD intrinsic: Single Instruction Multiple Data (SIMD) has been available on mainstream processors such as Intel's MMX, SSE, SSE2, SSE3, Motorola's (or IBM's) Altivec and AMD's 3DNow!. SIMD intrinsic are primitive functions to parallelize data processing on the CPU register level. For example, the addition of two unsigned chars will take the whole register size, although the size of this data type is just 8-bit, leaving 24-bit in the register to be filled with 0 and wasted. Using SIMD (such as MMX), 8 unsigned chars can be loaded (or 4 shorts or 2 integers) and executed in parallel on the register level.

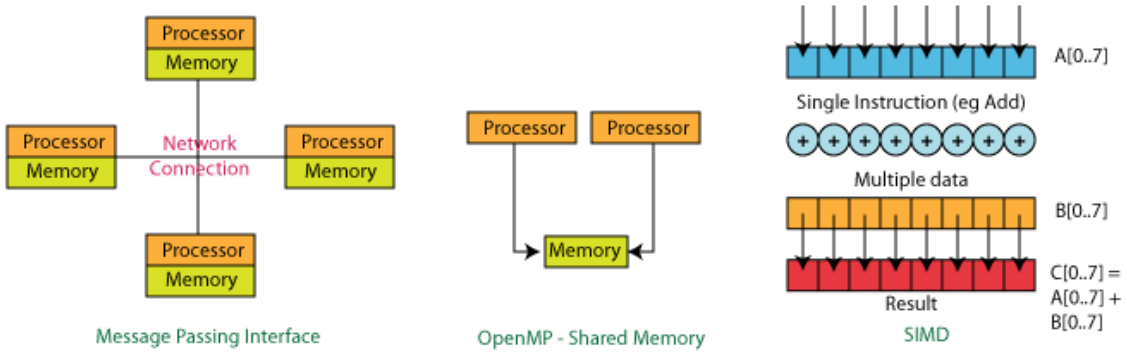

Figure 1: Overview of MPI, OpenMP and SIMD architectures

3. OpenMP

The OpenMP standard [4] is a specification for a portable implementation of shared memory parallelism in FORTRAN, C, and C++. The specification provides a set of compiler directives and runtime library routines that extend FORTRAN, C, and C++ to achieve shared memory parallelism. OpenMP language extensions include work-sharing constructs, data environment and synchronization. The standard also includes a callable runtime library with accompanying environment variables.

OpenMP defines a set of C pre-processor pragmas (or directives for FORTRAN), which describe parallelism to the compiler (OpenMP has a limited ability to express task parallelism as well). OpenMP-compliant compilers are available for most operating systems, which makes OpenMP quite portable. Most important, however, OpenMP is compact and often non-intrusive and threading existing serial code rarely requires significant code modifications.

OpenMP uses the fork-and-join parallelism model [6]. In fork-and-join, parallel threads are created and branched out from a master thread to execute an operation and will only remain active until the operation has finished, then all the threads are destroyed, thus leaving only one master thread. The process of splitting and joining the threads including synchronization for the end result are handled by OpenMP.

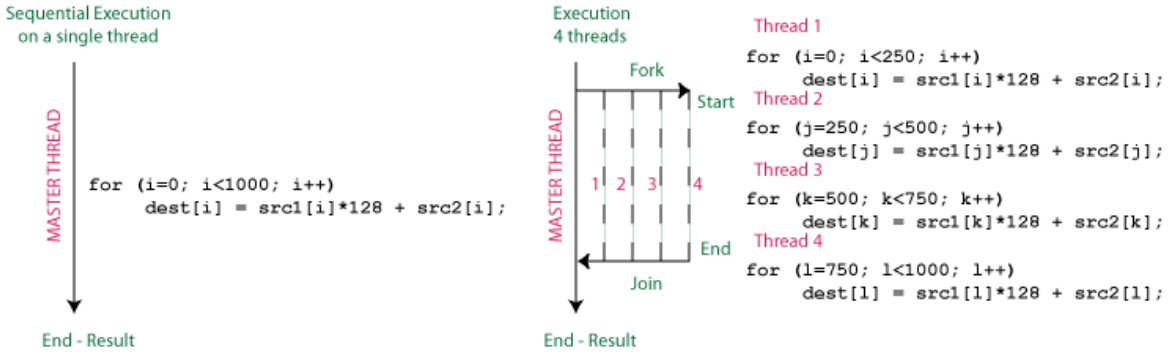

Figure 2: The fork-and-join parallelism model

A typical question is how many threads are needed for a specific problem [5]. The number of threads required to solve a problem is generally limited to the number of CPUs available. As seen in the Fork-and-Join figure above, whenever threads are created, a little time is taken to create a thread and later to join the end result and destroy the threads. When the problem is small, and the number of CPUs are less than the number of threads, the total execution time will be longer (slower) because more time has been spent to create threads, and later switch between the threads (due to preemptive behaviour) then to actually solve the problem. Whenever a thread context is switched, data must be saved/loaded from the memory. This takes time.

Therefore, since all the threads will be executing the same operation (hence the same priority), one thread is sufficient per CPU (or core). The more CPUs available, the more threads can be created.

Most compiler directives in OpenMP use the Environment Variable OMP_NUM_THREADS to determine the number of threads to create. The number of threads can be controlled with the following C++ functions:

```
// Get the number of processors in this system
int iCPU = omp_get_num_procs();
// Now set the number of threads
omp_set_num_threads(iCPU);
```

4. Analysis and results

Digital image processing uses specific algorithms and techniques, also known as filters. Most of the times, image processing filters require time-consuming iterations. The current study focuses on finding a solution to reduce the execution time of such iterations, namely the for-loops and the double for-loops, which are most commonly found in such filters. In order to optimize the crossing of the loops the workload can be distributed to multiple threads running on multiple cores.

4.1. Parallel for Loop

The following is a C++ code to convert a 32-bit Colour (RGBA) image to 8-bit Greyscale image, a common operation used in image processing algorithms [3]. The sequence demonstrates the use of a simple parallel for loop.

```
// pDest is an unsigned char array of size width * height
// pSrc is an unsigned char array of size width * height * 4 (32-bit)
// Use pragma for to make a parallel for loop
omp_set_num_threads(threads);
#pragma omp parallel for
for(int z = 0; z < height*width; z++)
{
    pDest[z] = (pSrc[z*4+0]*3735 +
               pSrc[z*4 + 1]*19234+ pSrc[z*4+ 2]*9797)>>15;
}
```

The #pragma omp parallel for directive will parallelize the for-loop according to the number of threads set.

The following plot illustrates the performance gained for a 3264x2488 image on a 1.66GHz Dual Core system (Fig.3-a). By executing the problem using 2 threads on a dual-core CPU, the time has been cut by half (the speedup has doubled). However, as the number of threads is increased, the performance does not improve further due to increased time to fork and join.

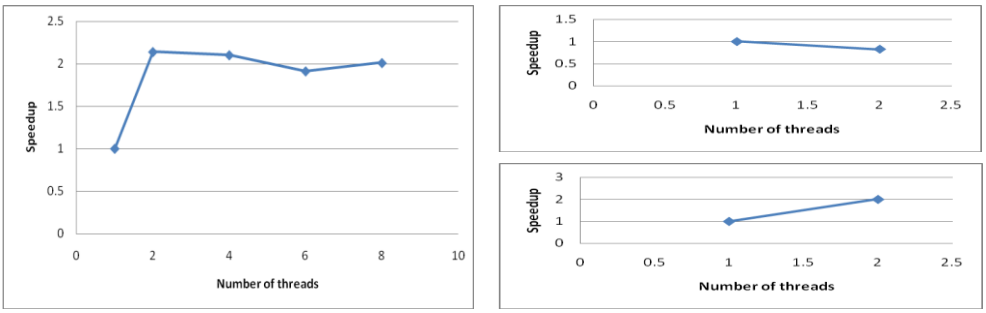

Figure 3: Speedup of parallel for-loop; b,c. Speedup of the two solutions for parallel double for-loop

4.2. Parallel double for loop

The same problem above (converting color to grayscale) can also be written in a double for-loop way, like in the C++ sequence below.

```
      for(int y = 0; y < height; y++)
           for(int x = 0; x< width; x++)
                      pDest[x+y*width] = (pSrc[x*4 + y*4*width + 0]*3735 +
pSrc[x*4 + y*4*width + 1]*19234+ pSrc[x*4 + y*4*width + 2]*9797)>>15;
```

In this case, there are two solutions. The first makes the inner loop parallel using the parallel for directive. When 2 threads are being used, the execution time has actually increased (Fig.3-b). This is because for every iteration of y, a fork-join operation is performed, which eventually leads to the increased execution time.

```
      for(int y = 0; y < height; y++) {
           #pragma omp parallel for
           for(int x = 0; x< width; x++) {
                      pDest[x+y*width] = (pSrc[x*4 + y*4*width + 0]*3735 +
pSrc[x*4 + y*4*width + 1]*19234+ pSrc[x*4 + y*4*width + 2]*9797)>>15;
                      }
      }
```

For the second solution, instead of making the inner loop parallel, the outer loop is the better choice. Here, another directive is introduced - the private directive [7]. The private clause directs the compiler to make variables private so multiple copies of a variable do not execute. In this case, the execution time did indeed get reduced by half (Fig.3-c).

```
      int x = 0;
      #pragma omp parallel for private(x)
      for(int y = 0; y < height; y++) {
```

```
    for(x = 0; x< width; x++)      {
     pDest[x+y*width] = (pSrc[x*4 + y*4*width + 0]*3735 + pSrc[x*4 +
y*4*width + 1]*19234+ pSrc[x*4 + y*4*width + 2]*9797)>>15;
    }
}
```

5. Conclusions

This paper presented a possible optimization solution to digital image processing filters. The method uses commonly available parallel computing architectures, like the ones in multiple core CPUs, in order to divide the workload over several processors by assigning it to several execution threads.

The most promising results have been obtained with a two-threaded approach on a dual-core processor where the execution time of the algorithms was reduced by half. Therefore, the optimum number of threads to be used is equal to the number of CPUs available.

Since the method proved to be viable, further research will be done by the authors in order to study the behavior of the algorithms on other multiple core processors (quad-core processors and so on).

References

[1] Gonzales, Woods, *Digital image processing*, Prentice Hall, 2008

[2] Malik, D. S., *C++ Programming: Program Design Including Data Structures*, Course Technology, 2008.

[3] Miller, A., Ford, L. F., *Microsoft Visual C++ 2005 Express Edition Programming for the Absolute Beginner*, Course Technology, 2005.

[4] OpenMP Application Program Interface, May 2008

[5] Gabb, H.A., Magro, B., *Faster Image Processing with OpenMP*, Dr.Dobb's Portal, 2004

[6] Begin Parallel Programming with OpenMP, http://www.codeproject.com/KB/cpp/BeginOpenMP.aspx

[7] 32 OpenMP traps for C++ developers, http://www.codeproject.com/KB/cpp/32_OpenMP_traps.aspx