Multicore Image Processing with OpenMP

Greg Slabaugh, Richard Boyes, Xiaoyun Yang

One of the recent innovations in computer engineering has been the development of multicore processors, which are composed of two or more independent cores in a single physical package. Today, many processors, including digital signal processor (DSP), mobile, graphics, and general-purpose central processing units (CPUs) [1] have a multicore design, driven by the demand of higher performance. Major CPU vendors have changed strategy away from increasing the raw clock rate to adding on-chip support for multi-threading by increases in the number of cores; dual- and quad-core processors are now commonplace. Signal and image processing progammers can benefit dramatically from these advances in hardware, by modifying single-threaded code to exploit parallelism to run on multiple cores.

This article describes the use of OpenMP (Open Multi-Processing) to multi-thread image processing applications to take advantage of multicore general purpose CPUs. OpenMP is an extensive and powerful application programming interface (API), supporting many functionalities required for parallel programming. The purpose of this article is to provide a high level overview of OpenMP, and present simple image processing operations to demonstrate the ease of implementation and effectiveness of OpenMP. More sophisticated applications could be built on similar principles.

OPENMP

Historically, a key challenge in parallel computing has been the lack of a broadly supported, simple to implement parallel programming model. As a result, numerous vendors provided different models, with often mixed degrees of complexity and portability. Software programmers subsequently found it difficult to adapt applications to take advantage of multicore hardware advances.

OpenMP was designed to bridge this gap, providing an industry standard, parallel programming API for shared memory multi-processors, including multicore processors. A vendor-independent OpenMP Architecture Review Board (ARB), which includes most of the major computer manufacturers, oversees the OpenMP standard and approves new versions of the specification. Support for OpenMP is currently available in most modern Fortran and C/C++ compilers as well as numerous operating systems, including Microsoft Windows, Linux, and Apple Macintosh OS X. Version 1.0 of OpenMP was released in 1997. The latest version, 3.0, was released in 2008. Please see the official OpenMP website [2] for the full specification, list of compilers supporting the standard, and reference documents.

We should note that OpenMP is certainly not the only way of achieving parallelism on multicore systems. Other implementation models, such as CILK, Pthreads, and MPI [3, 4] exist and may be a good choice depending on the hardware, application, and the preference of the programmer. In our experience, OpenMP has the advantages of being exceedingly simple to learn and implement, in addition to being powerful and well suited to modern processor architectures.

USING OPENMP

OpenMP works as a set of pre-processor directives, runtime-library routines, and environment variables provided to the programmer, who instructs the compiler how a section of code can be multi-threaded. In Fortran, the directives appear as comments, while in C/C++ they are implemented as pragmas. In this way, compilers that do not support OpenMP will automatically ignore OpenMP directives, while compilers that do support the standard will process, and potentially optimize the code based on the directives. Since the OpenMP API is independent of the machine/operating system, properly written OpenMP code for one platform can easily be recompiled and run on another platform. However, in this article, we present C++ code examples that were compiled using Microsoft Visual C++ 2005 and executed in Windows XP. Using this compiler, one can enable OpenMP in the project settings (Configuration Properties→C/C++→Language→OpenMP Support) and include omp.h.

An OpenMP application always begins with a single thread of control, called the *master thread*, which exists for the duration of the program. The set of variables available to any particular thread is called the thread's *execution context*. During execution, the master thread may encounter parallel regions, at which the master thread will fork new threads, each with its own stack and execution context. At the end of the parallel region, the forked threads will terminate, and the master thread continues execution. Nested parallelism, for which forked threads fork further threads, is supported.

0.0.1 Loop level parallelism

As mentioned above, parallelism is added to an application by including pragmas, which, in C++, have the following form:

```
#pramga omp <directive> [clauses]
```

There are numerous directives, but this article focusses on the `parallel for` directive, which offers a simple way to achieve loop-level parallelism, often existing in signal and image processing algorithms. The optional clauses modify the behavior of the directive.

The parallelization of loops is the most common use of OpenMP. Consider the following code, which computes a sine wave with amplitude *A* and frequency *w*:

```
for (int n=0; n<N; n++)
    x[n] = A*sin(w*n);
```

With OpenMP, this code can be trivially parallelized as

```
#pragma omp parallel for
for (int n=0; n<N; n++)
    x[n] = A*sin(w*n);
```

Here, the directive instructs the compiler that the next `for` loop is to be parallelized. The compiler will then distribute the work among a set of forked threads. If we assume that there are four threads forked, and $N = 100$, then the iterations may be spread amongst the processors such that iterations $0-24$ are given to thread 1, iterations $25-49$ are given to thread 2, iterations $50-74$ are given to thread 3, iterations $75-99$ are given to thread 4. Such allocation of work assumes static scheduling, which will be discussed below. The four threads will run simultaneously. If a forked thread completes its work before any other forked thread, it will block. Once all forked threads complete their work, the master thread then resumes execution. Note the simplicity of using OpenMP – the loop was parallelized with a *single line* of code.

The code in the above example was easily parallelized because it does not contain loop dependencies, which means the compiler can execute the loop in any order. Consider a modification of this loop:

```
x[0] = 0;
for (int n=1; n<N; n++)
    x[n] = x[n-1] + A*sin(w*n);
```

The loop is no longer trivially parallelized, as the computation of the `x[n]` now depends on `x[n-1]`. Thread 2 may start by computing `x[25]`, which depends on `x[24]`. However, thread 1 might not yet have computed `x[24]`, resulting in a run-time error. The programmer must be careful to ensure the parallelized loop is free of loop dependencies, as the compiler does not check this. As a result of such dependencies, some algorithms require additional coding to remove the dependencies [4] to render them amenable to parallelization.

0.0.2 Variable scope

As mentioned above, every thread has its own execution stack that contains variables in the scope of the thread. When parallelizing code, it is very important to identify which variables are shared between the threads, and which are private. In the parallelized sine wave example above, the variables x, A, w, and N were shared, while n was private; that is, each thread has its own n but shares all the other variables.

OpenMP provides explicit constructs to specify shared and private variables in the execution stack. By default, all variables are shared, *except*

1. The loop index
2. Variables local (declared within) the loop
3. Variables listed in private clauses

One can explicitly assign shared and private variables using directive clauses:

```
#pragma omp parallel for \
private(n) \
shared(A, x, w)
for (int n = 0; n < N; n++)
    x[n] = A * sin(w * n);
```

Copies of the variables in the private clause will be placed into each thread's execution context. Note however that any variable in a private clause is initially undefined. This can lead to coding errors. For example, consider

```
int x = 5;
#pragma omp parallel for private(x)
for (int n = 0; n < N; n++)
    // The value of x is undefined
```

in this case, the `firstprivate` clause can be used to copy the value of a variable to the execution stack of each thread:

```
int x = 5;
#pragma omp parallel for firstprivate(x)
for (int n = 0; n < N; n++)
    // The value of x is 5 for each thread
```

0.0.3 Scheduling

Earlier we mentioned static scheduling, which divides work of the loop evenly among the different threads. Static scheduling is the default work sharing construct and works well for *balanced* loops that have a relatively constant cost per iteration. However, some loops are unbalanced, with some iterations taking much longer than others. For such loops, static scheduling is not ideal, as fast threads will complete their work early and block until slow threads have completed their work. With OpenMP, it is possible to specify the scheduling mechanism (for example, using the `static` or `dynamic` clause). In a dynamic schedule, the number of iterations for each thread can vary depending on the workload. When free, each thread requests more iterations until the loop is complete. Dynamic scheduling is more flexible but does add additional overhead in coordinating the distribution of work amongst the threads. Later, we will show an example where dynamic scheduling makes a significant difference in runtime of a parallelized image processing algorithm.

By default, the system will decide how many threads to fork during runtime. The number of spawned threads can be retrieved using `integer omp_get_num_threads(void)`. In addition, the number of threads may be set using `omp_set_num_threads(integer)` or by using an environment variable, `OMP_NUM_THREADS`.

APPLICATIONS OF IMAGE PROCESSING USING OPENMP

In the previous section, we provided an introduction to OpenMP and a short description of how one may achieve loop level parallelization using the `parallel for` pragma. In this section, we demonstrate examples that show the ease and power of OpenMP for image processing. The examples are, by design, simple so that the principles can be easily demonstrated.

IMAGE WARPING

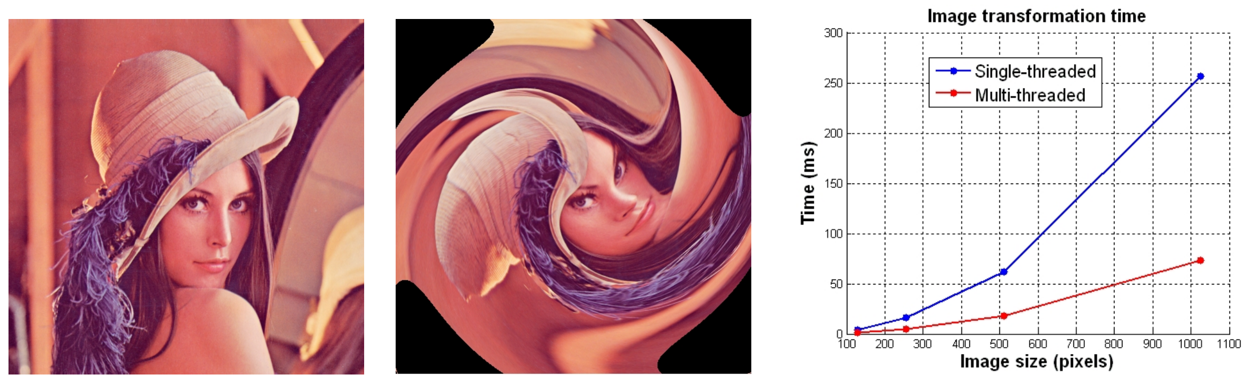
An image warp is a spatial transformation of an image, and is commonly found in photo-editing software as well as registration algorithms. In this example, we apply a “twist” transformation of the form

$$\begin{aligned} x' &= (x - c_x)\cos\theta + (y - c_y)\sin\theta + c_x \\ y' &= -(y - c_y)\sin\theta + (y - c_y)\cos\theta + c_y \end{aligned} \qquad (1)$$

where $[c_x, c_y]^T$ is the rotation center, $\theta = r/2$ is a rotation angle that increases with radius r, $[x,y]^T$ is the point before transformation, and $[x', y']^T$ is the point after transformation. The effect of this image transformation in shown in Figure 1 (b).

We implemented this transformation using OpenMP; the code listing appears in Listing 1. In the code, both the original image and transformed image are shared variables, along with the width and height of the image. The code loops over the pixels $[x', y']^T$ of the transformed image, mapping them back to the original image using the inverse transform of Equation 1. In the original image, the pixels are bilinearly interpolated. Each thread requires its own variables for `x`, `y`, `index`, `radius`, `theta`, `xp`, and `yp`. Since these variables are initialized within the parallelized code, we use the shared clause. OpenMP will multithread the outer loop (over `yp`) using static scheduling.

On a 512 x 512 image, and using a quad-core 2.4 GHz CPU, the single-threaded code requires 62.2 milliseconds (ms) to process the image, while the multi-threaded code requires 17.7 ms, corresponding to a 3.5x speedup. In Figure 1 (c), we present a plot showing the speedup as a function of image size (measured in one dimension of the square image). The code, very easily multi-threaded, achieves an excellent speedup when executed on multiple cores.



(a) (b) (c)

Figure 1: Example of a twist transformation applied to an image.

Listing 1: Parallelized code for the image warp.

```
int index, xp, yp, tx = width / 2, ty = height / 2;
float x, y, radius, theta, PI = 3.141527f, DRAD = 180.0f / PI;
#pragma omp parallel for \
shared(inputImage, outputImage, width, height) \
private(x, y, index, radius, theta, xp, yp)
for (yp = 0; yp < height; yp++) {
  for (xp = 0; xp < width; xp++) {
    index = xp + yp * width;
    radius = sqrtf((xp - tx) * (xp - tx) + (yp - ty) * (yp - ty));
    theta = (radius / 2) * DRAD;
    x = cos(theta) * (xp - tx) - sin(theta) * (yp - ty) + tx;
    y = sin(theta) * (xp - tx) + cos(theta) * (yp - ty) + ty;
    outputImage[index] = BilinearlyInterpolate(inputImage, width, height, x, y);
  }
}
```

MATHEMATICAL BINARY MORPHOLOGY

Mathematical morphology was originally developed for binary images and later was extended to grayscale images. Morphological operations are widely used in image segmentation, noise removal and many other applications, and employ a structuring element to probe an image and create an output image [5]. At its core, mathematical morphology has two basic operations: erosion and dilation. Erosion and dilation of an image/set X by a structuring element B are defined in Equation 2,

$$\begin{aligned} Erosion_B(X) &= \{x \mid B_x \subseteq X\} \\ Dilation_B(X) &= \{x \mid B_x \cap X \neq \emptyset\} \end{aligned} \tag{2}$$

where B represents structuring element, B_x denotes B centered at x. The result of erosion by B can be explained as the locus of points hit by the center of B when B moves entirely inside X. The result of dilation by B are the locus of the points covered by B when the center of B moves inside X. Other operations can be defined using combinations of erosion and dilation. For example, a closing is defined as a dilation followed by an erosion.

We implemented binary morphology erosion and dilation using OpenMP; the erosion code is listed in Listing 2 and the dilation code is similar. At each pixel, the function `FullFit` checks if the structuring element entirely fits into foreground binary region of the input image, as defined by Equation 2. Here, OpenMP will multi-thread the outer loop (over `y`) using dynamic scheduling.

Listing 2: Parallelized code for binary erosion.

```
int x, y;
#pragma omp parallel for \
shared(inputImage, outputImage, structuringElement, width, height) \
private(x, y) schedule(dynamic)
for (y = 0; y < height; y++) {
  for (x = 0; x < width; x++) {
        int index = x + y*width;
        if (inputImage[index]) {
            if (FullFit(inputImage, x, y, structuringElement))
                outputImage[index]=1;
            else
                outputImage[index]=0;
        }
  }
}
```

On a 512 x 512 image, and using a quad-core 2.4 GHz CPU, for a closing operation by a disk structure element with radius 15 pixels, the single-threaded code requires 79.4 ms to process the image, while the multi-threaded code requires 33.9 ms, corresponding to a 2.34x speedup. Figure 2 (a) and (b) illustrates the input image and the closed result, respectively. In Figure 2 (c), we present a plot showing the computation time as a function of structure element size (in radial pixels).

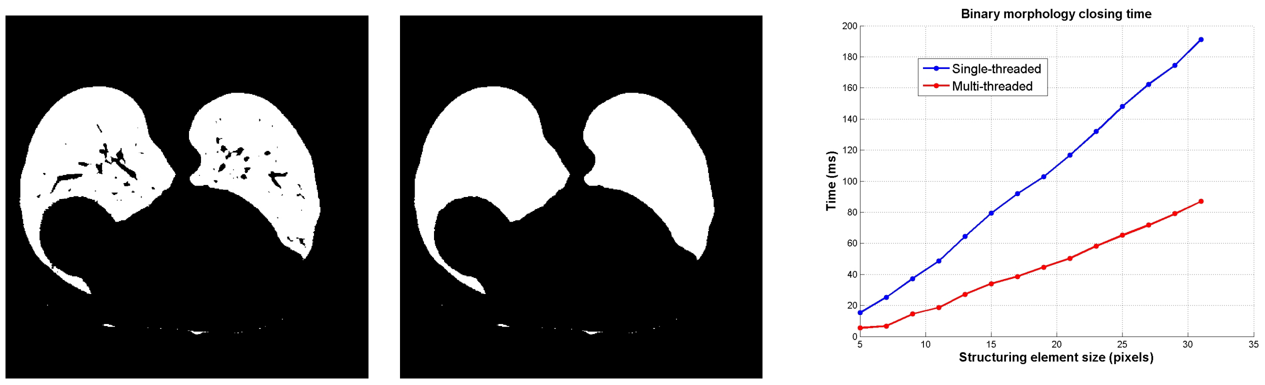
Binary morphology is a good example where the dynamic scheduling is helpful, as the same example scheduled statically requires 95.1 ms. In this example, the workload for a thread is unbalanced by the shape of the input. There may be large differences in the number of foreground pixels within the part of the image allocated to each thread, meaning dynamic scheduling is a better choice for how to distribute the work.

MEDIAN FILTERING

Median filtering is a commonly applied non-linear filtering technique that is particularly useful in removing speckle and salt and pepper noise [6]. Simply put, an image neighborhood surrounding each pixel is defined, and the median value of this neighborhood is calculated and is used to replace the original pixel in the output image:

$$I_{med}[x,y] = median\left(I_{orig}[i,j], i,j \in nbor[x,y]\right) \tag{3}$$

In this example we choose a square neighborhood around each pixel, defined using the halfwidth of the neighborhood, i.e., for a halfwidth of n, the number of pixels in the neighborhood would be $(2n+1))^2$. At each pixel, the function `GetNbors` retrieves the neighbors; any neighbors that lie outside the image domain are assigned to be that of the nearest pixel within the image boundary. These neighbors are then sorted using the C++ STL sort function and the median selected.



(a) (b) (c)

Figure 2: Example of a morphological closing applied to an image.

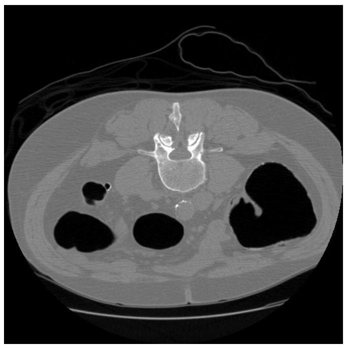
Listing 3: Parallelized code for median filtering using a halfwidth of three.

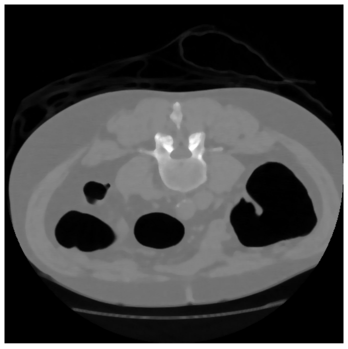
```
int x, y, halfWidth, nborSize;
PixelType nbors[MAX_NBOR_SIZE];
halfWidth = 3;
nborSize = 2*halfWidth + 1;
nborSize *= nborSize;
#pragma omp parallel for \
shared(inputImage, outputImage, structuringElement, width, height) \
private(x, y, nbors) firstprivate(halfWidth, nborSize) schedule(static)
for (y = 0; y < height; y++) {
  for (x = 0; x < width; x++) {
     GetNbors(inputImage, x, y, width, height, halfWidth, nbors);
     sort(&nbors[0], &nbors[nborSize]);
     int index = x + y*width;
     outputImage[index] = nbors[nborSize/2];
  }
}
```

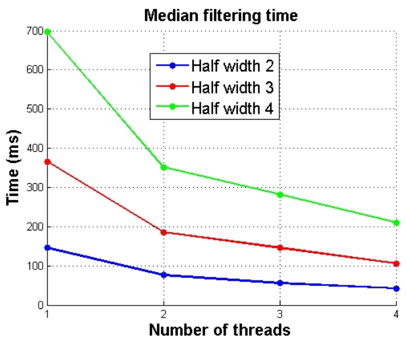
On a 512×512 medical image, and using a quad-core 2.4GHz CPU we show the result of median filtering using a half width of 3, i.e., the number of neighbors $= (2 \times 3 + 1)^2 = 49$ in Figure 3 (a) and (b). In Figure 3 (c) we demonstrate the linear acceleration of the median filtering using different numbers of threads on different sizes of neighborhoods.

NORMALIZATION

Normalization is a process whereby the pixel intensities are scaled linearly. The linear scale factors can include those that will give the normalized image a prescribed minimum and maximum, or, say, a new intensity average. This is usually performed to bring the intensities into a standard range. In our example, we wish to alter the pixel intensities so that they have a mean intensity of zero and a standard deviation of unity, which is common when analyzing images from a statistical viewpoint. This is achieved by first calculating the mean and standard deviation of the pixel intensities, and then scaling them as


(a)


(b)



(c)

Figure 3: Example of median filtering applied to an image.

$$I_{new} = \frac{(I_{orig} - \mu)}{\sigma}, \tag{4}$$

where μ and σ are the mean and standard deviation of the image intensities. To parallelize the estimation of μ and σ, we use an OpenMP *reduction variable*, which indicates that a variable has to be accumulated from all the threads in some way within the parallel loop. In our example the reduction performs a sum (+), although there are several other types, including subtraction (-), product (*), and bitwise and logical operations. Note that in Fortran minimum and maximum can also be used in reduction clauses as they are built in functions, whereas in C++ they are not.

Listing 4: Parallelized code for rescaling data

```
int index;
int n = width*height;
float mean = 0.0, var = 0.0, svar, std;

// Calculate the mean of the image intensities
#pragma omp parallel for \
shared(inputImage, n) reduction(+:mean) \
private(index) schedule(static)
for (index = 0; index < n; index++) {
  mean += (float)(inputImage[index]);
}
mean /= (float)n;

// Calculate the standard deviation of the image intensities
#pragma omp parallel for \
shared(inputImage, n) reduction(+:var) \
private(index, svar) schedule(static)
for (index = 0; index < n; index++) {
  svar = (float)(inputImage[index]) - mean;
  var += svar*svar;
}
var /= (float)n;
std = sqrtf(var);

// Rescale using the calculated mean and standard deviation
```

```
#pragma omp parallel for \
shared(inputImage, outputImage, n) private(index) \
firstprivate(mean, std) schedule(static)
for(index = 0; index < n; index++) {
  outputImage[index] = (inputImage[index] - mean)/std;
}
```

To test the code, we renormalized the image from Figure 3 (a) for different threads, and the processing took 5.6 ms for one thread, and 1.6 ms for four threads, demonstrating a near factor of four acceleration.

OUTLOOK

This article has only scratched the surface of the capabilities of OpenMP for parallelized signal and image processing on multicore machines. More advanced features, such as synchronization and parallel regions, extend the basic functionalities described here. However, with simple use of the OpenMP `parallel for` directive, it is remarkably easy for the signal processing programmer to achieve loop level parallelism. As general purpose CPUs continue to advance, OpenMP will continue provide an uncomplicated way to harness the increasing power of multicore architectures.

AUTHORS

Greg Slabaugh (greg.slabaugh@medicsight.com) is the Head of Research and Development (R&D) at Medicsight PLC.

Richard Boyes (richard.boyes@medicsight.com) is a Scientific R&D Engineer at Medicsight PLC.

Xiaoyun Yang (xiaoyun.yang@medicsight.com) is a Senior Scientific R&D Engineer at Medicsight PLC.

References

[1] G. Blake, R. G. Deslinski, and T. Mudge, "A survey of multicore processors: A review of their common attributes," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 26–37, 2009.

[2] `http://www.openmp.org`.

[3] H. Kim and R. Bond, "Multicore software technologies: A survey," *IEEE Signal Processing Magazine*, vol. 26, no. 6, pp. 80–89, 2009.

[4] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, first ed., 2001.

[5] P. Soille, *Morphological Image Analysis*. Springer-Verlag, second ed., 2003.

[6] A. Jain, *Fundamentals of Digital Image Processing*. Prentice Hall, first ed., 1989.