ISSN: 0975-766X
CODEN: IJPTFI
Available Online through *Research Article*
www.ijptonline.com

MORPHOLOGICAL IMAGE PROCESSING WITH OPENMP

R.Jagadeeswari[1]
[1]Research Scholar, Bharath Institute of Higher Education and Research, Bharath University, Chennai-73.
Email: jagajr_hodit@rediffmail.com

Received on: 15.10.2016 *Accepted on: 22.11.2016*

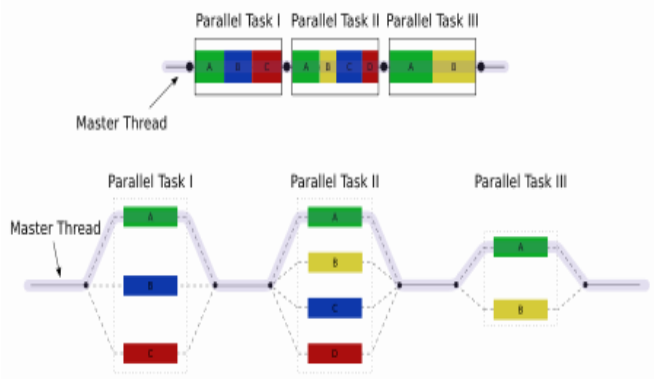
Abstract

Mathematical morphology (MM) is a theory and technique for the analysis and processing of geometrical structures, based on set theory, lattice theory, topology, and random functions. MM is most commonly applied to digital images, but it can be employed as well on graphs, surface meshes, solids, and many other spatial structures. Topological and geometrical continuous-space concepts such as size, shape, convexity, connectivity, and geodesic distance, can be characterized by MM on both continuous and discrete spaces. MM is also the foundation of morphological image processing, which consists of a set of operators that transform images according to the above characterizations. MM was originally developed for binary images, and was later extended to grayscale functions and images. The subsequent generalization to complete lattices is widely accepted today as MM's theoretical foundation. Open MP (Open Multi-Processing) is an API (application programming interface) that supports multi-platform shared memory multiprocessing programming in C, C++, and Fortran, on most processor architectures and operating systems, including Linux, Unix, AIX, Solaris, Mac OS X, and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. Open MP is managed by the non-profit technology consortium Open MP Architecture Review Board (or Open MP ARB), jointly defined by a group of major computer hardware and software vendors, like AMD, IBM, Intel, Cray, HP, Fujitsu, NVIDIA, NEC, Microsoft, Texas Instruments, VMware, Oracle Corporation, and more. In our experience, Open MP achieving parallelism in morphological.

Keywords: Multi-threading; Parallelizable; Binary morphology; Erosion; Dilation.

Introduction: An illustration of multithreading where the master thread forks off a number of threads which execute blocks of code in parallel. Open MP is an implementation of multithreading, a method of parallelization whereby the

master "thread" (a series of instructions executed consecutively) "forks" a specified number of slave "threads" and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors. The section of code that is meant to run in parallel is marked accordingly, with a preprocessor directive that will cause the threads to form before the section is executed. Each thread has an "id" attached to it which can be obtained using a function (called omp_get_thread_num()). The thread id is an integer, and the master thread has an id of "0". After the execution of the parallelized code, the threads "join" back into the master thread, which continues onward to the end of the program. By default, each thread executes the parallelized section of code independently. "Work-sharing constructs" can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using Open MP in this way. The runtime environment allocates threads to processors depending on usage, machine load and other factors. The number of threads can be assigned by the runtime environment based on environment variables or in code using functions. The Open MP functions are included in a header file labelled "omp.h" in C/C++.



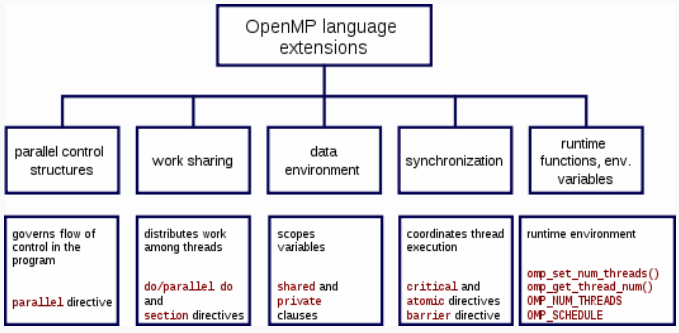
History

The Open MP Architecture Review Board (ARB) published its first API specifications, Open MP for Fortran 1.0, in October 1997. October the following year they released the C/C++ standard. 2000 saw version 2.0 of the Fortran specifications with version 2.0 of the C/C++ specifications being released in 2002. Version 2.5 is a combined C/C++/Fortran specification that was released in 2005. Version 3.0, released in May, 2008. Included in the new features in 3.0 is the concept of tasks and the task construct. These new features are summarized in Appendix F of the Open MP 3.0 specifications . Version 3.1 of the Open MP specification was released July 9, 2011. Mathematical Morphology was born in 1964 from the collaborative work of Georges Matheron and Jean Serra, at the École des Mines de Paris, France. Matheron supervised the PhD thesis of Serra, devoted to the quantification of mineral characteristics from thin cross

sections, and this work resulted in a novel practical approach, as well as theoretical advancements in integral geometry and topology. In 1968, the Centre de Morphologie Mathématique was founded by the École des Mines de Paris in Fontainebleau, France, led by Matheron and Serra. During the rest of the 1960s and most of the 1970s, MM dealt essentially with binary images, treated as sets, and generated a large number of binary operators and techniques: Hit-or misstransform,dilation, erosion, opening, closing, granulometry, thinning, skeletonization, ultimate erosion, conditional bisector, and others. A random approach was also developed, based on novel image models. Most of the work in that period was developed in Fontainebleau. In the 1980s and 1990s, MM gained a wider recognition, as research centers in several countries began to adopt and investigate the method. MM started to be applied to a large number of imaging problems and applications. In 1986, Jean Serra further generalized MM, this time to a theoretical framework based on complete lattices. This generalization brought flexibility to the theory, enabling its application to a much larger number of structures, including color images, video, graphs, meshes, etc. At the same time, Matheron and Serra also formulated a theory for morphological filtering, based on the new lattice framework. The 1990s and 2000s also saw further theoretical advancements, including the concepts of connections and levelings. In 1993, the first International Symposium on Mathematical Morphology (ISMM) took place in Barcelona, Spain. Since then, ISMMs are organized every 2–3 years, each time in a different world: Fontainebleau, France (1994); Atlanta, USA (1996); Amsterdam, Netherlands (1998); PalAlto, CA, USA (2000); Sydney, Australia (2002); Paris, France (2004); Riode Janeiro, Brazil (2007);Groningen, Netherlands (2009); and Intra (Verbania), Italy (2011).

The core elements

Chart of Open MP constructs



The core elements of Open MP are the constructs for thread creation, workload distribution (work sharing), data-environment management, thread synchronization, user-level runtime routines and environment variables. In C/C++, Open MP uses #pragmas. The Open MP specific pragmas are listed below:

Thread creation

omp parallel. It is used to fork additional threads to carry out the work enclosed in the construct in parallel. The original process will be denoted as master thread with thread ID 0.

Example (C program): Display "Hello, world" using multiple threads.

```
#include <stdio.h>
#include <stdlib.h>
 int main(int argc, char *argv[])
{
 #pragma omp parallel
 {
   printf("Hello, world.\n");
 }
 return 0;
}
```

Output on a computer with 2 Cores and 2 threads.

```
Hello, world.
Hello, world.
```

Work-sharing constructs

used to specify how to assign independent work to one or all of the threads.

- omp for or omp do: used to split up loop iterations among the threads, also called loop constructs.
- sections: assigning consecutive but independent code blocks to different threads
- single: specifying a code block that is executed by only one thread, a barrier is implied in the end
- master: similar to single, but the code block will be executed by the master thread only and no barrier implied in the end.

Example: initialize the value of a large array in parallel, using each thread to do a portion of the work

```
int main(int argc, char *argv[]) {
   const int N = 100000;
   int i, a[N];
   #pragma omp parallel for
   for (i = 0; i < N; i++)
```

```
    a[i] = 2 * i;

  return 0;

}
```

Open MP clauses

Since Open MP is a shared memory programming model, most variables in Open MP code are visible to all threads by default. But sometimes private variables are necessary to avoid race conditions and there is a need to pass values between the sequential part and the parallel region (the code block executed in parallel), so data environment management is introduced as data sharing attribute clauses by appending them to the Open MP directive. The different types of clauses are

Data sharing attribute clauses

- Shared: the data within a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the work sharing region are shared except the loop iteration counter.
- Private: the data within a parallel region is private to each thread, which means each thread will have a local copy and use it as a temporary variable. A private variable is not initialized and the value is not maintained for use outside the parallel region. By default, the loop iteration counters in the Open MP loop constructs are private.
- Default: allows the programmer to state that the default data scoping within a parallel region will be either shared, or none for C/C++, or shared, first private, private, or none for Fortran. The none option forces the programmer to declare each variable in the parallel region using the data sharing attribute clauses.
- First private: like private except initialized to original value.
- Last private: like private except original value is updated after construct.

Synchronization clauses

- Critical: the enclosed code block will be executed by only one thread at a time, and not simultaneously executed by multiple threads. It is often used to protect shared data from race conditions.
- Atomic: the memory update (write, or read-modify-write) in the next instruction will be performed atomically. It does not make the entire statement atomic; only the memory update is atomic. A compiler might use special hardware instructions for better performance than when using critical.

- ordered: the structured block is executed in the order in which iterations would be executed in a sequential loop
- Barrier: each thread waits until all of the other threads of a team have reached this point. A work-sharing construct has an implicit barrier synchronization at the end.
- No wait: specifies that threads completing assigned work can proceed without waiting for all threads in the team to finish. In the absence of this clause, threads encounter a barrier synchronization at the end of the work sharing construct.

Which Loops Are Parallelizable?

Parallelizable

- Number of iterations known upon entry, and does not change
- Each iteration independent of all others
- No data dependence

Not Parallelizable

- Conditional loops (many while loops)
- Iterator loops (e.g., iterating over a std::list<…> in C++)
- Iterations dependent upon each other
- Data dependence

Example: Parallelizable?

```
/* Gaussian Elimination (no pivoting):
x = A\b */
for (int i = 0; i < N-1; i++) {
for (int j = i; j < N; j++) {
double ratio = A[j][i]/A[i][i];
for (int k = i; k < N; k++) {
A[j][k] -= (ratio*A[i][k]);
b[j] -= (ratio*b[i]);
}
```

Scheduling clauses

- Schedule (type, chunk): This is useful if the work sharing construct is a do-loop or for-loop. The iteration(s) in the work sharing construct are assigned to threads according to the scheduling method defined by this clause. The three types of scheduling are:

1. Static: Here, all the threads are allocated iterations before they execute the loop iterations. The iterations are divided among threads equally by default. However, specifying an integer for the parameter "chunk" will allocate "chunk" number of contiguous iterations to a particular thread.
2. Dynamic: Here, some of the iterations are allocated to a smaller number of threads. Once a particular thread finishes its allocated iteration, it returns to get another one from the iterations that are left. The parameter "chunk" defines the number of contiguous iterations that are allocated to a thread at a time.
3. Guided: A large chunk of contiguous iterations are allocated to each thread dynamically (as above). The chunk size decreases exponentially with each successive allocation to a minimum size specified in the parameter "chunk"

IF control

- If: This will cause the threads to parallelize the task only if a condition is met. Otherwise the code block executes serially.

Initialization

- First private: the data is private to each thread, but initialized using the value of the variable using the same name from the master thread.
- Last private: the data is private to each thread. The value of this private data will be copied to a global variable using the same name outside the parallel region if current iteration is the last iteration in the parallelized loop. A variable can be both first private and last private.
- Thread private: The data is a global data, but it is private in each parallel region during the runtime. The difference between thread private and private is the global scope associated with thread private and the preserved value across parallel regions.

Data copying

- Copy in: copy in to pass the value from the corresponding global variables. No copy out is needed because the value of a thread private variable is maintained throughout the execution of the whole program.
- Copy private: used with single to support the copying of data values from private objects on one thread (the single thread) to the corresponding objects on other threads in the team.

Reduction

- reduction(operator | intrinsic : list): the variable has a local copy in each thread, but the values of the local copies will be summarized (reduced) into a global shared variable. This is very useful if a particular operation (specified in "operator" for this particular clause) on a datatype that runs iteratively so that its value at a particular iteration depends on its value at a previous iteration. Basically, the steps that lead up to the operational increment are parallelized, but the threads gather up and wait before updating the datatype, then increments the datatype in order so as to avoid racing condition. This would be required in parallelizing Numerical Integration of functions and Differential Equations, as a common example.

Others

- flush: The value of this variable is restored from the register to the memory for using this value outside of a parallel part
- master: Executed only by the master thread (the thread which forked off all the others during the execution of the OpenMP directive). No implicit barrier; other team members (threads) not required to reach.

User-level runtime routines

Used to modify/check the number of threads, detect if the execution context is in a parallel region, how many processors in current system, set/unset locks, timing functions, etc.

Environment variables

A method to alter the execution features of OpenMP applications. Used to control loop iterations scheduling, default number of threads, etc. For example OMP_NUM_THREADS is used to specify number of threads for an application.

Sample programs:

In this section, some sample programs are provided to illustrate the concepts explained above.

Hello World

This is a basic program that exercise the parallel private and barrier directives, as well as the functions omp_get_thread_num and omp_get_num_threads(not to be confused)

C program can be compiled using gcc-4.4 with the flag -fopenmp

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
 int main (int argc, char *argv[]) {
  int th_id, nthreads;
  #pragma omp parallel private(th_id)
  {
    th_id = omp_get_thread_num();
    printf("Hello World from thread %d\n", th_id);
    #pragma omp barrier
    if ( th_id == 0 ) {
      nthreads = omp_get_num_threads();
      printf("There are %d threads\n",nthreads);
    }
  }
  return EXIT_SUCCESS;
}
```

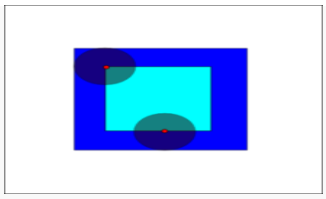
Binary morphology:

Basic operators

The basic operations are shift-invariant (translation invariant) operators strongly related to Minkowski addition.

Let *E* be a Euclidean space or an integer grid, and *A* a binary image in *E*.

Erosion



The erosion of the dark-blue square by a disk, resulting in the light-blue square.

The erosion of the binary image *A* by the structuring element *B* is defined by:

$$A \ominus B = \{z \in E | B_z \subseteq A\},$$

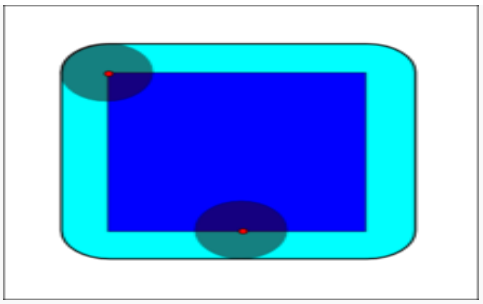
where B_z is the translation of B by the vector z, i.e., $B_z = \{b + z | b \in B\}, \forall z \in E$.

When the structuring element *B* has a center (e.g., *B* is a disk or a square), and this center is located on the origin of *E*, then the erosion of *A* by *B* can be understood as the *locus of points reached by the center of B when B moves inside A. For example, t*he erosion of a square of side 10, centered at the origin, by a disc of radius 2, also centered at the origin, is a square of side 6 centered at the origin.

The erosion of *A* by *B* is also given by the expression: $A \ominus B = \bigcap_{b \in B} A_{-b}$.

Example application: Assume we have received a fax of a dark photocopy. Everything looks like it was written with a pen that is bleeding. Erosion process will allow thicker lines to get skinny and detect the hole inside the letter "o".

Dilation



The dilation of the dark-blue square by a disk, resulting in the light-blue square with rounded corners.

The dilation of *A* by the structuring element *B* is defined by:

$$A \oplus B = \bigcup_{b \in B} A_b .$$

The dilation is commutative, also given by: $A \oplus B = B \oplus A = \bigcup_{a \in A} B_a$.

If *B* has a center on the origin, as before, then the dilation of *A* by *B* can be understood as the locus of the points covered by *B* when the center of *B* moves inside *A*. In the above example, the dilation of the square of side 10 by the disk of radius 2 is a square of side 14, with rounded corners, centered at the origin. The radius of the rounded corners is 2.

The dilation can also be obtained by: $A \oplus B = \{z \in E | (B^s)_z \cap A \neq \varnothing\}$, where B^s denotes the symmetric of B, that is, $B^s = \{x \in E | -x \in B\}$.

Example application: Dilation is the opposite of the erosion. Figures that are very lightly drawn get thick when "dilated". Easiest way to describe it is to imagine the same fax/text is written with a thicker pen.

Example for Morphological Image Processing:

A shape (in blue) and its morphological dilation (in green) and erosion (in yellow) by a diamond-shape structuring element.

Parallelized code for binary EROSION:

```
Int x,y;
#paragma 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(FukkFit(inputImage,x,y,structuringElement))
               outputImage[index]=1;
           else
               outputIkage[index]=0;
         }
      }
}
```

Conclusion

This article has only scratched the surface of the capabilities of Open MP for parallelized signal and image processing on morphology. More advanced features, such as synchronization and parallel regions, extend the basic functionalities described here. However, with simple use of the Open MP *parallel for* directive, it is remarkably easy for the signal

processing programmer to achieve loop level parallelism, Open MP will continue provide an uncomplicated way to harness the increasing the power of morphology.

References

1. Reference/tutorial page on nersc.gov
2. http://www.efg2.com/Lab/Library/ImageProcessing/Algorithms.htm
3. http://computervisiononline.com/tiki-index.php?page=Image+Processing
4. *The Image Processing and Measurement Cookbook* by Dr. John C. Russ www.reindeergraphics.com/tutorial/index.shtml
5. The Morphology Digest is intended as a forum between workers in the field of Mathematical Morphology and related fields (stochastic geometry, random set theory, image algebra, etc.).
6. www.cwi.nl/ftp/morphology/digest
7. Mathematical Morphology and Image Interpolation. http://cmm.ensmp.fr/~beucher/interpol/interpol.html