Use OpenMP to Speedup Math Computation

Yong-Ming Li

May, 2010

1 Introduction

For the past thirty years, our math algorithms and geometric creation and manipulation routines have all been implemented for serial programming. In other words, each program consists of a sequence of instructions, where each instruction executes one after the other. It runs from start to finish on a single processor. Although we have been continuously improving the performance of key algorithms, the overall speed of math computation still heavily depends on the advance of CPU clock rate. However, chip makers have struggled to design CPUs that run much faster than about 3.5 GHz due to thermodynamic limits in current semiconductor process technologies (e.g., wasted energy in the form of heat). The highest clock speed microprocessor ever sold commercially to date is found inside IBM's zEnterprise 196 mainframe, introduced in July, 2010. Its cores run continuously at 5.2 GHz. Due to the physical limitations, the major CPU vendors have shifted their attention away from ramping up clock speeds to adding parallelism support with multi-core processors that can handle different tasks simultaneously. This seems like pretty good news. But math code may not run any faster if we do not do anything about it. This is where OpenMP (Open Multi-Processing) comes into the picture. OpenMP helps C/C++ developers create multi-threaded applications with minimum and manageable changes in their serial code. Unlike MPI (Message Passing Interface) that is most suited for a system with multiple processors and multiple memory (e.g., a cluster of computers with their own local memory), OpenMP is designed for shared memory systems like we have on our desktop and laptop computers. In computer hardware, shared memory refers to a large block of random access memory that can be accessed by several different central processing units (CPUs) in a multiple-processor computer system.

OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization. It uses the so called *fork-join* model of parallel execution as shown below:

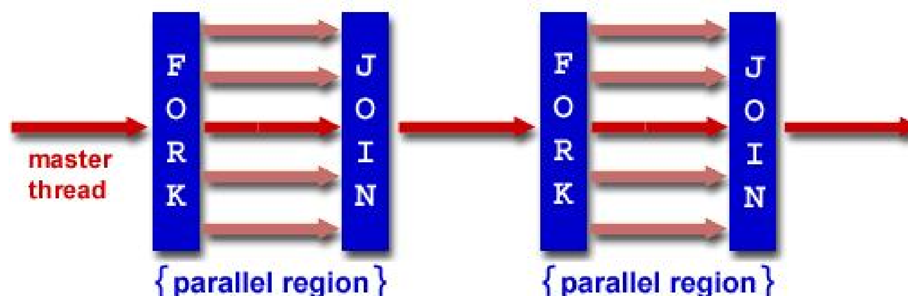

Figure 1: OpenMP: The fork-join model

All OpenMP programs begin as a single process known as the master thread. It executes sequentially until the first *parallel region* construct is encountered. The master thread then creates a team of parallel threads. The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads. When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.

2 Enable OpenMP in Visual C++

OpenMP is available since Visual Studio 2005. To use OpenMP, you need to do the following two things:

- Include <omp.h>
- Enable OpenMP compiler switch in Project Properties as illustrated in Figure 1.

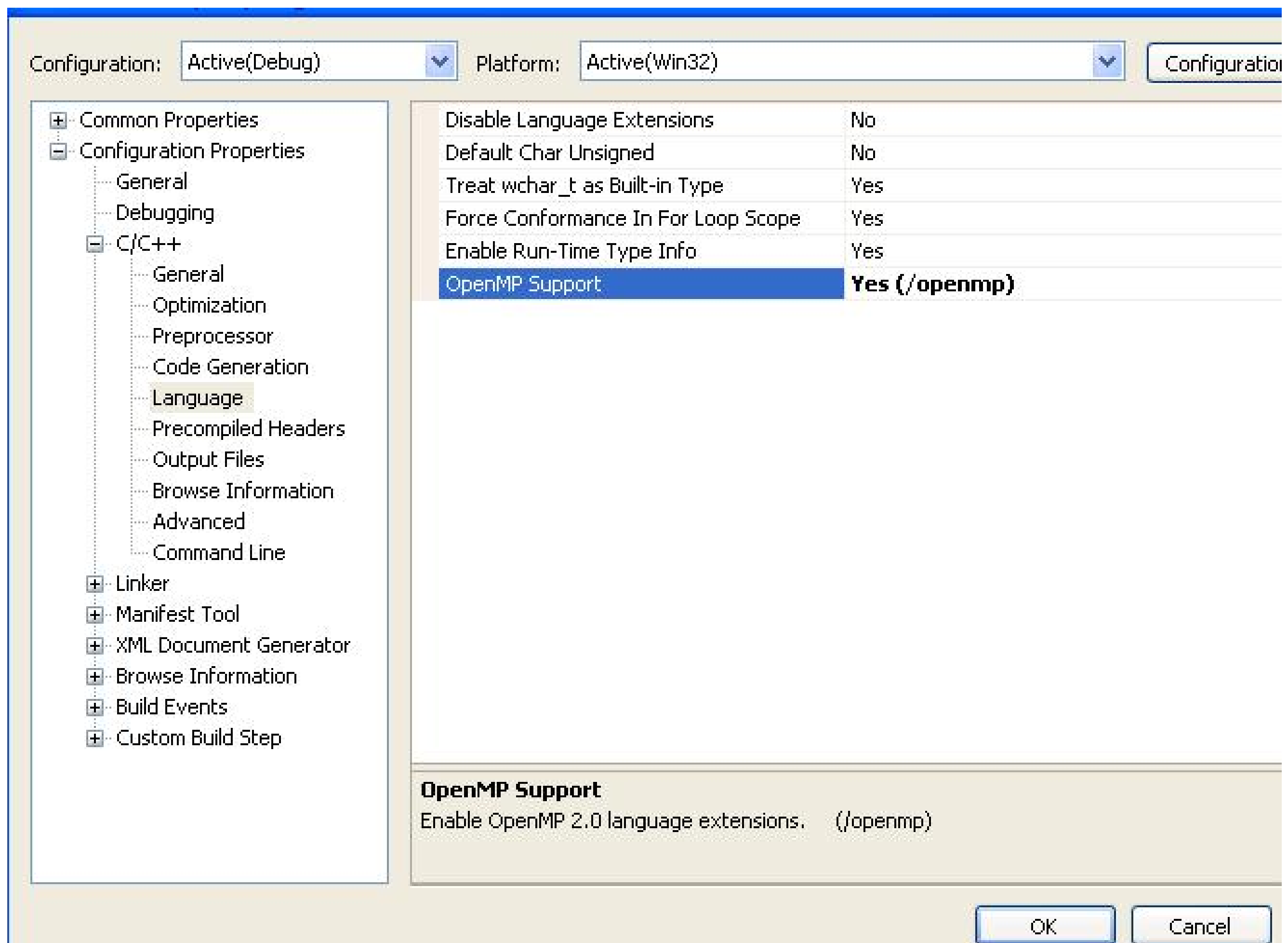

Figure 2: Enabling OpenMP for Visual C++

3 Specify Parallelism

OpenMP is easy to use and consists of only two basic constructs: `pragmas` and `runtime` routines. The OpenMP `pragmas` typically direct the compiler to parallelize sections of code. All OpenMP

`pragmas` begin with `#pragma omp`. As with any `pragma`, these directives are ignored by compilers that do not support the feature.

OpenMP `runtime` routines are used primarily to set and retrieve information about the environment. They will be discussed in later section. There are also APIs for certain types of synchronization. In order to use the functions from the OpenMP `runtime`, the program must include the OpenMP header file `omp.h`.

You add parallelism to an application with OpenMP by simply adding `pragmas` and, if needed, using a set of OpenMP function calls from the OpenMP `runtime`. These `pragmas` use the following form:

```
#pragma omp <directive> [clause[ [,] clause]...]
```

The directives specify either work-sharing or synchronization constructs. It is not the purpose of this article to survey all the directives and clauses. We discuss only what we use frequently such as `parallel` and `for`. We start by considering evaluation of 100 points on a circle centered at the origin:

```
for (i=0; i<100; i++)
{
   x[i] = radius * cos(alpha[i]);
   y[i] = radius * sin(alpha[i]);
}
```

To improve the performance of the above code on a multi-core processor computer, we can use two OpenMP directives as follows:

```
#pragma omp parallel
{
   #pragma omp for
   for (i=0; i<100; i++)
   {
      x[i] = radius * cos(alpha[i]);
      y[i] = radius * sin(alpha[i]);
   }
}
```

The first directive tells the compiler that the structured block of code should be executed in parallel on multiple threads. The second directive (`#pragma omp for`) is a work-sharing directive that tells OpenMP that, when called from a parallel region, the `for` loop should have its iterations divided among the thread team. If the above code is run on a computer with four processors, the iterations of the loop may get allocated such that processor 1 gets iterations 1-25, processor 2 gets iterations 26-50, processor 3 gets iterations 51-75, and processor 4 gets iterations 76-100. One thing that is not explicit in this program is a barrier synchronization at the end of the parallel region. All threads will block there until the last thread completes. It should also be pointed out that, without the second directive, each thread would execute the complete `for` loop. As a result, each thread would do redundant work.

It seems very easy to add parallel programming capability into serial code. If this is the case, there is no need to write this article. As it will become clear in the subsequent sections, some special considerations need to be taken when specifying parallelism in a program designed and implemented for serial computation.

4 OpenMP `runtime` routines

OpenMP provides a number of `runtime` routines that can be used to obtain information about threads in the program. Some commonly used routines are:

- `omp_get_num_procs()` - Returns the number of processors that are available to the program
- `omp_get_num_threads()` - Returns the number of threads in the current team. In a sequential section of the program, this method returns 1.
- `omp_get_thread_num()` - Returns the current thread ID.
- `omp_set_num_threads(int)` - Specifies the number of threads used in subsequent parallel sections.

In addition, OpenMP provides a number of lock routines that can be used for thread synchronization.

Running the following example with Intel® Core™i7 processor, we will get `numProcs=8` and `numThreads=1`. The reason we got `numThreads=1` is because we are not in parallel region yet. Therefore, there is only one master thread.

```
int numProcs, numThreads;
numProcs = omp_get_num_procs();
numThreads = omp_get_num_threads();

#pragma omp parallel
{
   #pragma omp for
   for (i=0; i<100; i++)
   {
      x[i] = radius * cos(alpha[i]);
      y[i] = radius * sin(alpha[i]);
   }
}
```

If we move the calls to the `runtime` routines inside the parallel region, we should be able to get, by default, `numProcs=8` and `numThreads=8`. It is possible to explicitly set the desired number of threads for the parallel region as illustrated below:

```
omp_set_num_threads(4);
#pragma omp parallel
{
   #pragma omp for
   for (i=0; i<100; i++)
   {
      x[i] = radius * cos(alpha[i]);
      y[i] = radius * sin(alpha[i]);
   }
}
```

Inside the parallel region, four threads will be used to share the `for` loop. Outside the parallel region, there is still a single master thread. It should be pointed out that a call to `runtime` routines may be combined with `#pragma` directive as follows:

```
#pragma omp parallel num_threads(4)
{
   #pragma omp for
   for (i=0; i<100; i++)
   {
      x[i] = radius * cos(alpha[i]);
      y[i] = radius * sin(alpha[i]);
   }
}
```

5 Tolerance handling

Many geometric operations involve the use of tolerances such distance tolerance, data fitting tolerance, angular tolerance, etc. For example, we consider two curves are connected if the distance between the end point of the first curve and start point of the second curve is less than the distance tolerance. Each CAD system has its own default tolerances. This may introduce an incompatibility issue when a CAD model from a different system is imported into ours. This incompatible issue could be resolved if our math routines had all been implemented to take tolerances as input. Since this is not the case, we have to provide a set of methods to get and reset system default tolerances. For example, we can set system tolerances either smaller or larger when we validate an imported model. When this validation and healing process is done, we can set the system tolerances back to the original values. Such mechanism works well for serial programming but will most likely not work for parallel programming. This is because tolerances values are saved as global variables. When they get changed by one thread, the new values are immediately exposed to other threads. This may introduce unexpected results or complications. There is no easy solution to this kind of problems rather than rewriting interface to take in tolerances. For now, my recommendation is that we do not introduce parallelism into those routines where system default tolerances will be modified.

6 Data scope

In writing parallel programs, understanding which data is shared and which is private becomes very important, not only to performance, but also for correct operation. Shared variables are shared by all the threads in the thread team. Thus a change of the shared variable in one thread becomes visible to all the threads in the parallel region. Private variables, on the other hand, have private copies made for each thread in the thread team, so changes made in one thread are not visible to the other threads (*Caution about the use of structure/class where C pointers are member variables*: modification to the private variable in this case may be visible to other threads. We will discuss it in detail later). By default, all the variables in a parallel region are shared, with some exceptions such as for loop index and locally defined variables.

We now discuss how data scope may affect our math computation when adding parallel programming capability. Each math routine has an error flag to inform the caller whether this routine succeeded or failed so that he can take a corresponding action. The error code is usually initialized

to IMSUCC (i.e., no error) at the beginning of each function, which may be a problem for parallel programming. For an example, let's look at the computation of minimum distance between a point and a surface. Referring to Figure 3, the minimum distance is obtained by computing the Euclidean distance between the given point and its projection point.

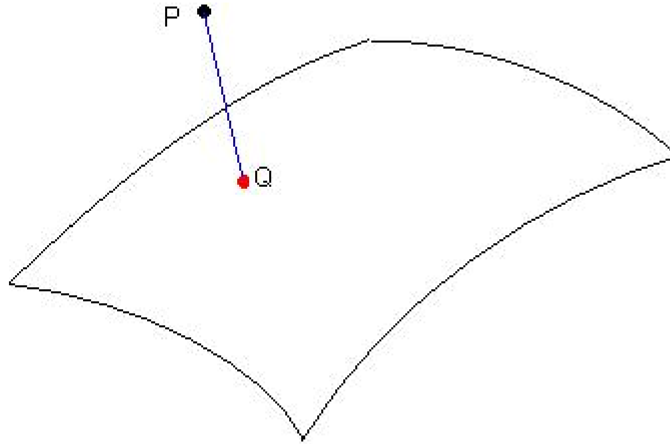

Figure 3: The minimum distance is given by Euclidean norm $||\mathbf{P} - \mathbf{Q}||$.

In our math library, we have a routine to perform such task. In order to compute the minimum distances between a set of points and a surface, we can loop through each point as illustrated in the following code fragment:

```
for (i=0; i<num_pts; i++)
{
   IMMinDistPointSurface(
             surf,          // Input: surface
             pPoints[i],    // Input: position point
             pProjPts[i],   // Output: projection point of pPoints[i]
             pUVParams[i],  // Output: (u, v) parameters w.r.t. pProjPts[i]
             pDists[i],     // Output: minimum distance
             rc);           // Output: error code
   if (rc != IMSUCC)
   {
      goto wrapup;
   }
}
```

The above computation can be expensive when there are many points to be processed and the surface is a complex B-spline surface. Let's introduce a parallelism to this code fragment:

```
#pragma omp parallel
{
   #pragma omp for
   for (i=0; i<num_pts; i++)
   {
      IMMinDistPointSurface(surf, pPoints[i], pProjPts[i],
                            pUVParams[i], pDists[i], rc);
      if (rc != IMSUCC)
      {
```

```
            goto wrapup;
        }
    }
}
```

If we compile it, we would receive an warning:

```
error C3010:  'wrapup' :  jump out of OpenMP structured block not allowed.
```

Even if there was no compiling error, we would still have a problem in getting an error report. By default, `rc` is a shared variable to all the threads. If the computation encounters an error in one thread processing, the error code will be assigned with a specific value and the process is directed to the error handling code (e.g., properly free any dynamically allocated memory). In the meantime, another thread is computing independently and may initialize the shared variable `rc` to `IMSUCC`. Therefore, the calling function will not be able to know there was a failure. To fix the data scope issue, we can define a local error flag and record any failure in a global flag. We also need to take out `goto wrapup` statement. The correct code is listed as follows:

```
#pragma omp parallel
{
    int  rc1;
    #pragma omp for
    for (i=0; i<num_pts; i++)
    {
        IMMinDistPointSurface(surf, pPoints[i], pProjPts[i],
                              pUVParams[i], pDists[i], rc1);
        if (rc1 != IMSUCC && rc == IMSUCC)
        {
            rc = rc1;
        }
    }
}
if (rc != IMSUCC)
    goto wrapup;
```

Problems associated with shared data may not be as obvious as we just discussed. It requires us to understand how each argument is used in the sub-function. We take the same code fragment to explain it. The above code fragment saves each projection point, (u, v) parameters, and minimum distance in pre-allocated memory `pProjPts`, `pUVParams`, and `pDists` respectively. Sometime, callers want only the minimum distances and hence do not need to allocate memory for `pProjPts` and `pUVParams` but repeatedly use static (structure) variables `pjpt` and `uv`. as shown below:

```
#pragma omp parallel
{
    int  rc1;
    #pragma omp for
    for (i=0; i<num_pts; i++)
    {
        IMMinDistPointSurface(surf, pPoints[i], pjpt,  uv,  pDists[i], rc1);
        if (rc1 != IMSUCC && rc == IMSUCC)
```

```
        {
            rc = rc1;
        }
    }
}
```

The above code works fine if it is used for serial programming but will generate unpredictable result for parallel programming. This is because `pjpt` is declared somewhere outside the parallel region and hence a shared variable. Since it will be used by the sub-function to determine the minimum distance, the result becomes unpredictable as it is getting modified by other threads. To correct this issue, we may either define it locally as we did for `rc1` or use optional clause to declare it as a private as shown below:

```
#pragma omp parallel
{
    int rc1;
    #pragma omp for private(pjpt)
    for (i=0; i<num_pts; i++)
    {
        IMMinDistPointSurface(surf, pPoints[i], pjpt,  uv,  pDists[i], rc);
        if (rc1 != IMSUCC && rc == IMSUCC)
        {
            rc = rc1;
        }
    }
}
```

It should be pointed out that the `private` clause does not initialize the variable when a new copy is created for each thread. Since `pjpt` is an output, it is fine without initializing it. If we do want to initialize a variable to the value before entering the parallel region, we need to use `firstprivate` clause. There is also `lastprivate clause` that will initialize a variable to the value of last iteration before leaving the parallel region.

I would like to give one more example on how data scope may affect the outcome results unexpectedly when a serial code is constructed parallel. Math library has many geometric structures such as `GCone`, `GBspSurface`, `GTypedSurface`, etc. These structures are often the input to sub-functions. Although it is advised not to modify the input, we do occasionally modify the input for certain purpose and change them back before exiting the sub-function. For example, we may set the number of boundaries of input surface to zero when we want to process it as an infinite surface and then change it back to its original values after the computation is done. This is not a problem for a serial program. However, it creates a serious problem for parallel programming and it is usually very difficult to find the cause. Our geometric structures usually have C pointers associated with allocated memory that stores the geometry. By declaring such a structure as `private` will not help even though a private copy of structure will be created for each thread. This is because the new copy is passed into a sub-function exactly in the same way as passing a structure by value to a function in C. Assume that we pass the following `GTypedSurface` structure into a sub-function by value:

```
struct tagGTypedSurface
{
   GSurface*  m_pSurface;
   UCHAR      m_type;
};
```

Changing the value of `m_type` in sub-function will not affect the original `GTypedSurface` since a copy of structure was sent to the sub-function. However, any modification to `m_pSurface` actually alters the content of the original `m_pSurface` as they both share the same memory address.

Our examples in this section all have simple `for` loops. Accordingly, it is not very difficult to identify what variables need to be declared as private. In real world programming, however, the `for` loop block may contains quite a few operations. In this case, it becomes increasingly difficult to properly manage the scope of all variables inside the `for` loop. My recommendation is to make sub-function out of the block so that the `for` loop can be simplified.

7 Use of `std::vector` in parallel region

The Standard Template Library (STL) containers (especially, the `std::vector`) are widely used in our math library to store data because of their ability to resize themselves automatically when inserting or erasing an object. In the following code fragment, we can see how STL vectors are used to save projection points, parameters and minimum distances.

```
double                 d, uv[2];
GPosition              pjpt;
std::vector<GPosition> pjpt_list;
std:vector<double>     upar_list;
std::vector<double>    vpar_list;
std:vector<double>     dmin_list;

for (i=0; i<num_pts; i++)
{
   IMMinDistPtTypedSurface(surf, pPoints[i], pjpt, uv, d, rc);
   if (rc != IMSUCC)
      goto wrapup;
   pjpt_list.push_back(pjpt);
   dmin_list.push_back(d);
   upar_list.push_back(uv[0]);
   vpar_list.push_back(uv[1]);
}
```

For a serial program, results are saved sequentially. Therefore, `pjpt_list[i]` corresponds to the projection point of `pPoints[i]`. If we make the `for` loop block to be a parallel region, each thread would race to save data in STL vectors. Accordingly, the sequence of the data becomes unpredictable.

If we know in advance how many data will be saved, then we can perform parallel programming as follows:

```
std::vector<GPosition> pjpt_list;
std:vector<double>     upar_list;
std::vector<double>    vpar_list;
std:vector<double>     dmin_list;

pjpt_list.resize(num_pts);
dmin_list.resize(num_pts);
upar_list.resize(num_pts);
vpar_list.resize(num_pts);
#pragma omp parallel
{
   int         rc1;
   double      d, uv[2];
   GPosition   pjpt;

   #pragma omp for
   for (i=0; i<num_pts; i++)
   {
      IMMinDistPtTypedSurface(surf, pPoints[i], pjpt, uv, d, rc1);
      if (rc != IMSUCC && rc1 != IMSUCC)
         rc = rc1;

      pjpt_list[i] = pjpt;
      dmin_list[i] = d;
      upar_list[i] = uv[0];
      vpar_list[i] = uv[1];
   }
}
```

It is very common that a STL vector is used in math routines because the size of array is unknown in advance. In this case, we can use the `omp ordered` directive that identifies a structured block of code that must be executed in sequential order.

```
#pragma omp parallel
{
   int         rc1;
   double      d, uv[2];
   GPosition   pjpt;

   #pragma omp for schedule(dynamic) ordered
   for (i=0; i<num_pts; i++)
   {
      IMMinDistPtTypedSurface(tsf, pPoints[i], pjpt, uv, &d, &rc1);
      if (rc != IMSUCC && rc1 != IMSUCC)
         rc = rc1;
      #pragma omp ordered
      {
         pjpt_list.push_back(pjpt);
         dmin_list.push_back(d);
         uPar_list.push_back(uv[0]);
```

```
            vPar_list.push_back(uv[1]);
        }
    }
}
```

There is a little performance drawback when data is forced to save sequentially. However, it is ignitable when it is constructed properly. The `omp ordered` directive must be used as follows:

- It must appear within the extent of `omp for` or `omp parallel for` construct containing an ordered clause.
- It applies to the statement block immediately following it. Statements in that block are executed in the same order in which iterations are executed in a sequential loop.
- An iteration of a loop must not execute the same `omp ordered` directive more than once.
- An iteration of a loop must not execute more than one distinct `omp ordered` directive.

With `#pragma omp for` directive, the iterations are divided among threads equally by default. By adding `schedule(dynamic)` clause to the above routine, 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. We found it runs faster by adding `schedule(dynamic)` clause.

8 Conflict with internal memory checking tool

Long before commercial memory tools such `BoundsChecker` were available, we had developed a memory tool to catch memory leak, memory scratch (i.e., accessing memory beyond defined size), and use of uninitialized memory. This tool is available internally to math developers and has played a vital role in developing clean and reliable math routines. As a matter of fact, I have not seen a single incident that this tool did not catch a memory leak but caught by other commercial memory tools. Our memory tool can be summarized as follows:

- When users call memory allocation, we allocate two more cells so that we can put a special flag at the beginning and end of allocated memory block. The memory address is also saved in a global list.
- When users call memory free, we check if the allocated memory still has the added two flags to make sure users did not scratch the memory. We then delete this memory and remove the address from our list.
- When a session ends, we check if all allocated memory addresses have been removed from the list. If not, we have memory leaks.

Since the list that tracks memory allocation and free is a global data and hence not thread-safe because each thread races to update the counter and the content of the list. One of the remedies to this problem is to block the critical region of code to ensure this blocked region can be executed one thread at a time. This can be done by using the directive clause to mark the critical region as follows:

```
#pragma omp critical(dataupdate)
{
  ...
}
```

By doing so, however, it will degrade the performance and hence diminish the advantage of parallel programing. For this reason, we disable the memory tool for end-user release.

I tried `BoundsChecker` on an OpenMP application without success (It went deadlocked when multi-thread was enabled). Therefore, it is unclear whether `BoundsChecker` supports OpenMP. OpenMP is supported by Intel Parallel Inspector (available as part of Intel Parallel Studio or as a standalone product). It adds memory and thread checking into Microsoft Visual Studio. Its memory checking includes memory leaks, dangling pointers, uninitialized variables, use of invalid memory references, mismatched memory, allocation and deallocation, stack memory checks, and stack trace with controllable stack trace depth. Thread checking includes race conditions, deadlocks, depth configurable call stack analysis.

9 Debug parallel programming

Debugging OpenMP applications with Visual Studio 2005 and later is complicated. In particular, stepping into and/or out of a parallel region with `F10/F11` can be very confusing. This is because the compiler creates some additional code to call into the runtime and to invoke the thread teams. The debugger has no special knowledge about this, so the programmer sees what looks like odd behavior. For example, by pressing the `F11` key we would expect to step into a sub-function. Instead, we are led to a different place because another thread has kicked in.

My recommendation for debugging an OpenMP application is to turn off multi-thread programming by either enforcing the parallel region to run on a single thread or disabling OpenMP (.e.g., changing the setting of project). If a problem can be reproduced, then it is likely the problem is not related to parallelism. Otherwise, we have to debug it in a hard way. Assume that we receive a return error when running multi-thread. In this case, we set a breakpoint at `if (rc!=IMSUCC)` and run the program in debug. When it stops there, we cannot simply step back to a certain place and execute the program from there as we usually do for debugging a single thread application. In this case, we need to enable the `Threads` window by clicking `Debug->Windows->Threads` as shown below:

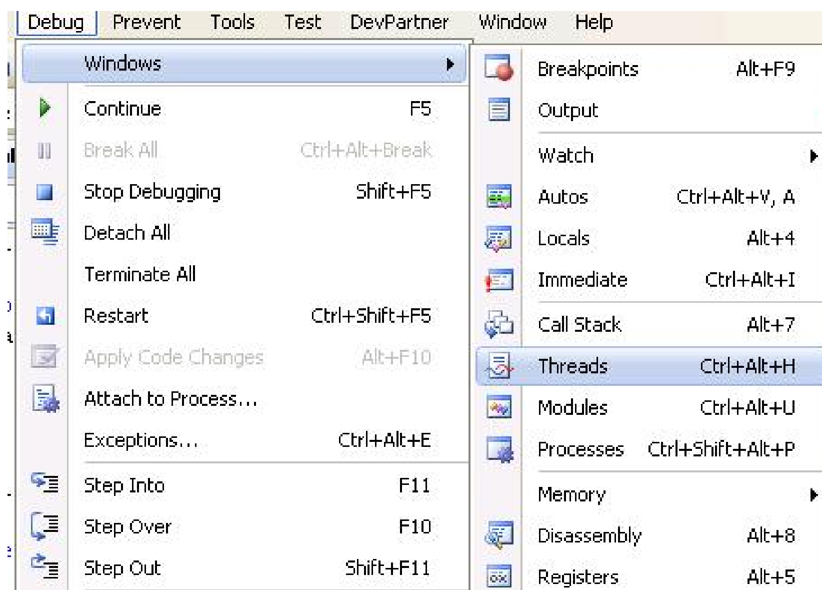

Figure 4: Enabling Threads Window.

The `Threads` windows will show the various threads that are running in the thread team. The thread IDs will not correlate with the OpenMP thread IDs; however, they will reflect the Windows thread ID that OpenMP is built upon. We then highlight all thread IDs except for the one currently running and click the right mouse button to select `freeze`. This will freeze all highlighted threads. Now, we can step back and execute the program from there to investigate why computation failed.

10 Performance analysis

Our performance tests are all based on my Sony laptop computer with the following system configuration:

```
Processor:              Intel Core i7 CPU   Q720 @ 1.60 GHz  1.60 GHz
Installed memory (RAM): 4.00 GB
System:                 64-bit Operating System (Windows 7 Home Premium)
```

It has four cores with eight threads.

Our first example is to compute minimum distances between 5,868 points and a B-spline surface whose shape is shown in Figure 4.

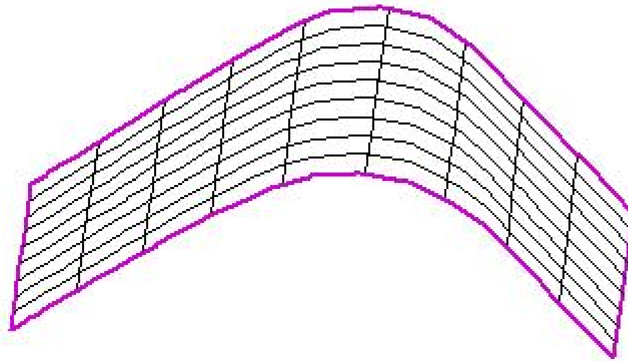

Figure 5: A cubic B-spline surface with 46×8 control vertices.

The CPU time consumed by running the following code in release with one, two, four, and eight threads is respectively 6.24, 4.52, 2.68, and 1.82 seconds. It should be pointed out that constructing parallel region with one thread is equivalent to a traditional serial program. As it was indicated, the performance improvement is significant when parallelism is introduced to a traditionally serial program.

```
#pragma omp parallel num_threads(numSpecifiedThreads)
{
   int  rc1;
   #pragma omp for
   for (i=0; i<num_pts; i++)
   {
      IMMinDistPointSurface(surf, pPoints[i], pProjPts[i],
                            pUVParams[i], pDists[i], rc1);
      if (rc1 != IMSUCC && rc == IMSUCC)
```

```
        {
            rc = rc1;
        }
    }
}
```

If this code is run on a less powerful computer such as Dell Optiplex GX620 that has only one core and two hyper-thread, we will still see roughly 25% performance improvement (9.46 seconds with one thread and 7.11 seconds with two threads).

11 Conclusions

Most math routines are not object oriented programming. They are basically functional programming because they are, after all, C implementation of *mathematical functions*. Therefore, data encapsulation emphasized in object oriented programming has not been the primary concern. For this reason, I have encountered a quite few cases where modifications to unprotected data in low level math routines had caused wrong results when parallelism was introduced. Therefore, it is relatively involved to construct parallelism in math functions. But with some efforts, it is possible and realistic to add parallel programming capability into our traditionally serial programs to greatly speedup the overall performance. There are some math functions such as n-surface merge, the least squares fitting of B-spline surface, ship hull 2D-expansion, range-box computation, and ACIS to Gtypes conversion are standalone sub systems. Performance improvement in these areas will strengthen our competitive position and make our customers happy.

At the write of this document, many things still remain unclear to me. For example, to my knowledge no any article has specifically mentioned whether **malloc** is thread-safe or not. I thought it was until I ran into a crash where one thread is processing the data in allocated memory while another thread allocates memory of the same address, which corrupted the data in use. More research and investigation are needed to find out whether **malloc** is thread-safe or not If not, it is of little interest to add parallel programming into legacy C code as **malloc** is so widely used in our legacy C code.