Objectives

1. To parallelize simple C programs using OpenMP
2. To understand performance issues and trade-offs in parallel programming

OpenMP consists of a set of compiler ‘#pragmas’ that control how the program works. A #pragma is a compiler directive and is going to tell the compiler to do something special beyond the scope of C language. Directive specifics are ‘#pragma parallel’, ‘#pragma for’ etc. The pragmas are designed so that even if the compiler does not support them, the program will still yield correct behavior, but without any parallelism. By default, the Visual Studio does not support the ‘#pragma omp’ clause. However, no error or warning will be shown. The compiler will just ignore the ‘#pragma’ and the program will run serially.

**How to use openmp in Visual Studio**: If /openmp isn't specified in a compilation, the compiler ignores OpenMP clauses and directives. OpenMP Function calls are processed by the compiler even if /openmp isn't specified. To allow OpenMP pragmas in Visual Studio, select ‘Project’, and then ‘Properties’. A pop-up window will open. Expand the ‘configuration properties’ and select ‘C/C++’ and then ‘Language’. You will see on the right an ‘OpenMP Support’ option, select ‘yes’. Do not forget to include <omp.h> library.

**How to use openmp in Linux**: Just compile using ‘-fopenmp’ option. Do not forget to include <omp.h> library.

See the code example below. Pay attention to the ‘**#pragma omp parallel’ clause.** Without this construct, there is no multiple threads. The code that is inside the **#pragma omp parallel { } region, runs by multiple threads. All the threads run the same code.** It is a fork join program. Each thread has an ID number. We can extract this number by using the follwing openmp pragma construct *omp\_get\_thread\_num().*

If we define data outside the **#pragma** omp parallel{ } region, they are allocated to heap memory (visual to any thread, shared data). If they are defined inside the omp parallel{ } region, they are allocated to the threads individual stack (private to the thread, local).

int main(){

*double A[1000];*

*omp\_set\_num\_threads(4); //requests 4 threads.*

*#pragma omp parallel { //fork a number of threads – I asked 4*

*int ID = omp\_get\_thread\_num(); //get the ID for each thread*

*function1 (ID, A ) //each thread will run this function*

*} //end of multi-threading region*

*printf(“all done\n”); //just the main thread runs this command*

*return 0;*

*}*

**Task1**: Download, run and study the ‘hello.c’ program. Every thread has its own ‘*nthreads, tid*’ variables, while ‘procs, maxt, inpar, dynamic, nested’ are shared amongst all threads. The code that is inside the ‘*#pragma omp parallel { }*’ region, runs by all threads. The thread with ID number zero, is always the main thread and therefore only the master thread will run this code ‘*if (tid == 0) { }*‘. The following commands get environment information (there are others too):

* nthreads = omp\_get\_num\_threads(); //returns the number of threads used inside #pragma omp parallel { }
* procs = omp\_get\_num\_procs(); //returns the number of physical CPU cores of this machine
* maxt = omp\_get\_max\_threads(); //returns the maximum number of threads available. by default this number will be set to the maximum number of available cores
* inpar = omp\_in\_parallel(); //This function returns true if currently running in parallel, false otherwise.
* dynamic = omp\_get\_dynamic(); //This function returns true if enabled, false otherwise. We will further explain this next week
* nested = omp\_get\_nested(); //This function returns true if nested parallel regions are enabled, false otherwise. If undefined, nested parallel regions are disabled by default. We will further explain this next week

**Task2**: Download ‘array\_addition\_serial.c’ program. Try to parallelize ‘un\_opt()’ function. Three different solutions are provided. Make sure you understand them all.

**Task3**: Download ‘reduction\_serial.c’ program. Try to parallelize un\_opt() function. Several different solutions are provided. Make sure you understand them all. Pay attention to the OpenMP Reduction clause. Each thread has its own copy of ‘ave’, each thread does its own summation and when they are done, they are combined with the global copy of ‘ave’. The reduction clause is necessary (a race condition occurs) because otherwise we never know which threads will be writing concurrently to ‘ave’. Without using the reduction clause, some threads might clashing and trying to update the memory at the same time. This case, you never know what the value of ‘ave’ will be.

**reduction (op : list) –** A local copy of each ‘list’ variable is made and initialized depending on the ‘op’, e.g., 0 for ‘+’. Updates occur on the local copy. Local copies are reduced into a single value and combined with the original global value.

**Task4**: Download ‘PI.c’ program. Try to parallelize ‘un\_opt()’ function. Several different solutions are provided. Make sure you understand them all.

**Task5**: Measure the execution time of all the PI versions provided. Use different number of threads for each version. To get an accurate measurement, the execution time must be at least a few seconds. Compare the results.

**Version1()**: Notice that the OpenMP version that uses just one thread is slower than the serial version. This is because OpenMP adds an overhead. By using more threads, the execution time is reduced, but the scalability is low. This is because of the cache false sharing problem in sum array. Just after the ‘*#pragma omp parallel { }’*, the local variables are lost. So, for the sum to be visible, we must promote it to an array. However, this introduces the false sharing problem as Thread0 uses sum[0], Thread1 uses sum[1] etc, and sum[0:7] share the same cache line.

Although threads do not use share data, they use the same cache line, which is a shared hardware resource too. **False sharing** is a well-known performance issue on symmetric multiprocessor (SMP) systems, where each CPU core has its own private data cache memory and all private caches are connected to a shared cache. It occurs when threads on different processors modify variables that reside on the same cache line. This is called false sharing because each thread is not actually sharing access to the same variable. Normally, when a CPU core modifies the value of an item in a *shared* cache line, all copies of that particular cache line become *invalid* on the other cores that hold it. So if that same cache line is needed later by another core, whatever may have been present in its local L1/L2 is no longer valid; the core might have to go all the way out to main memory to get a useable replacement. This is because the memory system must guarantee cache coherence.

**Version2**: This implementation addresses the false sharing problem by not using different cache lines to store the threads’ sum variables. So, the scalability of this implementation is good. This implementation has an inefficiency. The developer must consider the cache line size to develop this version and thus if the program run on another machine (with different cache line size) this value must be amended appropriately. The sum[] array is promoted to a 2-d array sum[NUM\_THREADS][PAD]. Keep in mind that only the 1st column is used; the others contain trash. PAD value is selected so as each sum[][] row occupies an entire cache line; the cache line size in Intel processors is 64bytes and thus can store 8 double values. Thus sum[0][0] will always be stored into another cache line than sum[1][0], sum[2][0] etc.

***omp critical***: **The critical construct restricts execution of the associated structured block to a single thread at a time**. When the omp critical pragma is used, threads wait at the beginning of the critical section until no other thread in the team is executing it.

***omp atomic***: **The atomic construct ensures that a specific storage location is accessed atomically**, rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values

**Version3**: This version achieves same performance as version. In this version there is no shared array and thus there is no false sharing. This version does not consider the cache line size and thus it is portable. In the ‘#pragma omp critical’ clause, all the threads use their private sum variable to update the shared pi variable. This is not performed in parallel; only one thread at a time can enter the critical block. We could also use 'atomic' instead of 'critical'.

**Version4**: It is a slow alternative of version3, as the critical section includes many loop calculations. Therefore, a significant part of the program is not parallelized.

**Version5/6**: These are the recommended versions, showcasing the power of OpenMP. See how elegant version6 is, we added just a single line of code and made this program parallel. In version6, x is not defined inside the parallel region. Thus, by default it is a shared variable. private(x) creates a private x variable in each thread. Be Careful: x is uninitialized no matter what its previous value is.

Further Reading

1. Guide into OpenMP: Easy multithreading programming for C++, available at <https://bisqwit.iki.fi/story/howto/openmp/#ParallelConstruct>
2. OpenMP Application Programming Interface Examples, available at <https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwiOip2R-rrqAhX8XRUIHa5HC0QQFjAAegQIAxAB&url=https%3A%2F%2Fwww.openmp.org%2Fwp-content%2Fuploads%2Fopenmp-examples-4.5.0.pdf&usg=AOvVaw3BDlLKC3VhdJI1iTj1RE_p>