OpenMP Programming

Objectives

1. To understand performance issues and trade-offs in parallel programming
2. To parallelize C/C++ programs using OpenMP
3. To vectorize C/C++ programs using OpenMP
4. To evaluate different scheduling parameters and implementations and analyze performance trade offs
5.

Aim

The aim of this session is to acquaint yourselves with OpenMP framework. To write multithreaded and vectorized programs using OpenMP.

Introduction

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 the 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.

OpenMP Basics

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 following 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 – we 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 the '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

Array addition Example

Task2: Download '*array_addition_serial.c*' program. Try to parallelize '*un_opt()*' function. Three different solutions are provided. Make sure you understand them all.

The first implementation does not use the 'omp for' construct; this is **not** how we normally write OpenMP programs, but it is very important to understand how it works. The code in the parallel region is executed by all the threads; each thread has its own 'start' and 'end' values though, and thus each thread executes a different instance of the loop. The code is shown below

```
#pragma omp parallel
{//fork a team of threads
int id,i,Nthrds, start, end;

id=omp_get_thread_num();
Nthrds=omp_get_num_threads();
start=id   * N / Nthrds;
end=(id+1) * N / Nthrds;

if (id==Nthrds-1)
 end=N;

for (i=start; i<end; i++)
 A[i]=A[i]+B[i];
}//all threads join master
```

The recommended solution is shown below

```
#pragma omp parallel for
for (i=0; i<N; i++)
 A[i]=A[i] + B[i];
```

Which is equivalent to

```
#pragma omp parallel
{ //fork a team of threads
#pragma omp for
 for (i=0; i<N; i++)
  A[i]=A[i]+B[i];
} //all threads join master
```

Keep in mind that without the omp parallel clause there are no multiple threads. The threads are created just after the parallel clause, and they join the master thread in the end of that clause '}'.

Task3: Download '*reduction_serial.c*' program. Try to parallelize '*un_opt()*' function. Several different solutions are provided. Make sure you understand them all.

The version1() routine is the recommended one. 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 since a race condition occurs; otherwise, different threads might be writing concurrently to 'ave'. Without using the reduction clause, some threads might be clashing and trying to update the memory at the same time. In this case, the value of 'ave' is unknown.

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.

In version3() routine, the critical construct is used instead, which is explained below. Version3() routine can be rewritten using the atomic construct instead of the critical.

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

Pi Example

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.

PI Version1(): The code follows

```
//THIS IS THE 1ST PARALLEL VERSION - THIS IMPLEMENTATION IS NOT THE FASTEST,
BECAUSE OF THE FALSE SHARING in sum[] array
double version1(){

double step;

int i, nthreads;
double x, pi=0.0, sum[NUM_THREADS];

step=1.0/(double) num_steps;
omp_set_num_threads(NUM_THREADS);

#pragma omp parallel
{
   int i,id,nthrds; //local data
   double x;        //local data
   id=omp_get_thread_num();
   nthrds = omp_get_num_threads();
   if (id==0) nthreads=nthrds; //save a copy of num of threads as the enviroment
might choose to give us less threads than requested. What if we ask 1000 threads?

   for (i=id, sum[id]=0.0; i<num_steps; i=i+nthrds){
      x=(i+0.5)*step;
      sum[id] = sum[id] + 4.0 / (1.0 + x*x);
   }
}

//after this point the local variables are lost. So, for the sum to be visible, we
must promote the sum to an array.
for (i=0, pi=0.0; i<nthreads; i++)
  pi+=sum[i] * step;


return pi;

}
```

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, but sum[0:7] share the same cache line (hardware resource).

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 (Fig.1). 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.

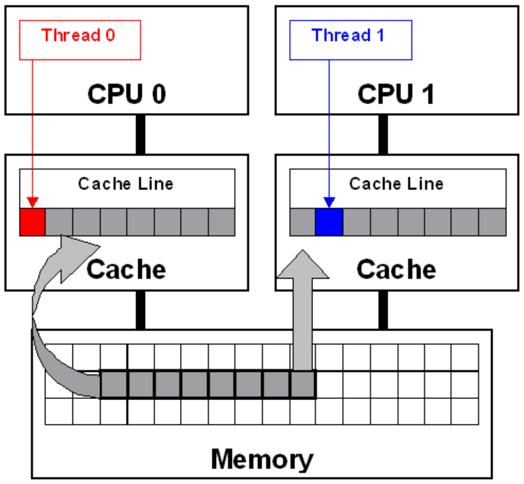

Fig.1. False sharing. Taken from https://software.intel.com/content/www/us/en/develop/articles/avoiding-and-identifying-false-sharing-among-threads.html

PI Version2: This implementation addresses the false sharing problem by using different cache lines to store the threads' sum variables. By doing so, the scalability of this implementation is improved. However, this implementation has an inefficiency. The developer must consider the cache line size and manually amend the PAD variable; thus, if the program runs on another machine (with different cache line size) the PAD value must be amended accordingly. The sum[] array is promoted to a 2-d array sum[NUM_THREADS][PAD]. Keep in mind that only the 1st column contains useful data; 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

PI Version3: This version achieves the same performance as version 2. In this version there is no shared array and thus there is no false sharing. Unlike version 2, this version 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'.

PI 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.

PI Version5/6: These are the recommended versions, showcasing the power of OpenMP. The code of the version5() follows.

```
double version5(){

int i;
double pi, sum=0.0;
double step;

  step=1.0/(double) num_steps;

#pragma omp parallel
{
 double x;
#pragma omp for reduction(+:sum) //Each thread has its own copy of sum. Each
thread does its own summation and when they are done, they are combined with the
global copy of sum.
 for (i=0; i<num_steps; i++){
  x=(i+0.5)*step;
  sum = sum + 4.0 / (1.0 + x*x);
}
}
pi = step * sum;


return pi;

}
```

Version6() is even more elegant as we are using just a single #pragma line. 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. We will learn more about this next week.

Part 2 - Dot Product Example and the Schedule Clause

Task1: Study the '*dot_prod_serial.c*' program. Drawing upon what you have learned so far, try to parallelize the '*dot_prod_serial ()*' routine. The solution is provided in `dot_prod_parallel_ver1() routine in` '*dot_prod_parallel.c*' file.

Task2: Study the other solutions provided in '*dot_prod_parallel.c*' file for Task1. Run '*dot_prod_parallel_ver3()*' routine for N=8 including both 'schedule(dynamic)' and '*schedule(static)*' options. The results will look like (these new omp scheduling clauses are explained in the next page):

Schedule (static): Each thread is assigned to two iterations

```
Number of threads = 4
Thread 0 is starting...
Thread 0: executes iteration i= 0
Thread 0: executes iteration i= 1
Thread 1 is starting...
Thread 1: executes iteration i= 2
Thread 2 is starting...
Thread 2: executes iteration i= 4
Thread 2: executes iteration i= 5
Thread 1: executes iteration i= 3
Thread 3 is starting...
Thread 3: executes iteration i= 6
Thread 3: executes iteration i= 7
```

Schedule (dynamic): This option assigns the loop iterations into 2 threads only.

```
Number of threads = 4
Thread 0 is starting...
Thread 0: executes iteration i= 0
Thread 0: executes iteration i= 1
Thread 0: executes iteration i= 2
Thread 0: executes iteration i= 3
Thread 0: executes iteration i= 4
Thread 0: executes iteration i= 5
Thread 3 is starting...
Thread 3: executes iteration i= 7
Thread 2 is starting...
Thread 1 is starting...
Thread 0: executes iteration i= 6
```

schedule(static, chunk), where ***chunk***=4: Chunk specifies that each thread will execute 4 iterations.

```
Number of threads = 4
Thread 0 is starting...
Thread 0: executes iteration i= 0
Thread 0: executes iteration i= 1
Thread 0: executes iteration i= 2
Thread 0: executes iteration i= 3
Thread 2 is starting...
Thread 3 is starting...
Thread 1 is starting...
Thread 1: executes iteration i= 4
Thread 1: executes iteration i= 5
Thread 1: executes iteration i= 6
Thread 1: executes iteration i= 7
```

Schedule(dynamic, chunk), where chunk=4: In the general case, where N is large, each thread asks for which iterations to execute, then executes 4 iterations of the loop, then asks for more, and so on. In this case, thread 0 asked for 4 iterations, it executed them and the scheduler decided to give the other 4 iterations to thread 0.

```
Number of threads = 4
Thread 0 is starting...
Thread 0: executes iteration i= 0
Thread 0: executes iteration i= 1
Thread 0: executes iteration i= 2
Thread 0: executes iteration i= 3
Thread 0: executes iteration i= 4
Thread 0: executes iteration i= 5
Thread 0: executes iteration i= 6
Thread 0: executes iteration i= 7
Thread 2 is starting...
Thread 1 is starting...
Thread 3 is starting...
```

The schedule clause: It affects how loop iterations are mapped onto threads.

- ***Schedule (static, [,chunk])*** . [ ] is optional. Assigns blocks of iterations of size chunk to each thread. Used when the number of iterations is pre-determined and predictable in advance. Scheduling is done at compile time. OpenMP divides the number of iterations into chunks that are approximately equal in size, and it distributes at most one chunk to each thread.
- ***Schedule (dynamic, [,chunk])****. Omp scheduler d*ecides at runtime which iterations will be allocated to each thread. Each thread grabs chunk iterations off a queue until all iterations have been handled. Used when the number of iterations is unpredictable, highly variable work per iteration. Scheduling is done at runtime. In the dynamic schedule, there is no predictable order in which the loop items are assigned to different threads. Each thread asks the OpenMP runtime library for an iteration number, then handles it, then asks for the next, and so on, e.g., each thread asks for a chunk of iterations, executes them, then asks for another chunk, and so on. There is no particular order in which the chunks are distributed to the threads. The order changes each time when we execute the for loop. The dynamic scheduling type is appropriate when the iterations require

different computational costs. This means that the iterations are poorly balanced between each other. The dynamic scheduling type has higher overhead than the static scheduling type because it dynamically distributes the iterations during the runtime.

- ***Schedule (guided, [,chunk]).*** Threads dynamically grab blocks of iterations. The size of the blocks starts large and shrinks down to size chunk as the calculation proceeds. **Not really used often**. The guided scheduling type is similar to the dynamic scheduling type. OpenMP again divides the iterations into chunks. Each thread executes a chunk of iterations and then requests another chunk until there are no more chunks available. The difference with the dynamic scheduling type is in the size of chunks. The size of a chunk is proportional to the number of unassigned iterations divided by the number of the threads. Therefore, the size of the chunks decreases. The guided scheduling type is appropriate when the iterations are poorly balanced between each other. The initial chunks are larger, because they reduce overhead. The smaller chunks fills the schedule towards the end of the computation and improve load balancing. This scheduling type is especially appropriate when poor load balancing occurs toward the end of the computation.
- ***Schedule (runtime).*** Schedule and chunk size taken from the omp_schedule enviroment variable (or the runtime library). Used when we are not sure about which one is best (static or dynamic)
- ***Schedule (auto)***. Schedule is left up to the runtime to choose (does not have to be any of the above). This option is new in OpenMP. It lets the compiler to decide and do its best.

Task3: Run '*dot_prod_parallel_ver2()*' routine for a large N value and measure the execution time for schedule(dynamic) vs schedule(static). Do not specify the chunk size. You will find out that the dynamic scheduling type is much slower (has higher overhead) than the static scheduling type because it takes decisions at runtime.

Task4: Repeat Task3 again but use schedule (dynamic, chunk) instead of schedule(dynamic), where chunk=N/4. Now it runs faster than before. Why? Because it did not decide about the chunk size at runtime.

Omp simd construct

OpenMP provides a set of compiler directives that are used to provide extra information to a compiler to allow it to automatically parallelise and/or vectorise code (typically loops). These are built into the compiler and accessed by using pragmas (via #pragma). Pragmas are hints that the compiler can choose to use or ignore, depending on whether it has built-in support for that capability. OpenMP 4.0 introduced omp simd, accessed via #pragma omp simd as a standard set of hints that can be given to a compiler to encourage it to auto-vectorise code.

Compilers may not vectorize loops when they are complex or possibly have dependencies, even though the programmer is certain the loop will execute correctly as a vectorized loop. The simd construct assures the compiler that the loop can be vectorized.

Be careful. Using omp simd bypasses the compiler analysis. So, use with caution as

- Incorrect results are possible
- Poor performance is possible
- memory errors are possible

How to use 'omp simd' in Visual Studio (VS): This is a new feature. Visual Studio 2019 now offers SIMD functionality via command line. To do so, go on view tab and select 'terminal'. Navigate to the directory where the source code is located; to this end, you can use the ‘cd’ command. Type ‘cl source.cpp -openmp:experimental’. A file ‘source.exe’ will be created. Run this file using ‘./source.exe’. If you want to take advantage of the compiler’s optimization options (they are shown in project->properties->C/C++->optimization), then type ‘cl -O2 source.cpp -openmp:experimental’. If you want to see which loops are vectorized type ‘cl -O2 -Qvec-report:2 main.cpp –openmp:experimental’. Note that VS does not support most of the ‘omp simd’ clauses such as reduction, aligned etc; in this case, a warning will be shown.

Remember from the vectorization lab session that the arrays must be memory aligned, otherwise the performance will be poor.

We can either allocate aligned memory statically using

float A[N] __attribute__((aligned (64))); //Linux only

__declspec(align(64)) float A[N] //Visual studio only

or dynamically using

*_mm_malloc (N * sizeof(float),64); //dynamically allocates memory 64byte aligned*

#pragma omp simd : The simd construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions).

aligned(y,x,a:64) : Data alignment is important for SIMD instructions. Unaligned memory accesses are normally slower than aligned memory accesses (unaligned accesses cross a cache-line boundary). Additionally, some SIMD instructions can only be used with aligned memory addresses. However, the compiler often cannot determine the alignment properties of data that is linked from other files or when they are dynamically allocated. The aligned clause asserts to the compiler that a variable is aligned. Each pointer in the aligned clause can have a positive integer alignment applied to it. If no alignment value is given to the compiler, an implementation defined default value is assumed. Using this clause allows the compiler to safely use SIMD instructions that have strict alignment requirements. If this clause is used, the programmer is responsible for ensuring that the data is in fact aligned. Otherwise, the attempted use of aligned memory accesses on unaligned memory may result in segmentation faults [1].

reduction(+:tmp) : The reduction clause instructs the compiler to perform a vector reduction on a variable. A reduction operation is performed by computing a partial value inside the parallel region. When the parallel region ends, the partial values are then aggregated into the final value. The reduction clause does this by creating a private vector copy of the variable inside the SIMD loop which is used to store the partial values. When the SIMD region ends, the vector copy of the original variable is horizontally aggregated. The final value is then moved from the vector copy to the original variable. The reduction clause takes a character representing the type of reduction performed in the loop and a variable to be reduced inside the loop [1].

OpenMP supports a rich set of simd related clauses, but in this module we will not go further.

How can we be sure that the compiler vectorized the code? To be verify that, compile using *-fopt-info-vec-optimized* option (gcc only) or check the assembly code or compare the performance to the serial version**.** In VS, use the '-Qvec-report:2' option.

Task5: Try to amend '*dot_prod_parallel_ver2()*' routine in '*dot_prod_parallel.c*' program in order to allow vectorization. The solution is provided in '*dot_prod_parallel_ver4()'*.

Changing the storage attributes (just an introduction)

We will be discussing about this next week thoroughly, but for now you must know that we can define in the #pragma clause whether a variable is shared amongst all threads or is private.

- ***shared*** (a): all threads can access 'a'. 'a' has been defined outside of the parallel region
- ***private*** (a): each thread creates an un-initialized copy of 'a'

For example, in the following code, we tell the compiler that 'i, j' are private variables and 'Y' is shared (it has been defined outside of the parallel region). The 'i' loop is the loop next to the omp clause and it is private by default. However, 'j' is not private by default and thus it must be defined as private, manually. 'Y' is shared by default as it has been declared outside of the parallel region, so the 'shared(Y)' clause is not needed, but it helps readability and might prevent bags.

```
#pragma omp parallel for shared(Y) private(i,j)
  for (i=0; i<N; i++)
   for (j=0; j<N; j++)
      Y[i]=…
```

Matrix-Vector Multiplication Example

Task6: Download and study '*MVM_serial.c'*. Try to parallelize the '*MVM_serial()'* routine.

Task7: Study the solutions provided in '*MVM_parallel.c'* program.

'*MVM_parallel_ver1()*' and MVM_parallel_ver2() are almost identical. When *'#pragma omp parallel for'* is applied to a loop, its iterator becomes private by default. However, if there are nested loops, we must manually specify them as private variables. The y, a, x arrays are shared by default, as they are defined outside the parallel region. So the shared(y,a,x) is not needed; however, it is good practice as unexperienced users forget that.

Task8: Measure the execution time of MVM_parallel_ver3() for N=4096 and NUM_THREADS=[1,2,4,8]. Repeat for N=1024 and N=128.

You will figure out that the program scales well only for large input sizes. This is because the overhead for creating and synchronizing the threads is comparable to the threads' execution time. If the *Tserial/Tparallel* value is close to the number of threads used, then the scalability value is considered high. The code will scale well only when each thread executes at least a minimum number of instructions. The number of instructions per thread is found experimentally and depends on the target platform.

Task9: Study the '*MVM_parallel_ver4()*' routine. This implementation uses both multi-threading and vectorization. The '*aligned(y, x, a : 64)*' is not necessary, but it can improve performance as it informs the compiler that the arrays are 64byte aligned.

Task10: Study the '*MVM_parallel_ver5()*' routine. This implementation also uses both multi-threading and vectorization, but the vectorization has been applied using x86-64 AVX intrinsics. Compare the performance of *MVM_parallel_ver4()* and *MVM_parallel_ver5()* routines.

You will figure out that their execution times are similar. However, note that the x86-64 implementation provided is not the most efficient; first, it can be re-written in a more efficient way and second, we can include other optimizations which cannot be applied in the OpenMP version, such as register blocking. Although, the '*omp simd*' construct works well for simple programs like above, the x86-64 AVX intrinsics can provide improved performance in the general case; especially when intrinsics are combined with advanced optimizations.

Task11: study the *MVM_parallel_ver6()* routine. Use N=1024 and compare the execution time to *MVM_parallel_ver5()* routine for NUM_THREADS=[1,2,4]. Why does this routine perform that better? Register blocking has further reduced the number of L/S instructions. This routine is not performance efficient for larger N values, because the number of dL1 misses increases (multiple 'a' rows and not one are loaded into dL1).

Task12: compute the FLOPS (floating point operations per second) achieved by a) the serial version, b) vectorized version, c) multithreaded version, d) vectorized and multithreaded version. Use N=[128,512,1024,2048,4096,8192]. The results in a quad core Intel CPU are shown in Fig.1. The FLOPs value is given by the Eq.1. Our algorithm does $2N^2$ operations and thus Eq.1 gives Eq.2.

FLOPS=number of FP arithmetical operations / time in seconds (1)

FLOPS=$2N^2$/time in seconds (2)

For large input sizes the FLOP values are lower as the arrays cannot fit in the precious cache memories. This phenomenon is shown in its extreme in the yellow line; the number of LLC misses is increased by 3.5% and therefore performance is dropped. Further understanding this figure is out of the scope of this module. Performance can be further improved by reducing the number of memory accesses and cache misses, e.g., **the implementation provided in Task11 achieves 92 GigaFLOPS** on the studied PC. Optimizations such as register blocking, loop tiling and software prefetching boost performance a lot.

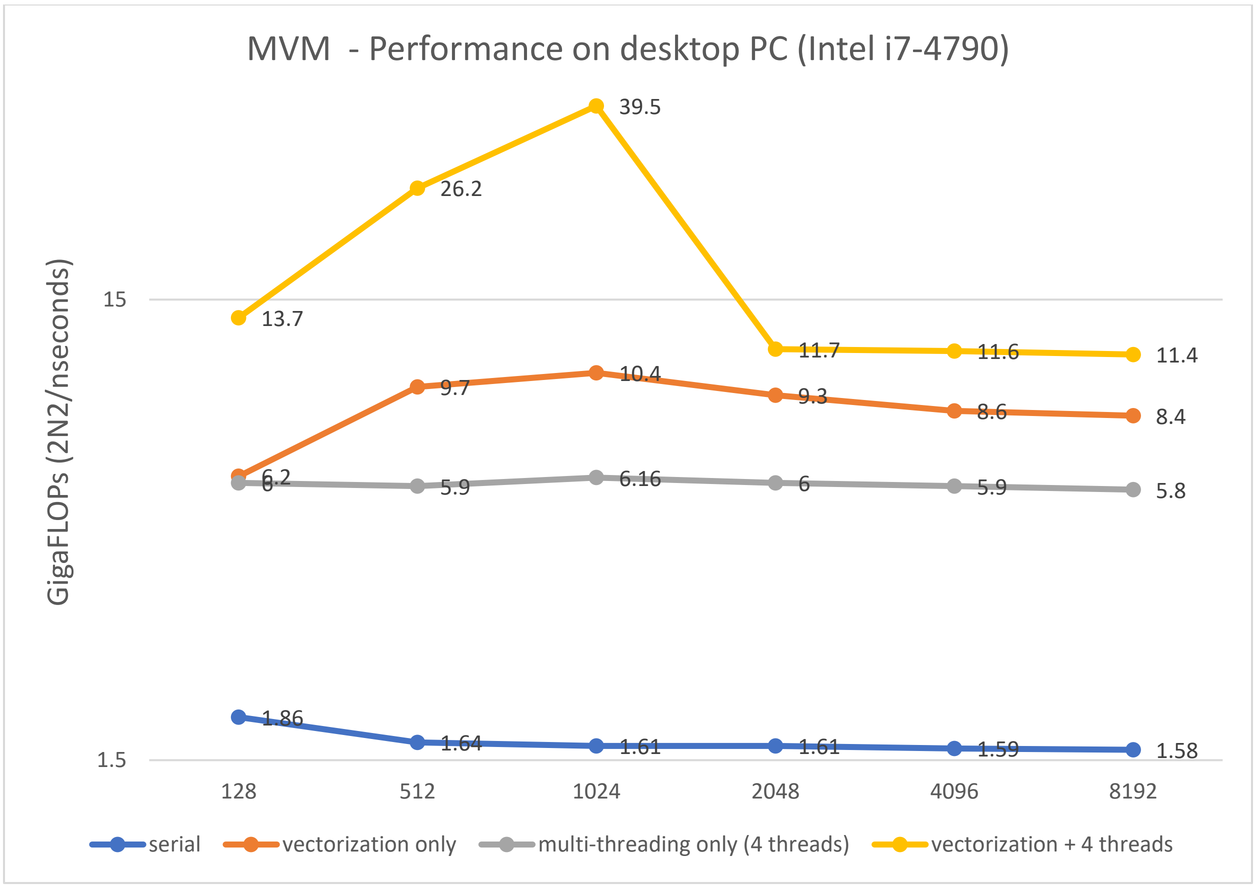

Fig.1 MVM Performance in FLOPs

Part 3 - More advanced OpenMP clauses and barriers (optional)

Single construct - #pragma omp single : The single construct specifies that the given statement/block is executed by only one thread. It is unspecified which thread. Other threads skip the statement/block and wait at an implicit barrier at the end of the construct. Do not assume that the single block is executed by whichever thread gets there first. According to the standard, the decision of which thread executes the block is implementation-defined.

An '*omp single'* example follows:

```
#pragma omp parallel {
  funct1();  //all threads execute this
  #pragma omp single
  {
  funct2(); //just one thread executes this
  } //other threads wait here for the single thread to finish
 funct3(); //all threads execute this
}
```

Master construct - #pragma omp master : The master construct is similar to single, except that the statement/block is run by the *master* thread, and there is no implied barrier; other threads skip the construct without waiting. The following two examples are equivalent.

```
#pragma omp parallel //example1
{
 funct1();
#pragma omp master  {
 funct2();
 }
funct3();
}


#pragma omp parallel //example2, equivalent to example1
{
 funct1();
  if(omp_get_thread_num() == 0)  {
   funct2();
  }
funct3();
}
```

#pragma omp barrier: The barrier directive causes threads encountering the barrier to wait until all the other threads have encountered the barrier. Each thread waits at the barrier until all threads arrive. *Note*: There is an implicit barrier at the end of each parallel block, and at the end of 'omp for' and 'omp single' statement, unless the nowait directive is used. A Barrier construct example is shown below:

```
#pragma omp parallel
{
int id=omp_get_thread_num();
A[id]=funct1(id);

#pragma omp barrier //no thread will execute funct2, before A[] is stored
B[id]= funct2(id, A);
}
```

#pragma omp nowait: The 'nowait' directive overrides the barrier implicit at the end of a directive. The nowait directive can only be attached to the sections for and single. It cannot be attached to the within-loop ordered clause, for example. An example follows:

```
#pragma omp parallel
{
  #pragma omp for nowait
    for (i=1; i<n; i++)
       b[i] = (a[i] + a[i-1]) / 2.0; //threads will not wait here. If a thread finishes before the others, it will
continue

  #pragma omp for nowait
    for (i=0; i<m; i++)
      y[i] = sqrt(z[i]); //threads will not wait here. If a thread finishes before the others, it will continue
}
```

Collapse clause: Two or more loops are merged to one and are parallelized. Used when the number of iterations in the loop is small. Use the collapse-clause to increase the total number of iterations that will be partitioned across the available number of OMP threads. An example follows

```
#pragma omp parallel for collapse(2)
for ( i=0; i<15; i++) //there are only 15 iterations to parallelize. Using collapse there will be 1200
  for( j=0; j<80; j++)
        func(i,j);
```

Loop nesting: Nested parallelism" is disabled in OpenMP by default (it can be used though), and thus a second pragma will be ignored at runtime. An example follows:

```
#pragma omp parallel for
for ( i=0; i<15; i++)
  #pragma omp for // This is ignored, nesting like this is not allowed by default.
  for( j=0; j<80; j++)
   func(i,j);
```

Changing the storage attributes

- ***shared*** (a): all threads can access 'a'
- ***private*** (a): each thread creates an un-initialized copy of 'a'
- ***firstprivate*** (a): each thread creates an initialized copy of 'a'

- ***lastprivate*** (a) : the value of 'a', of the last iteration of the loop, is stored back as global. If a loop goes from i=[0,N-1], then the thread that executed the iteration N-1, its value of tmp will be copied out to the global scope.
- ***default (private | shared | none)***. The default clause forces a programmer to explicitly specify the data-sharing attributes of all variables in a parallel region. Using this clause then forces the programmer to think about data-sharing attributes. This is beneficial because the code is clearer and has fewer bugs. Default(shared) clause makes all the variables shared. Another usage of default(shared) clause is to specify the data-sharing attributes of the majority of the variables and then additionally define the private variables, e.g., #pragma omp parallel for default(shared) private(a, b). You can also write parallel regions with the default(none) clause and then specify the private and shared ones. The default clause is to check whether you have remembered to consider all variables as private/shared, using the default (none) setting.

Two examples follow. The first example is wrong as *private(tmp)* does not initialize the variable, it just creates it. However, *firstprivate()* will initialize it.

```
void wrong(){
int tmp=0;
#pragma omp parallel for private(tmp) //create a var tmp that is private (un-initialized)
 for (i=0; i<N; i++)
  tmp+=j;  //the first value of tmp is not zero

printf (tmp); //problem, will see the global tmp, not the private. The private tmp is disappeared
}

Void good(){
tmp=1;
#pragma omp parallel for firstprivate(tmp) //create a var tmp that is private and initialized
for (i=0; i<N; i++) {
  if ((i%2)==0)
    A[i]=tmp;
Else
    A[i]=0;
}
}
```

Task1: Consider the following code. Are a,b,c local to each thread or shared inside the parallel region?

```
int a=1,b=1, c=1;

#pragma omp parallel private(b) firstprivate(c)

{ }
```

Answer: a is shared, while b and c are local

Task2: What are the a,c,b initial values inside the parallel region and after the parallel region, in the code above?

Answer: inside the region a=1, b=undefined, c=1. After the region a=b=c=1. b and c revert to their original values.

Omp sections and tasks

omp sections: The section construct is one way to distribute different tasks to different threads. The sections construct indicates the start of the construct. Each section refers to a different task and is executed by one only thread. If there are more threads than tasks, some of the threads will be idle. An example follows

```
    #pragma omp sections
    {
      #pragma omp section  //just one thread executes this section
     { funct1(); }
     #pragma omp section //just one thread executes this section
    { funct2();
    funct3(); }
    #pragma omp section   //just one thread executes this section
    { funct4(); }
  } //the other threads wait here
```

The code above indicates that any of the routines funct1, (funct2, funct3) and funct4 may run in parallel, but that funct2 and funct3 must be run in sequence. Each function is executed exactly once. As usual, if the compiler ignores the pragmas, the result is still a correctly running program.

Task3: study the 'sections.c' program.

Unlike the previous examples, where they were based on **loop parallelism**, this example exploits **task parallelism**; two different loop kernels are executed in parallel. Pay attention to the '*nowait*' clause. This allows for the threads not to wait in the section. If you cannot understand this, remove the 'nowait' clause and run the program several times to see what the output is.

Task4: Study the '*mandel_serial.c*' program. The 'mandel_parallel.c' contains an OpenMP parallel implementation of the '*mandel_serial.c*' program, but it does not work properly. Can you spot the problem? The solution is provided in 'mandel_solution.c' file.

Task5: Study the 'find_the_problem1.c' program. This program is problematic. Can you spot the problem? *Tip: think of the variables scope (private vs shared).*

Task6: Study the 'find_the_problem2.c' program. This program prints a 'Segmentation fault (core dumped)' error message, which means that memory has been violated. Can you spot the problem?

Answer: Array A is private, which means that every thread will try to allocate an array of size NxN. The memory segment that is used, is the stack, not the heap, as A[][] is a private array; the size of the array is very large and the program cannot allocate such space on the threads' stack. This makes the program to crash.

Omp task: When a thread encounters a task construct, a task is generated from the code for the associated structured block. The encountering thread may immediately execute the task or defer its execution. In the latter case, any thread in the team may be assigned the task. If deferred, the task is placed in a conceptual pool of tasks associated with the current parallel region. All team threads will take tasks out of the pool and execute them until the pool is empty. **A thread that executes a task might be**

different from the thread that originally encountered it. The code associated with a task construct will be executed only once. The tasks can be executed at any order. **Omp task is an advanced and new OpenMP feature and it will not be assessed by this module.** An example follows

```
#pragma omp parallel
{
 #pragma omp single
 {
  #pragma omp task
     { funct1(); }
  #pragma omp task
     { funct2(); }
  #pragma omp task
     { funct3(); }
 }
}
```

Omp taskwait: The taskwait construct specifies a wait on the completion of child tasks of the current task. An example follows

```
#pragma omp parallel
{
 #pragma omp single
 {
  #pragma omp task
     { funct1(); }
  #pragma omp task
     { funct2(); }
  #pragma omp task
     { funct3(); }
 #pragma omp taskwait //all the tasks must end here
 }
}
```

Design Patterns for Parallel Programing

No matter which programming language we use (MPI, OpenMP, OpenCL) there are specific algorithmic concepts for parallel programming that are universal. These are the design patterns for parallel programming. So far, we have used just one programming language (OpenMP) and two different design patterns. The design patterns we have seen so far are the a) **Single Program Multiple Data (SPMD) pattern,** and b) **Loop parallelism pattern;** additionally, we are about to study the **divide and conquer design pattern**.

In the SPMD pattern, a single program runs on many processing elements and creates a collection of units of execution (here threads) and each one will run the same program. Remember Pi program and version1(), version2() and version3() routines; these routines are examples of this pattern. Furthermore, most of the MPI programs use this pattern. As far as the loop parallelism pattern, we have seen many examples already and it is the pattern most used in OpenMP and shared memory hardware architectures.

Divide and Conquer algorithm

In computer science, divide and conquer is an algorithm design paradigm based on multi-branched recursion. A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem (Fig.1) [https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm#Parallelism].

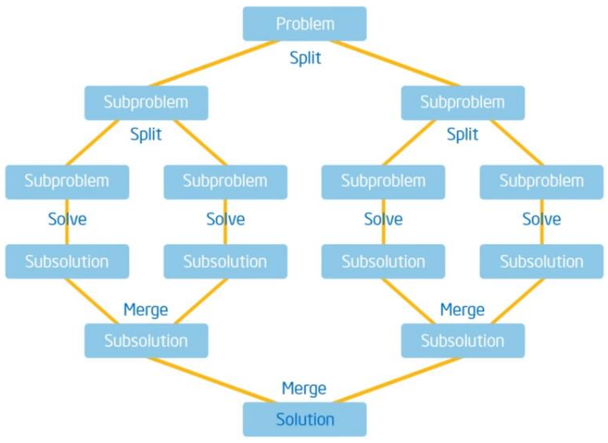

Fig.1 Divide and conquer algorithm

Fibonacci sequence

In mathematics, the Fibonacci numbers, commonly denoted F_n, form a sequence, called the Fibonacci sequence, such that each number is the sum of the two preceding ones, starting from 0 and 1. $F_0=0$, $F_1=1$ *and* $Fn=F_{n-1}+F_{n-2}$. The beginning of the sequence is thus: *0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, … .* The serial code is shown below (as a recursive function). Fig.2 shows the recursive calls for n=5.

```
int fib (int n){
int x,y;
if (n<2)
 return n;

x=fib (n-1);
y=fib (n-2);
return x+y;
}
```

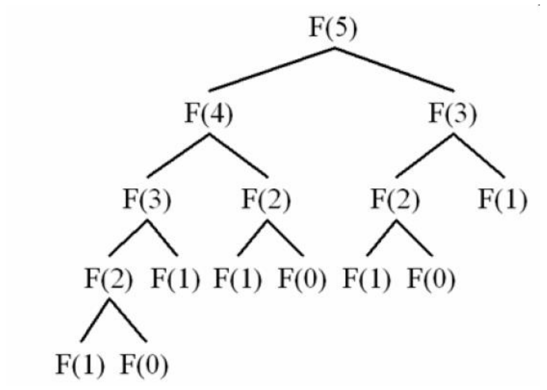

Fig.2 Fibonacci recursive function calls for n=5

Task7: In fib.c program, a parallel version of the Fibonacci sequence program is provided using omp tasks and divide and conquer design pattern. Study the program and make sure you understand how it works.

Pi Program using the Divide and Conquer design Pattern

Task8. Drawing upon the Fibonacci program, try to implement a parallel version of the PI program using the divide and conquer approach. Use omp tasks.

A solution is provided in '*PI_divide&conquer.c'* file. the main idea behind this implantation is to recursively split the pi program's loop into half until the number of iterations is smaller than a threshold.

In the divide_conquer() routine, the appropriate initializations are made and just a single thread executes the pi_kernel() routine. The pi_kernel() routine is an iterative function which consists of three parts. First, an if-condition checks whether the problem is small enough to be executed in a single thread; to do so, a constant 'BULK' is defined which refers to a threshold; this threshold contains the number of iterations each thread will execute. If the problem size is small (or equivalently if the number of iterations is small) it is executed by single thread. Otherwise, it splits the problem into two equal parts and it assigns them to two threads; the two new threads call the pi_kernel() again. Last, a taskwait is needed to make sure that the results of the two sub-problems are merged.

ENVIROMENT VARIABLES.

The OpenMP specification defines several **environment variables** that control the execution of OpenMP programs.

OMP_NUM_THREADS : Sets the number of threads to use during execution of a parallel region. You can override this value by a **NUM_THREADS** clause, or a call to **OMP_SET_NUM_THREADS()**.

OMP_STACKSIZE : Sets the stack size for each thread. The value is in kilobytes.

OMP_WAIT_POLICY : The OMP_WAIT_POLICY environment variable provides a hint to an OpenMP implementation about the desired behavior of waiting threads. It can be either ACTIVE or PASSIVE. In active, the thread actively spins waiting for something to be available. This consumes CPU power. In passive, the thread is put into sleep. Putting a thread into sleep and waiting it up, costs a lot. If you believe your program is not going to wait long, then use active.

OMP_PROC_BIND : It can be either true or false. It sets the thread affinity policy to be used for parallel regions at the corresponding nested level. If the environment variable is set to false, the execution environment may move a thread to another CPU core. If it is true, threads are not shuffled among the cores. Use true for cache intensive algorithms.

GOMP_CPU_AFFINITY : Bind threads to specific CPU cores. 'For example, GOMP_CPU_AFFINITY="0 3 1-2 4-15:2" will bind the initial thread to CPU 0, the second to CPU 3, the third to CPU 1, the fourth to CPU 2, the fifth to CPU 4, the sixth through tenth to CPUs 6, 8, 10, 12, and 14 respectively and then start assigning back from the beginning of the list. GOMP_CPU_AFFINITY=0 binds all threads to CPU 0.', taken by [5].

To learn more about environment variables visit https://www.openmp.org/spec-html/5.0/openmpch6.html .

Example (Linux only): to set OMP_PROC_BIND=true in Linux, type the following commands:

```
export OMP_PROC_BIND=TRUE //sets it
echo $OMP_PROC_BIND //prints it, to make sure it worked
```

Example (Linux only): In Linux, it is very easy to manually pass the environmental variables to every binary. For example, the following command will run force openmp runtime to use only 4 threads and each map thread0 to core0, thread1 to core1, thread2 to core2 and thread3 to core3.

```
GOMP_CPU_AFFINITY='0,1,2,3' OMP_NUM_THREADS=4  ./executable
```

You can print the core id that each thread is allocated to, by using the following code

```
#pragma omp parallel
{
int core_id = sched_getcpu();
int id = omp_get_thread_num();
printf("Thread %d is running on core %d", id, core_id);
}
```

To use *sched_getcpu()* function you must include the following code in the beginning of your code (before any other header file) ***'#define _GNU_SOURCE'***.

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
3. GNU libgomp available at https://gcc.gnu.org/onlinedocs/libgomp/index.html
4. GOMP_CPU_AFFINITY, available at https://gcc.gnu.org/onlinedocs/libgomp/GOMP_005fCPU_005fAFFINITY.html