# Objectives

* To parallelize C programs using OpenMP
* To vectorize C programs using OpenMP
* To evaluate different scheduling parameters and implementations and analyze performance trade offs

**Task1**: Study the ‘*dot\_prod\_serial.c*’ program and try to parallelize the ‘*dot\_prod\_serial ()*’ routine.

**Task2:** Study the Task1 solutions provided in ‘*dot\_prod\_parallel.c*’. Run ‘*dot\_prod\_parallel\_ver3()*’ routine for N=8 including both ‘schedule(dynamic)’ and ‘*schedule(static)*’ options. All different omp scheduling clauses are explained in the next page. The results will look like hereafter:

***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 amount of iterations is pre-determined and predictable in advance. Scheduling is done at compile time. OpenMP divides 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 amount 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 next, and so on, e.g., each thread asks for an iteration number, executes 3 iterations of the loop, then asks for another, 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.

**Task5:** Download and study ‘*MVM\_serial.c’*. Try to parallelize the ‘*MVM\_serial()’* routine.

**Task6:** 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.

**Task7:** 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.

# 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 autovectorise 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.

**How to use ‘omp simd’ in Visual Studio**: 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’. Type ‘cl source.cpp -openmp:experimental’. A file ‘source.exe’ will be created. Run this file using ‘./source.exe’.

**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

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

We can either statically memory align the arrays using

*float A[N] \_\_attribute\_\_((aligned (64))); //Linux only*

*\_\_declspec(align(64)) float A[N] //Visual studio only*

Or we can dynamically align the arrays 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 always slower than aligned memory accesses if they 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**.**

**Task8**: 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.

**Task9**: 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. Although, the ‘*omp simd’* construct works well for simple programs like that, the x86-64 AVX intrinsics can provide improved performance in the general case. Furthermore, the latter can be combined with advanced optimizations like register blocking.

**Task10:** 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).

**Task11:** 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()’*.

**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 2N2 operations and thus Eq.1 gives Eq.2.

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

*FLOPS=2N2/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 LLC misses is increased by 3.5% and this is why 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 Task10 achieves 92 GigaFLOPS. Optimizations such as register blocking, loop tiling and software prefetching boost performance a lot.

*Fig.1 MVM Performance in FLOPs*

# Further Reading

1. Effective Vectorization with OpenMP 4.5, available at <https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&cad=rja&uact=8&ved=2ahUKEwjCksHTqr3qAhX4SRUIHSmlBYAQFjAAegQIBhAB&url=https%3A%2F%2Finfo.ornl.gov%2Fsites%2Fpublications%2Ffiles%2FPub69214.pdf&usg=AOvVaw22CMKDJzHHKHKSFzm8P9qr>
2. Chapter 51 in 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=2ahUKEwjIooTyqr3qAhWYaRUIHZmEC58QFjAAegQIAhAB&url=https%3A%2F%2Fwww.openmp.org%2Fwp-content%2Fuploads%2Fopenmp-examples-4.5.0.pdf&usg=AOvVaw3BDlLKC3VhdJI1iTj1RE_p>