Schedule and Exercises for OpenMP

Schedule for talks and exercises

Day 1		Day 2	
9:00-9:45	Introduction to OpenMP	9:00-10:00	Synchronization constructs and program correctness
9:45-10:30	First exercise session	10:00-11:00	Fifth exercise session
10:30-11:15	Simple Work Sharing, Data Scoping and Synchronization	11:00-12:00	Tasking
11:15-12:15	Second exercise session	12:00-12:45	Lunch Break
12:15-13:00	Lunch Break	12:45-13:45	Sixth exercise session
13:00-14:30	Reductions; More on work sharing; Thread-private variables	13:45-15:15	Architectural Performance issues: Affinity, False Sharing
14:30-15:30	Third exercise session	15:15-16:15	Seventh exercise session
15:30-16:15	Vectorization (SIMD)	16:15-17:00	Architectural Performance issues: Exploiting bandwidth
16:15-17:00	Fourth exercise session		

First exercise session

Preparing the working environment (15 minutes)

1. Please log in to the front end machine with the command
   ssh vsc40000@login.hpc.ugent.be
   Note: replace vsc40000 with your own VSC user ID, see https://account.vscentrum.be
2. Please follow the instructions for setting up and connecting to a VNC session that are available at http://hpc.ugent.be/userwiki/index.php/User:VNC (requires login).

Once you have the VNC session running, you can do any of the following:

- Open a terminal
- Use the nano, vi or emacs editors to edit program text
- Load a compiler module:

  ```
  module load intel/2016a        # Intel compilers
  module load GCC                # GCC compilers
  ```

  (with the command `module avail GCC`, you can produce a list of available versions, and specifically load the one you wish to use)
- Load a tools module:

  ```
  module load Inspector/2016_update3
  module load Vtune/2016_update3
  ```
- Execute a batch job on a dedicated node (for reliable performance)

  ```
  qsub -I -l nodes=1:ppn=16 -l walltime=2:0:0
  ```

once the shell prompt returns, all commands are run on a worker node, **until the session is exited.** It is a good idea to use a separate terminal from that used for compiling programs.

Please copy the exercise templates to your HOME directory with

```
cp -a /var/tmp/openmp_exercises/skeletons/*   $HOME
```

The solutions will be made available after each exercise in the directory

```
/var/tmp/openmp_exercises/solutions
```

Getting acquainted with the compilers (30 minutes)

The folder HELLO contains serial code (Fortran and C) that corresponds to the slide example.

1. Add the OpenMP directives that are needed to execute `f()` in parallel; also modify `f()` to print out which thread is working on an instance of `f()`.
2. Find the name of the OpenMP switch for your compiler and build an executable; the supplied `Makefile` may be used for this purpose. Run the executable with 1, 4 and 8 threads.
3. Add code to `f()` instructing it to sleep for `omp_get_thread_num()` seconds. Then, measure the execution time of the resulting program using the UNIX time command. What do you observe?
4. Set the environment variable `OMP_DISPLAY_ENV` to either “true” or “verbose”. Then rerun the program and observe.

Second exercise session

Of the following two exercises, only the first is obligatory.

Parallelization of a code for calculating π using random numbers (30 minutes)

The quarter circle in the first quadrant with origin at (0,0) and radius 1 has an area of π/4. Look at the random number pairs in [0, 1] × [0, 1]. The probability that such a point lies inside the quarter circle is π/4, so given enough statistics we are able to calculate π using this "Monte Carlo" method. You can find a serial version (C and Fortran) in the samples folder PI. It prints its run time and the relative accuracy of the computed approximation to π.

Parallelize the code using OpenMP. Use the `rand_r()` function to get **separate** random number sequences for all threads. What is the best relative accuracy that you can achieve with twelve cores in five seconds of wall time?

On recursion and its elimination (30 minutes)

The first part of this exercise is only theoretical and should only take 5 minutes to complete.

1. Consider the following loop structure:

```
x(:) = …; a(:) = …; b(:) = …
!$omp parallel do
do i=1, n
  x(i) = a(i) * x(i-1) + b(i)
end do
!$omp end parallel do
```

   Why exactly is the above program non-conforming? More precisely: for which loop iterations does the work sharing construct run into trouble? How could a program find out which ones are the problematic iterations?

2. Parallelize the loop in the following piece of code using OpenMP (you can pick up the sample code from the folder RECURSION):

C

```
const double up = 1.00001;
double Sn = 1.0;
double opt[N+1];
int n;


for (n=0; n<=N; ++n) {
  opt[n] = Sn;
  Sn *= up;
}
```

Fortran

```
double precision, parameter :: &
       up = 1.00001d0
double precision :: Sn
double precision :: opt(0:ndim)
integer :: n

do n=0, ndim
  opt(n) = Sn
  Sn = Sn * up
end do
```

The parallelized code should work independently of the OpenMP schedule used. Try to avoid, as far as possible, expensive operations that might harm serial performance.

Hint: To solve this problem you might want to use the **`firstprivate`** and **`lastprivate`** OpenMP clauses discussed in the foregoing slide session.

Third exercise session

Triangular matrix-vector multiplication (60 minutes)

As a variant of the full matrix-vector multiply, consider the triangular matrix-vector multiplication,

$$\sum_{k=1}^{j} M_{jk} \cdot x_k = r_j \,, \qquad j = 1 \,..\, n$$

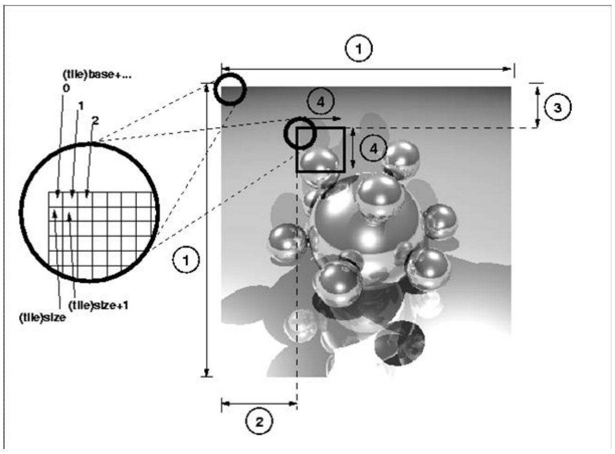
where the summation only runs over the lower triangular part of the n by n matrix M. Starting out with a copy of the serial code provided as a skeleton (in the folder MVM, the programs named tri_mvm*), parallelize the code with OpenMP. Please write parallel variants for both the j-loop and the k-loop as outer loops; what do you need to take care of for the latter? Investigate the baseline performance (compare with the performance of the serial code!) and the scaling behavior for problem sizes n=1000, 2000, 4000, 8000 using between 1 and 16 threads, for both variants. Which of the two performs better? Why can imposing an explicit schedule further improve performance?

(**C programmers**: please use GCC version 6.1 for this exercise)

Fourth exercise session

SIMD operations in a ray tracer (45 minutes)

The RAY folder contains an OpenMP parallel raytracer code (in a Fortran and a C version), which computes a pretty picture. It writes the picture to a file called "`result.pnm`". Look at the file using, e.g., the `display` program. The central function is `calc_tile()`, which computes one tile of the picture. The size of one tile and of the whole picture is hardcoded at the start of the main program. Please vectorize, as far as possible, the serially executed code in the procedures `intersect()` and `shade()` using OpenMP SIMD directives. What speed-up can be achieved? Once you are done, temporarily remove the OpenMP switch from the compilation to assess what fraction of the speed improvement is due to the code restructuring. After the exercise, the solution can be picked up from the subfolder RAY_SIMD.



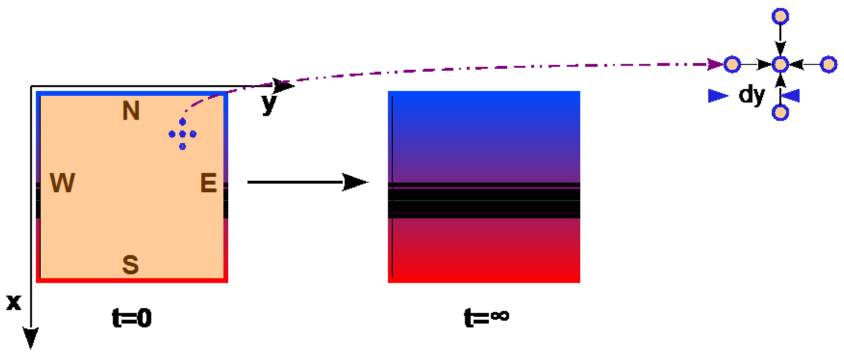
Fifth exercise session

Performance tuning the heat conduction equation (60 minutes)

The folder HEAT contains (quite trivial) OpenMP parallel code that calculates a stationary solution of the heat conduction equation

$$\frac{\partial\Phi}{\partial t} = \frac{\partial^2\Phi}{\partial x^2} + \frac{\partial^2\Phi}{\partial y^2}$$

on a square (two dimensional Jacobi iteration).



Starting out from initial values and (fixed) boundary values, increments are calculated using

$$\delta\Phi = \delta t \cdot [\Delta\Phi]_{\text{discretized}}$$

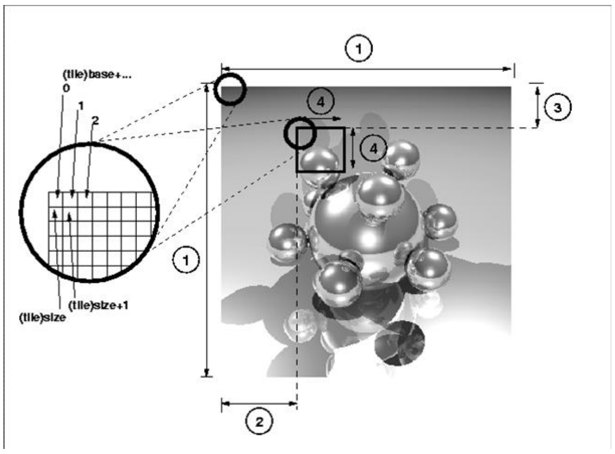
and the process is repeated until the stationary state is reached.

1. Look at the OpenMP directives that were used. Build the program for a problem size of 200 x 200 and run it with 1, 4, 8, 12 threads, noting down the performance numbers.
2. The supplied version of the heat equation solver has its parallel region inside the iteration loop, leading to many forks and joins. Consider reducing the parallel overhead by pulling the parallel region outside the iteration loop. What other changes are necessary to ensure correct execution of the code? It is strongly suggested to use Intel Inspector to identify problems as they arise. Furthermore, building and executing for the small problem size is suggested since the printout immediately indicates whether something has gone wrong.
3. Measure the performance for the improved version for the 200 x 200 problem size, with the same thread counts as in step 1. above. How much improvement do you see?

Sixth exercise session

Task-parallel ray tracer (60 minutes)

We are now returning to the ray tracer code from yesterday. You can either start out from your own solution, or use the program from the RAY_SIMD solution folder. The code assumes that the picture size is a multiple of the tile size. In the version given, the picture size is 2000 by 2000 and the tile size is 200 by 200. The program outputs its run time at conclusion.



Parallelize the code using OpenMP task directives for processing of each tile. Assuming that parallelization should be as coarse-grained as possible, consider which procedure you need to work on. You can deactivate the output for testing, but make sure that your parallel code computes the correct result (this is easy since you can always `display` the picture). What speedup does your code get from 1 to 12 threads? Why does tasking make sense for this type of problem? After the exercise, the solution can be picked up from the subfolder RAY_TASKS.

Seventh exercise session

Performance of OpenMP programs with affinity settings (10 minutes)

Study the performance of both versions of the HEAT code with suitably chosen affinity settings for the problem size 200 x 200. Alternatively, do the same for the RAY tracer or the triangular Matrix-Vector program.

Increasing the problem size in the HEAT example (20 minutes)

Using the `likwid-topology` tool (or hwloc `lstopo` if likwid is not available), determine the size of the largest caches on your compute node. Given the word size of 8 Bytes for a double precision variable, estimate how large the problem in the HEAT code can be if both phi and phin should fit into the cache. Run a problem size at least 4 times as large (you will probably need to fix the maximum iteration count at a lower value for sufficiently short run times) and ensure that the fastest available path is used for all memory accesses.

Parallel histogram computation (30 minutes)

Build the OpenMP program provided in the folder HISTO that calculates a histogram with 16 bins from the results of the standard `rand_r()` random number generator, and run it with 1 and 8 threads, respectively. Even if you can deduct from visual inspection what is going wrong, use the VTune Amplifier to perform an analysis of the code based on the procedure described in the Appendix at the end of the supplied slides. Then, fix the problem in the code and rerun the analysis.