OpenMP Workshop Notes

OpenMP is a multi-threaded parallel specification for execution on virtual memory machines. The scientist has to prefix blocks of code with OpenMP directives which tells the compiler to parallelise that block. The code uses OpenMP environment variables to control program behaviour and is linked against parallel libraries which is done automatically by the compiler.

Below are compilation (and link) commands for the three compilers that support OpenMP:

GNU C compiler: `gcc -fopenmp program.c -o program`
PGI C compiler: `pgcc -mp program.c -o program`
Intel C compiler: `icc -openmp program.c -o program`

GNU Fortran compiler: `gfortran -fopenmp program.f90 -o program`
PGI Fortran compiler: `pgfortran -mp program.f90 -o program`
Intel Fortran compiler: `ifort -openmp program.f90 -o program`

In this practical, it is recommended to use the Intel compiler as this compiler provides very good performance for Intel CPUs.

Log on to the Grace login node and download the OpenMP programs:

```
wget http://grace-head00.uea.ac.uk/grace-docs/openmp_examples.tar
tar -xvf openmp_examples.tar
```

For running the exercises, please request an interactive session by typing:

```
interactive -x
```

from the Grace login node. Then load the Intel compiler module file:

```
module load icc/intel/12.1
```

Practical 1 - Scheduling

The first practical will involve exploring how the different scheduling types work with different chunk sizes. Compile either `scheduling.c` or `scheduling.f90` and follow the steps below:

1. Parallelise the loop;
2. How many threads did the program use?
3. What is the default scheduling type and what is the chunk size?
4. For each thread, list the iterations executed;
5. Change the chunk size to 4 and list the iterations that each thread executed;
6. Change the scheduling type to `guided` with a chunk size of 3. For each thread, list the iterations executed;

Practical 2 - Loop Parallelisation

The second practical involves parallelising nested loops containing dependencies to solve the equation:

$$A_{i,j} = A_{i-1,j} + x + k$$

Compile either `dependency.c` or `dependency.f90` and complete the following:

1. Identify the dependency in the program;
2. Parallelise the correct loop;
3. Insert timing routines around the parallel and serial versions;
4. Measure speed up gained from using a) 2 threads b) 4 threads c) 6 threads d) 12 threads by setting the environment variable `OMP_NUM_THREADS`;

Practical 3 - Vector Addition

The third practical involves adding two vectors:

$$c_i = a_i + b_i$$

Work on program `vector_add.c` or `vector_add.f90` and complete the following:

1. Parallelise both the initialisation and summation loop;
2. Place timing routines around the parallel and serial summation loop;
3. Measure speed up gained from using a) 2 threads b) 4 threads c) 6 threads d) 12 threads by setting the environment variable `OMP_NUM_THREADS`;

Practical 4 - Numerical Integration

The fourth practical involves solving the bounded integral numerically:

$$\int_a^b f(x)dx$$

Work on program `integration.c` or `integration.f90` and complete the following:

1. Parallelise the main loop and using the reduction clause, identify the reduction variable;
2. Increase the variable N and notice how the approximation approaches the correct value at 8/3;

Practical 5 - Dot Product

The fifth practical involves calculating the dot product of two vectors:

$$\sum_{i=1}^{n} a_i \times b_i$$

Work on either `dot_product.c` or `dot_product.f90` and complete the following:

1. Parallelise both the initialisation and summation loops;
2. Identify the variable for reduction and use this to store the sum;

Feedback

When you have completed the workshop, please remember to complete the course questionnaire at:

http://rscs.uea.ac.uk/events/feedback

All feedback is greatly appreciated!