University of Amsterdam
SURFsara
High Performance Computing and Big Data

Dr Clemens Grelck
OpenMP and MPI
June 24, 2016

Programming Multi-Core Systems with OpenMP

1 Getting started with OpenMP

Of course, your first program should be *Hello World*, but as we are using OpenMP we want each thread to say *hello*. Write a simple program that starts a parallel section where each thread says hello and prints out his/her thread number as well as the total number of threads. You can adapt the code shown during the lecture, in particular if you are not a C expert.

Compile the code as shown during the lecture and run it with different numbers of threads on your own computer as well as on the SURFsara Lisa cluster. Do not forget to activate the compiler's OpenMP support and to switch on compiler optimizations at level -O2 (or more). Use the same compiler options throughout the entire assignment.

For running code on a cluster with batch job submission system, as the Lisa cluster, you need to explicitly set the number of threads in the code using the `omp_set_num_threads()` routine.

Directions: Report on your code and on the experiments.

2 Accelerating computations with OpenMP

Adapt the code that computes the element-wise product of two vectors as shown during the lecture. Run the compiled code for different vector lengths (starting at 100,000 elements) as well as for different numbers of repetitions of the element-wise vector product.

What speedups over sequential execution of the same code can you achieve on your own computer as well as on a single node of the SURFsara Lisa cluster?

Directions: Report on your code and on the experiments.

3 Parallelizing reduction operations with OpenMP

Extend your code developed for the previous assignment to compute the scalar product of two vectors instead of the element-wise product. The scalar product is the sum of all element-wise products. Compute the scalar product with a single loop and avoid an intermediate vector to store the element-wise vector product. Devise three parallel implementations of your scalar product code:

1. using a critical section,
2. using the reduction clause,
3. using your own reduction scheme that aims at good performance while not making use of the reduction clause.

Directions: Compare all three parallel implementations in terms of speedup over sequential execution for different vector sizes (again starting at 100,000 elements) and different numbers of repetitions as in the previous assignments.

Programming Scalable Systems with MPI

4 Overlapping communication and computation

During the lecture you learned that overlapping (costly) communication with (productive) computation is crucial in message passing programming to achieve good performance.

Exercise 1 Improve the time iteration loop of the wave equation, as shown on slide 58, such that it even better overlaps communication and computation.

Exercise 2 Devise a variant of the time iteration loop of the wave equation that exploits the split-phase communication features of MPI for improved performance.

Directions: Use pseudo code in analogy to the lecture slides, explain your solutions in detail and argue how you achieve the goal of overlapping communication and computation.

5 Collective communication

Collective communication is a form of structured communication, where instead of a dedicated sender and a dedicated receiver all MPI processes of a given communicator participate. A simple example is *broadcast* communication where a given MPI process sends a message to all other MPI processes in the communicator. Write a C+MPI function

```
int MPI_Broadcast( void *buffer,            /* INOUT : buffer address  */
                   int count,               /* IN    : buffer size */
                   MPI_Datatype datatype,   /* IN    : datatype of entry  */
                   int root,                /* IN    : root process (sender)  */
                   MPI_Comm communicator)   /* IN    : communicator  */
```

that implements broadcast communication by means of point-to-point communication. Each MPI process of the given communicator is assumed to execute the broadcast function with the same message parameters and root process argument. The root process is supposed to send the content of its own buffer to all other processes of the communicator. Any non-root process uses the given buffer for receiving the corresponding data from the root process.

Exercise 1 Implement the above function `MPI_Broadcast` using point-to-point communication.

Exercise 2 One advantage of collective communication is that the implementation can take advantage of the underlying physical network topology without making the application itself platform-specific. With growing node counts a fully connected network topology is technically infeasible.

Re-implement your `MPI_Broadcast` function such that it takes advantage of a 1-dimensional ring topology, where each node only has direct communication links with its two immediate neighbours with (circularly) increasing and decreasing MPI ranks. Message transfers between non-neighbouring nodes require multiple redirections via intermediate nodes; communication cost can be assumed to be linear in the number of links involved.

Directions: Write real C+MPI code for this assignment, not pseudo code. No experiments are required, but describe your code in detail: how it is adapted to the given network topology and how many atomic communication events (sending a message from one node to a neighbouring node) are required to complete one broadcast.

Submission deadline: July 17, 2016