University of Amsterdam
SURFsara
High Performance Computing and Big Data

Dr Clemens Grelck
OpenMP and MPI
Feb 1, 2018

Programming Multi-Core Systems with OpenMP
Programming Scalable Systems with MPI

1 Experimenting with OpenMP

Consider the C function `multimult` below. Design a suitable `main` function that properly allocates and de-allocates memory for the two vectors and measures the time spent in a single call to `multimult` using a high precision timer (e.g. `clock_gettime` on Linux). Parallelise the `multimult` function using OpenMP pragmas. Do **not** make use of any default rules, but make all clauses explicit.

```
void multimult( double *a, double *b, int len, int steps)
{
  double c;

  for (int t=0; t<steps; t++) {
    for (int i=0; i<len; i++) {
      c = a[i] * b[i];
      a[i] = c * (double) t;
    }
  }
}
```

Run experiments with different vector lengths, different numbers of steps and different numbers of threads up to the number of cores available. Use both your own machine and a single node of the SURFsara cluster. Systematically increase the values for `len` and `steps`. Write a report that shows your code, motivates the choice of pragmas and details the outcome of the experiments with speedup graphs for the different settings. Only report on *interesting* experimental parameters.

For running code on a cluster with batch job submission system, it is best to explicitly set the number of threads in the code using the `omp_set_num_threads()` routine. Do not forget to activate the compiler's OpenMP support and to switch on compiler optimisations at level -O2 (or higher).

2 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 one MPI process sends a message to all other MPI processes in the communicator. Write a C+MPI function (not pseudo code)

```
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.

3 Collective communication with topology adaptation

One advantage of collective communication is that the implementation can take advantage of the underlying physical network topology without making the application itself topology-specific. Rewrite your `MPI_Broadcast` function such that it takes advantage of the underlying network topology.

With growing node counts a fully connected network topology is technically infeasible. Instead, assume 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.

Describe in detail how your code is adapted to the network topology. Devise a closed formula, depending on the number of nodes, that describes the number of atomic communication events (sending a message from one node to a neighbouring node) needed to complete one broadcast operation.

Deadline:
Individual submission by February 15, 2018, any time