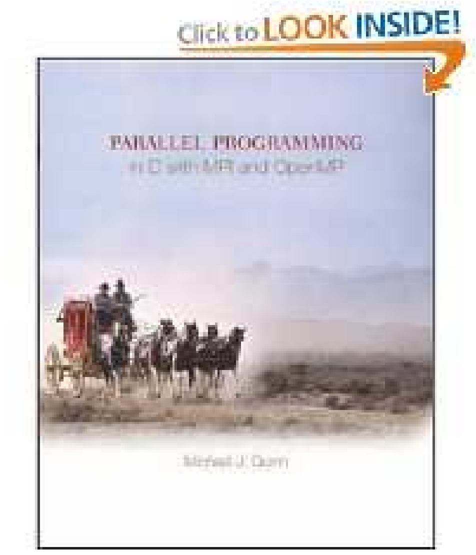
Mixed MPI-OpenMP programming

Overview

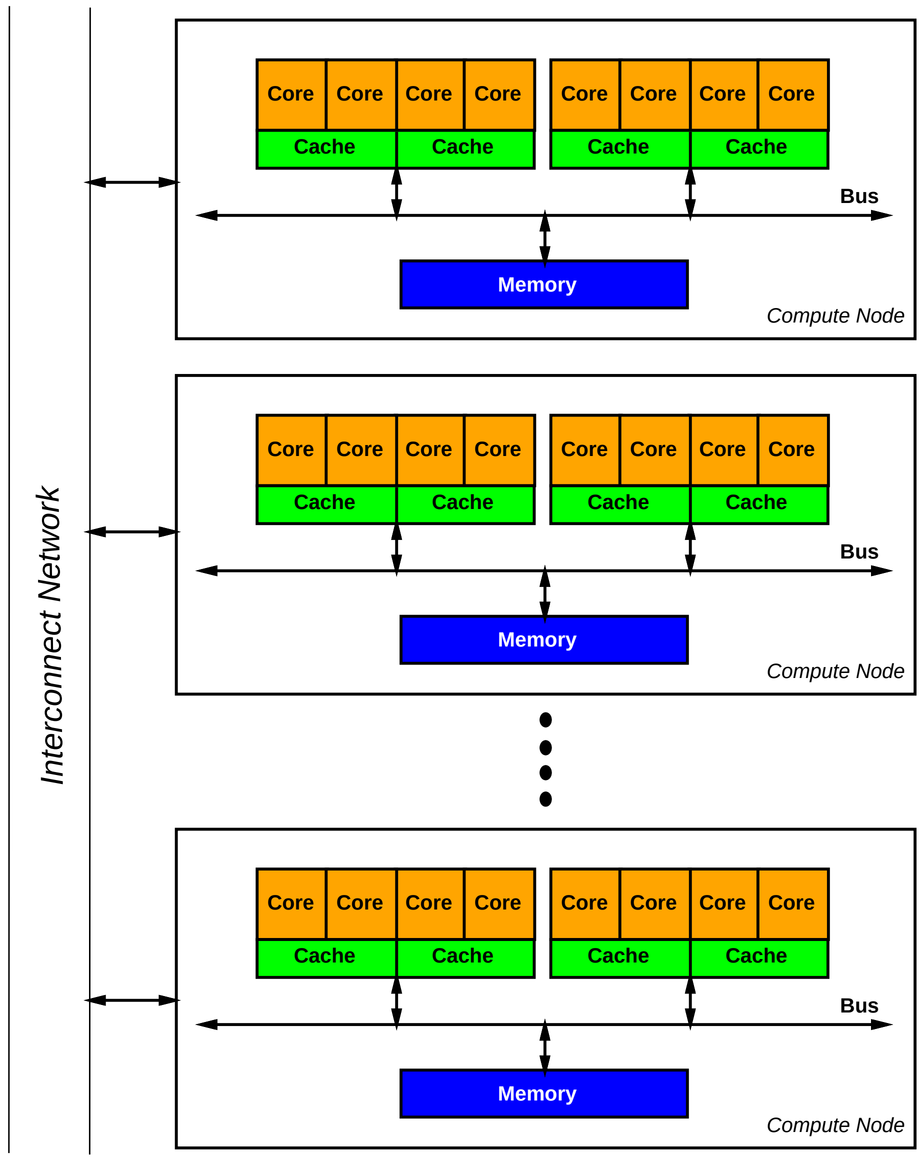
- Motivations for mixed MPI-OpenMP programming
- Advantages and disadvantages
- The example of the Jacobi method
- Chapter 18 in *Michael J. Quinn*, **Parallel Programming in C with MPI and OpenMP**



Motivation from hardware architecture

- There exist distributed shared-memory parallel computers
  - High-end clusters of SMP machines
  - Low-end clusters of multicore-based compute nodes
- MPI is the de-facto standard for communication between the SMPs/nodes
- Within each SMP/node
  - MPI can be used for intra-node communication, but may not be aware of the shared memory
  - Thread-based programming directly utilizes the shared memory
  - OpenMP is the easiest choice of thread-based programming

Multicore-based cluster



Motivation from communication overhead

- Assume a cluster that has m nodes, each node has k CPUs
- If MPI is used over the entire cluster, we have mk MPI processes
  - Suppose each MPI process on average sends and receives 4 messages
  - Total number of messages: $4mk$
- If MPI is used only for inter-node parallelism, while OpenMP threads control intra-node parallelism
  - Number of MPI processes: m
  - Total number of messages: $4m$
- Therefore, fewer MPI messages in the mixed MPI-OpenMP approach
  - Less probability for network contention
  - But the messages are larger
  - Total message-passing overhead is smaller

Motivation from amount of parallelism

- Assume a sequential code: 5% purely serial work, 90% perfectly parallelizable work, and 5% work difficult to parallelize
- Suppose we have a 8-node cluster, each node has two CPUs
- If MPI is used over the entire cluster, i.e., 16 MPI processes
  - Speedup:

$$\frac{1}{0.05 + 0.90/16 + 0.05} = 6.4$$

  - Note that the 5% non-easily parallelizable work is duplicated on all the 16 MPI processes
- If mixed MPI-OpenMP programming is used
  - Speedup:

$$\frac{1}{0.05 + 0.90/16 + 0.05/2} = 7.6$$

  - Note that the 5% non-easily parallelizable work is duplicated on the 8 MPI processes, but within each MPI process it is parallelized by the two OpenMP threads

Motivation from granularity and load balance

- Larger grain size (more computation) for fewer MPI processes
  - Better computation/communication ratio
- In general, better load balance for fewer MPI processes
  - In the pure MPI approach, due to the large number of MPI processes, there is a higher probability for some of the MPI processes being idle
  - In the mixed MPI-OpenMP approach, the MPI processes have a lower probability of being idle

Advantages

Mixed MPI-OpenMP programming

- can avoid intra-node MPI communicaiton overheads
- can reduce the possibility of network contention
- can reduce the need for replicated data
  - data is guaranteed to be shared inside each node
- may improve a poorly scaling MPI code
  - load balance can be difficult for a large number of MPI processes
  - for example, 1D decomposiiton by the MPI processes may replace 2D decomposition
- may adopt dynamic load balancing within one node

Disadvantages

Mixed MPI-OpenMP programming

- may introduce additional overhead not present in the MPI code
  - thread creation, false sharing, sequential sections
- may adopt more expensive OpenMP barriers than implicit point-to-point MPI synchronizations
- may be difficult to overlap inter-node communication with computation
- may have more cache misses during point-to-point MPI communication
  - the messages are larger
  - cache is not shared among all threads inside one node
- may not be able to use all the network bandwidth by one MPI process per node

Inter-node communication

There are 4 different styles of handling inter-node communication

- “Single”
  - all MPI communicaiton is done by the OpenMP master thread,
  - outside the parallel regions
- “Funnelled”
  - all MPI communicaiton is done by the master thread inside a parallel region
  - other threads may be doing computations
- “Serialized”
  - More than one thread per node carry out MPI communicaitons
  - but one thread at a time
- “Multiple”
  - More than one thread per node carry out MPI communicaitons
  - can happen simultaneously

Simple example of hello-world

```
#include <mpi.h>
#include <omp.h>
#include <stdio.h>

int main (int nargs, char** args)
{
  int rank, nprocs, thread_id, nthreads;

  MPI_Init (&nargs, &args);
  MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);

#pragma omp parallel private(thread_id, nthreads)
  {
    thread_id = omp_get_thread_num ();
    nthreads = omp_get_num_threads ();
    printf("I'm thread nr.%d (out of %d) on MPI process nr.%d (out of %d)\
           thread_id, nthreads, rank, nprocs);
  }

  MPI_Finalize ();

  return 0;
}
```

Example of the Jacobi method (1)

We want to solve a 2D Laplace equation:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0,$$

where u is known on the boundary.

Assume the solution domain is the unit square, and we use finite differences on a uniform mesh $\Delta x = \Delta y = h = \frac{1}{N-1}$:

$$\frac{u_{i-1,j} + u_{i,j-1} - 4u_{i,j} + u_{i,j+1} + u_{i+1,j}}{h^2} = 0$$

for $i = 1, 2, \ldots, N-2$ and $j = 1, 2, \ldots, N-2$

Example of the Jacobi method (2)

Let us use the Jacobi method to find $u_{i,j}$.

The Jacobi method is an iterative process, which starts with an initial guess $u^0_{i,j}$, and generates $u^1_{i,j}$, $u^2_{i,j}$, $\ldots$.

We stop the iterations when $u^k_{i,j} - u^{k-1}_{i,j}$ is small enough for all i, j.

Example of the Jacobi method (3)

Formula for calculating $u_{i,j}^k$ from u^{k-1} on all the interior points:

$$u_{i,j}^k = \frac{1}{4}\left(u_{i-1,j}^{k-1} + u_{i,j-1}^{k-1} + u_{i,j+1}^{k-1} + u_{i+1,j}^{k-1}\right)$$

Example of the Jacobi method (4)

- A serial C code uses 2D arrays w and u
- w contains u^k, while u contains u^{k-1}

```
for (;;) {
  tdiff = 0.0;

  for (i=1; i<N-1; i++)
    for (j=1; j<N-1; j++) {
      w[i][j] = (u[i-1][j]+u[i+1][j]+u[i][j-1]+u[i][j+1])/4.0;
      if (fabs(w[i][j] - u[i][j]) > tdiff)
        tdiff = fabs(w[i][j] - u[i][j]);
    }

  if (tdiff <= EPSILON) break;

  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      u[i][j] = w[i][j];
}
```

Example of the Jacobi method (5)

- The MPI code divides the i rows into blocks
- Each subdomain needs one ghost layer on top and one ghost layer on bottom
- MPI process `id` needs to exchage with processes `id-1` and `id+1` by using `MPI_Send` and `MPI_Recv`
- In addition, `MPI_Allreduce` is needed to find the maximum `tdiff` among all MPI processes

Example of the Jacobi method (6)

Mixed MPI-OpenMP implementation introduces a parallel region

```
int find_steady_state (int p, int id, int my_rows,
                       double **u, double **w)
{
   double     diff;              /* Maximum difference on this process */
   double     global_diff;       /* Globally maximum difference */
   int        i, j;
   int        its;               /* Iteration count */
   MPI_Status status;            /* Result of receive */
   double     tdiff;             /* Maximum difference on this thread */

   its = 0;
   for (;;) {

      /* Exchange rows for ghost buffers */
      if (id > 0)
         MPI_Send (u[1], N, MPI_DOUBLE, id-1, 0, MPI_COMM_WORLD);
      if (id < p-1) {
         MPI_Send (u[my_rows-2], N, MPI_DOUBLE, id+1, 0, MPI_COMM_WORLD);
         MPI_Recv (u[my_rows-1], N, MPI_DOUBLE, id+1, 0, MPI_COMM_WORLD,
                   &status);
      }
      if (id > 0)
         MPI_Recv (u[0], N, MPI_DOUBLE, id-1, 0, MPI_COMM_WORLD, &status);
```

Example of the Jacobi method (7)

```
      /* Update the new approximation */

      diff = 0.0;
#pragma omp parallel private (i, j, tdiff)
{
      tdiff = 0.0;
#pragma omp for
      for (i = 1; i < my_rows-1; i++)
         for (j = 1; j < N-1; j++) {
            w[i][j] = (u[i-1][j] + u[i+1][j] +
                       u[i][j-1] + u[i][j+1])/4.0;
            if (fabs(w[i][j] - u[i][j]) > tdiff)
               tdiff = fabs(w[i][j] - u[i][j]);
         }
```

Example of the Jacobi method (8)

```
#pragma omp for nowait
      for (i = 1; i < my_rows-1; i++)
         for (j = 0; j < N; j++)
            u[i][j] = w[i][j];
#pragma omp critical
      if (tdiff > diff) diff = tdiff;
}  /* end of parallel region */

      MPI_Allreduce (&diff, &global_diff, 1, MPI_DOUBLE, MPI_MAX,
                     MPI_COMM_WORLD);

      /* Terminate if the solution has converged */
      if (global_diff <= EPSILON) break;

      its++;
   }
   return its;
}
```

When to use mixed MPI-OpenMP programming?

- Rule-of-the-thumb: pure OpenMP must scale better than pure MPI within one node, otherwise no hope for mixed programming
- Whether mixed MPI-OpenMP programming is in fact more advantagenous is problem dependent