Programming with OpenMP and mixed MPI-OpenMP

Xing Cai

Simula Research Laboratory & University of Oslo

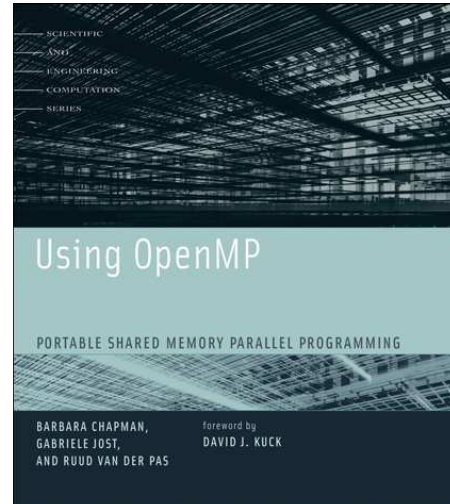
http://heim.ifi.uio.no/~xingca/openmp-lecture.pdf

What will we learn today?

- The most important ingredients of OpenMP programming
- Simple coding examples (in C)
- Mixed MPI-OpenMP programming

Resources

- B. Chapman, G. Jost, R. van der Pas. *Using OpenMP*. MIT Press, 2007



- B. Barney. *OpenMP tutorial*
  `http://www.llnl.gov/computing/tutorials/openMP/`
- OpenMP official web site
  `http://openmp.org/`

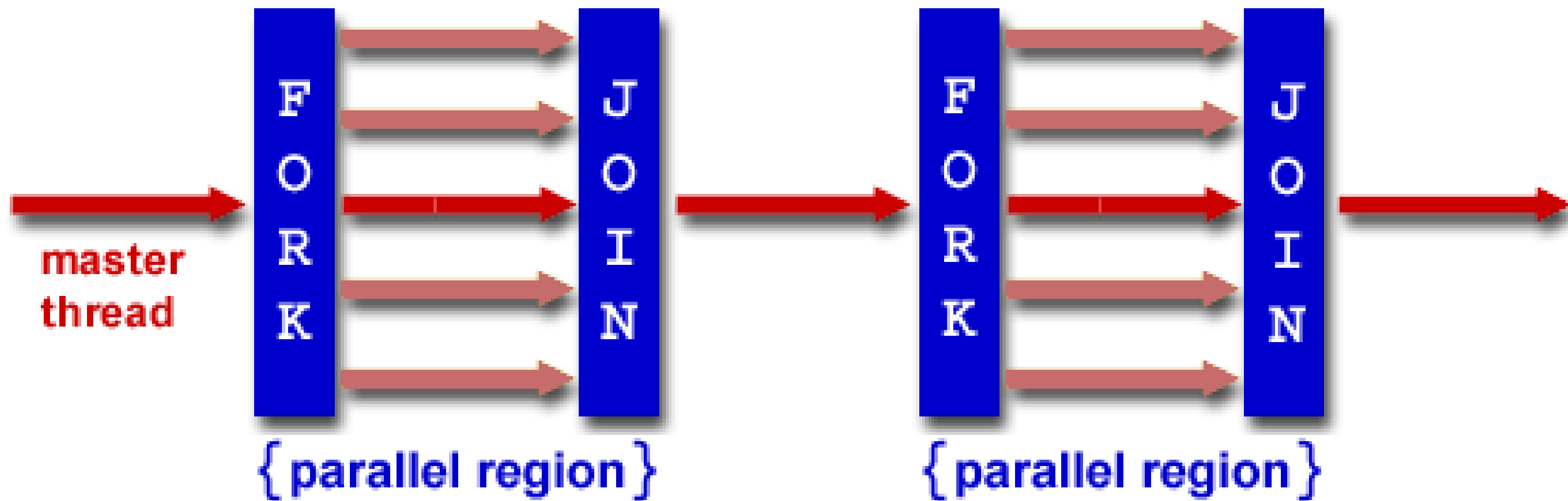
What is OpenMP?

- OpenMP — a portable standard for shared-memory programming
- The OpenMP API consists of
  - compiler directives (for insertion into sequential Fortran/C/C++ code)
  - a few library routines
  - some environment variables
- Advantages:
  - User-friendly
  - Incremental parallelization of a serial code
  - Possible to have a single source code for both serial and parallelized versions
- Disadvantages:
  - Relatively limited user control
  - Most suitable for parallelizing loops (data parallelism)
  - Performance?

The programming model of OpenMP

- OpenMP provides high-level thread programming
- Multiple cooperating threads are allowed to run simultaneously
- Threads are created and destroyed dynamically in a **fork-join** pattern
  - An OpenMP program consists of a number of parallel regions
  - Between two parallel regions there is only one master thread
  - In the beginning of a parallel region, a team of new threads is spawned
  - The newly spawned threads work simultaneously with the master thread
  - At the end of a parallel region, the new threads are destroyed

Fork-join model



https://computing.llnl.gov/tutorials/openMP/

OpenMP: first things first

- Remember the header file `#include <omp.h>`
- Insert compiler directives (`#pragma omp...` in C/C++ syntax), possibly also some OpenMP library routines
- Compile
  - For example, `gcc -fopenmp code.c`
- Execute
  - Remember to assign the environment variable `OMP_NUM_THREADS`
  - It specifies the total number of threads inside a parallel region, if not otherwise overwritten

General code structure

```
#include <omp.h>

main ()  {

  int var1, var2, var3;

  /* serial code */
  /* ... */

  /* start of a parallel region */
#pragma omp parallel private(var1, var2) shared(var3)
  {
    /* ... */
  }

  /* more serial code */
  /* ... */

  /* another parallel region */
#pragma omp parallel
  {
    /* ... */
  }
}
```

Parallel region

- A parallel region is a block of code that is executed by a team of threads
- The following compiler directive creates a parallel region
  `#pragma omp parallel { ... }`
- Clauses can be added at the end of the directive
- Most often used clauses:
  - `default(shared)` or `default(none)`
  - `public(list_of_variables)`
  - `private(list_of_variables)`

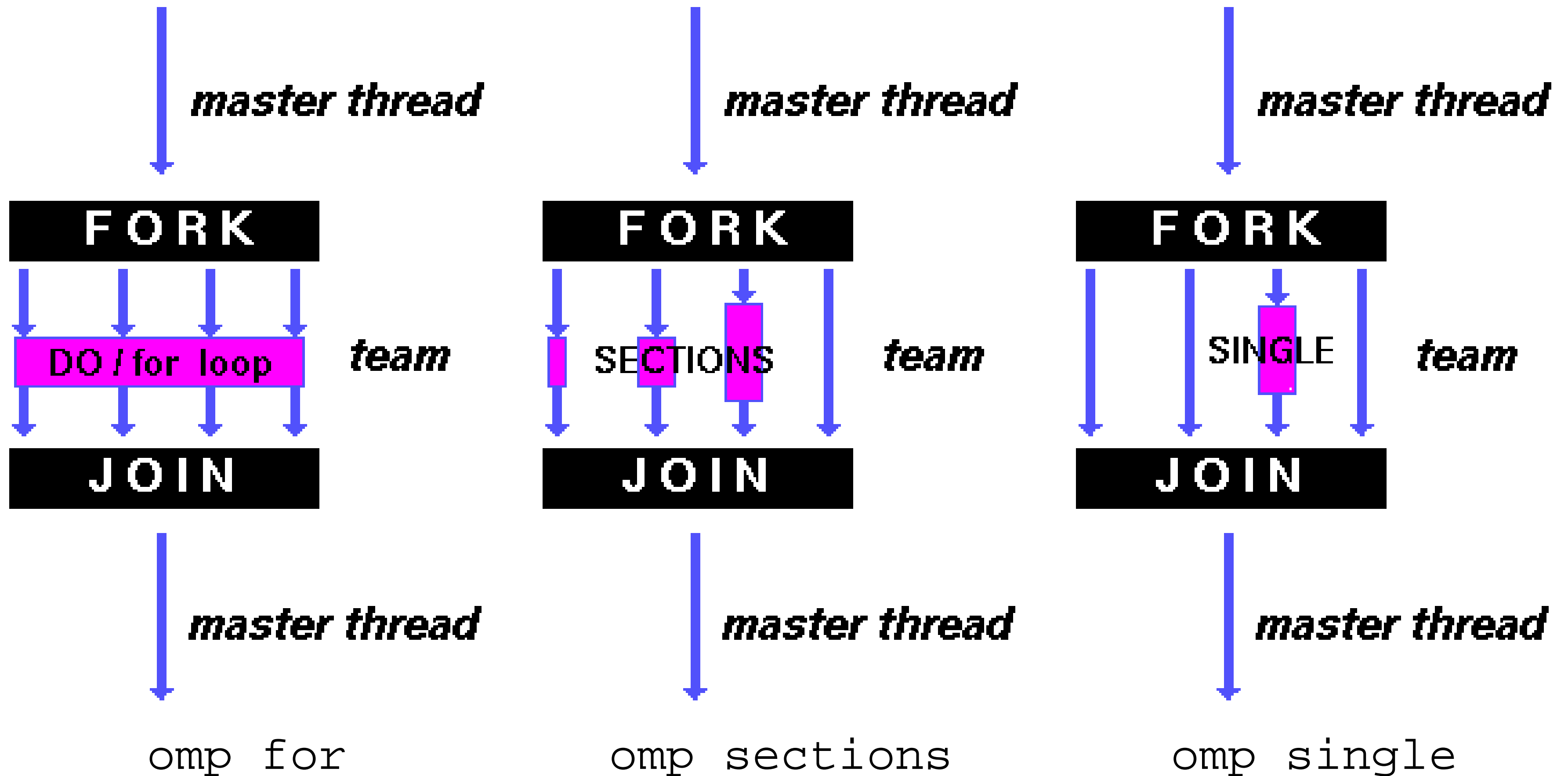
Hello-world in OpenMP

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

int main (int argc, char *argv[])
{
  int th_id, nthreads;

  #pragma omp parallel private(th_id) shared(nthreads)
  {
    th_id = omp_get_thread_num();
    printf("Hello World from thread %d\n", th_id);

    #pragma omp barrier

    if ( th_id == 0 ) {
      nthreads = omp_get_num_threads();
      printf("There are %d threads\n",nthreads);
    }
  }

  return 0;
}
```

Important OpenMP library routines

- `int omp_get_num_threads ()`
  returns the number of threads inside a parallel region
- `int omp_get_thread_num ()`
  returns the "thread id" for each thread inside a parallel region
- `void omp_set_num_threads (int)`
  sets the number of threads to be used
- `void omp_set_nested (int)`
  turns nested parallelism on/off

Work-sharing constructs



https://computing.llnl.gov/tutorials/openMP/

Parallel for loop

- Inside a parallel region, the following compiler directive can be used to parallelize a `for`-loop:
  `#pragma omp for`
- Clauses can be added, such as
  - `schedule(static, chunk_size)`
  - `schedule(dynamic, chunk_size)` (non-deterministic allocation)
  - `schedule(guided, chunk_size)` (non-deterministic allocation)
  - `schedule(runtime)`
  - `private(list_of_variables)`
  - `reduction(operator:variable)`
  - `nowait`

Example

```
#include <omp.h>
#define CHUNKSIZE 100
#define N     1000

main ()
{
  int i, chunk;
  float a[N], b[N], c[N];

  for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
  chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
  {
  #pragma omp for schedule(dynamic,chunk)
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];
  }  /* end of parallel region */
}
```

More about parallel for

- The number of loop iterations can not be non-deterministic
  - `break`, `return`, `exit`, `goto` not allowed inside the `for`-loop
- The loop index is private to each thread
- A reduction variable is special
  - During the `for`-loop there is a local private copy in each thread
  - At the end of the `for`-loop, all the local copies are combined together by the reduction operation
- Unless the `nowait` clause is used, an implicit barrier synchronization will be added at the end by the compiler
- `#pragma omp parallel` and `#pragma omp for` can be combined into
  `#pragma omp parallel for`

Example of computing inner-product

$$\sum_{i=0}^{N-1} a_i \, b_i$$

```
  int i;
  double sum = 0.;

  /* allocating and initializing arrays 'a' 'b' */
  /* ... */

#pragma omp parallel for default(shared) private(i) reduction(+:sum)
  for (i=0; i<N; i++)
    sum += a[i]*b[i];
}
```

Parallel sections

Different threads do different tasks independently, each section is executed by one thread.

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
      funcA ();

    #pragma omp section
       funcB ();

    #pragma omp section
       funcC ();
   }
}
```

Single execution

- `#pragma omp single { ... }`
  - code executed by one thread only, no guarantee which thread
  - an implicit barrier at the end
- `#pragma omp master { ... }`
  - code executed by the master thread, guaranteed
  - no implicit barrier at the end

Coordination and synchronization

- `#pragma omp barrier`
  - synchronization, must be encountered by all threads in a team (or none)
- `#pragma omp ordered { a block of codes }`
  - another form of synchronization (in sequential order)
- `#pragma omp critical { a block of codes }`
- `#pragma omp atomic { single assignment statement }`
  - more efficient than `#pragma omp critical`

Data scope

- OpenMP data scope attribute clauses:
  - `shared`
  - `private`
  - `firstprivate`
  - `lastprivate`
  - `reduction`
- Purposes:
  - define how and which variables are transferred to a parallel region (and back)
  - define which variables are visible to all threads in a parallel region, and which variables are privately allocated to each thread

Some remarks

- When entering a parallel region, the `private` clause ensures each thread having its own new variable instances. The new variables are assumed to be uninitialized.
- A shared variable exists in only one memory location and all threads can read and write to that address. It's the programmer's responsibility to ensure that multiple threads properly access a shared variable.
- The `firstprivate` clause combines the behavior of the `private` clause with automatic initialization.
- The `lastprivate` clause combines the behavior of the `private` clause with a copy back (from the last loop iteration or section) to the original variable outside the parallel region.

Parallelizing nested for-loops

- Serial code

```
for (i=0; i<100; i++)
  for (j=0; j<100; j++)
    a[i][j] = b[i][j] + c[i][j]
```

- Parallelization

```
#pragma omp parallel for private(j)
for (i=0; i<100; i++)
  for (j=0; j<100; j++)
    a[i][j] = b[i][j] + c[i][j]
```

- Why not parallelize the inner loop?
  - to save overhead of repeated thread forks-joins
- Why must `j` be `private`?
  - to avoid race condition among the threads
- Comments
  - OpenMP 2.5 allows parallelization of only one loop layer
  - OpenMP 3.0 has a new `collapse` clause

Nested parallelism

When a thread in a parallel region encounters another parallel construct, it may create a new team of threads and become the master of the new team.

```
#pragma omp parallel num_threads(4)
{
  /* .... */

  #pragma omp parallel num_threads(2)
  {
    /* .... */
  }
}
```

Parallel tasks

`#pragma omp task` (starting with OpenMP 3.0)

```
#pragma omp parallel shared(p_vec) private(i)
{
#pragma omp single
  {
    for (i=0; i<N; i++) {
      double r = random_number();
      if (p_vec[i] > r) {
#pragma omp task
         do_work (p_vec[i]);
      }
    }
  }
}
```

Common mistakes

- Race condition

```
int nthreads;
#pragma omp parallel shared(nthreads)
{
  nthreads = omp_get_num_threads();
}
```

- Deadlock

```
#pragma omp parallel
{
  ...
#pragma omp critical
  {
    ...
#pragma omp barrier
  }
}
```

How about performance?

Factors that influence the performance of OpenMP programs:

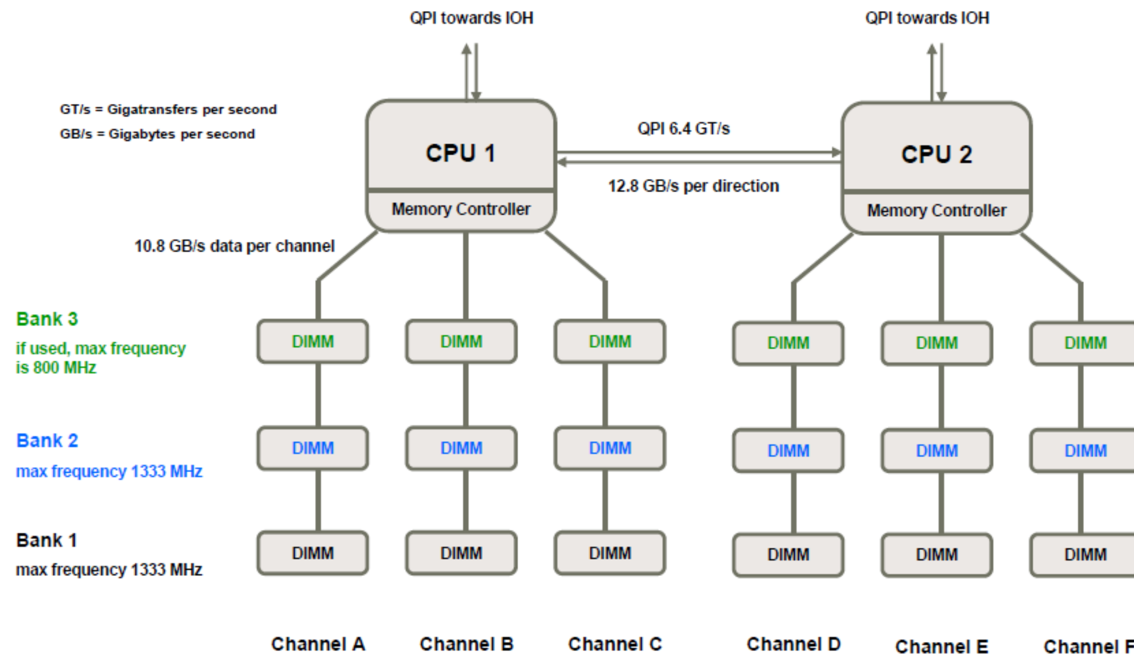
- How the memory is accessed by individual threads
- The fraction of work that is sequential (or replicated)
- The overhead of handling OpenMP constructs
- Load imbalance
- Synchronization costs

Good programming practices:

- Optimize use of `barrier`
- Avoid `ordered` construct
- Avoid large `critical` blocks
- Maximize parallel regions
- Avoid parallel regions in inner loops
- Use `schedule(dynamic)` or `schedule(guided)` to address poor load balance

The issue of NUMA

- Non-uniform memory access (e.g., dual-socket quad-core Nehalem)



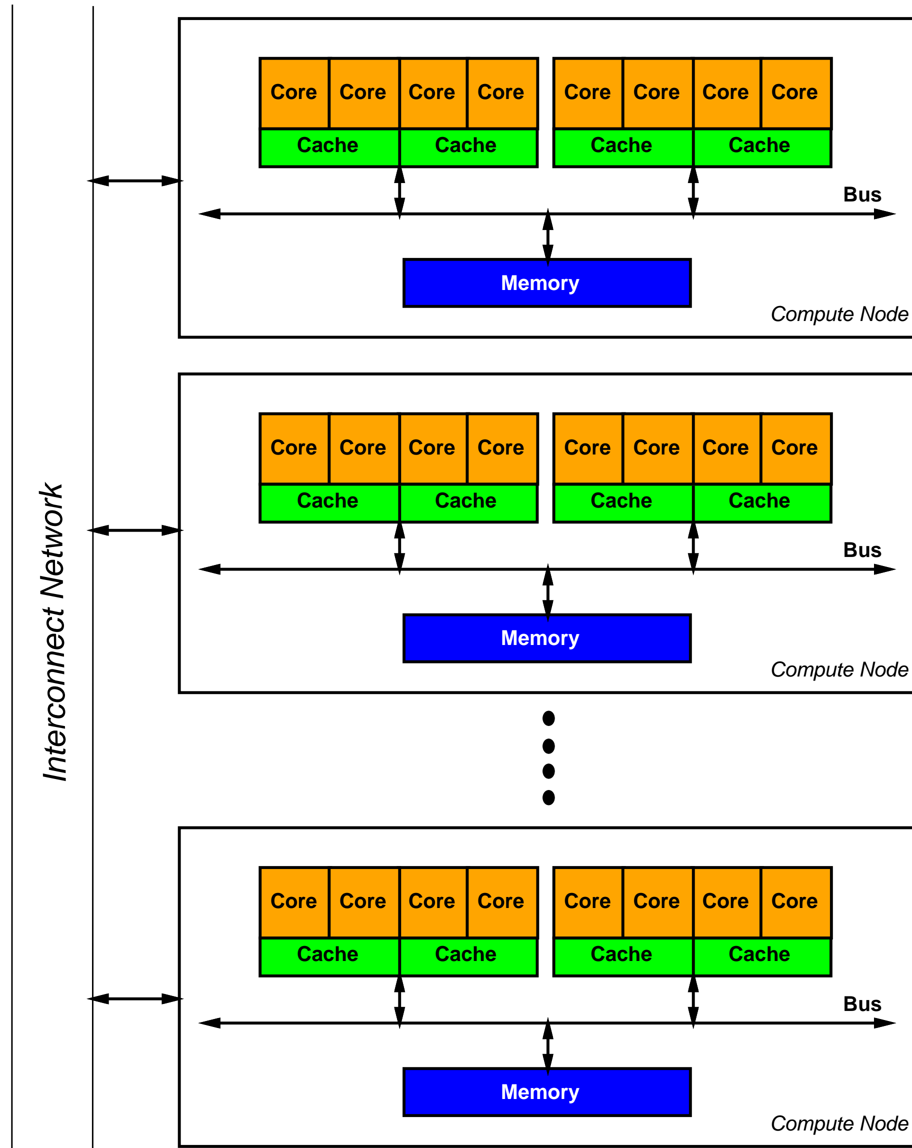
- Each thread should, if possible, only work with data close-by
  - Use of first touch in data initialization
  - Use of static scheduler with fixed chunk size
- Avoid false sharing on ccNUMA architecture

Mixed MPI-OpenMP programming

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 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 communication 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 decomposition 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 communication is done by the OpenMP master thread,
  - outside the parallel regions
- “Funnelled”
  - all MPI communication 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 communications
  - but one thread at a time
- “Multiple”
  - More than one thread per node carry out MPI communications
  - 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;
}
```

When to use mixed MPI-OpenMP programming?

- Poor scaling with MPI implementation (e.g. due to load imbalance or too fine granularity)
- Memory limitation associated with replicated data for MPI implementation
- Rule-of-the-thumb: performance of pure OpenMP implementation must be comparable with pure MPI implementation within one node