Adding MPI to OpenMP

Hybrid programming: MPI + X
MPI vs. OpenMP

• Pure MPI Pro:
  • Portable to distributed and shared memory machines
  • Scales beyond one node
  • No data placement problem
  • Explicit communication

• Pure MPI Con:
  • Difficult to develop and debug
  • High latency, low bandwidth (max PCI-x bus)
  • Large granularity
  • Difficult load balancing

• Pure OpenMP Pro:
  • Easy to implement parallelism
  • Low latency, high bandwidth (max memory bus)
  • Implicit Communication
  • Coarse and fine granularity
  • Dynamic load balancing

• Pure OpenMP Con:
  • Difficult to develop and debug
  • Only on shared memory machines
  • Scale within one node
  • Possible data placement problem (on NUMA architectures)
  • No specific thread order
Why hybrid programming?

- Hybrid MPI+X paradigm is the software trend for dealing with complexities of hybrid hierarchical architectures (such as heterogeneous multi-core architectures prevalent nowadays).

- Elegant in concept and architecture: using MPI across nodes and OpenMP within nodes. Good usage of shared memory system resource (memory, latency, and bandwidth).

- Avoids the extra communication overhead with MPI within node. Reduce memory footprint.

- OpenMP adds fine granularity (larger message sizes) and allows increased and/or dynamic load balancing.

- Some problems have two-level parallelism naturally.

- Some problems could only use restricted number of MPI tasks.

- Possible better scalability than both pure MPI and pure OpenMP.
Example 1

```c
int main(int argc, char* argv[]) {
    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    #pragma omp parallel private(omp_rank)
    {
        omp_rank = omp_get_thread_num();
        printf("Rank %d thread %d\n", rank, omp_rank);
    }
    MPI_Finalize();
}
```

• What is the expected outcome?
Example 1

```c
int main(int argc, char* argv[]) {
    MPI_Init(NULL, NULL);
    #pragma omp parallel private(omp_rank)
    {
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        omp_rank = omp_get_thread_num();
        printf("Rank %d thread %d\n", rank, omp_rank);
    }
    MPI_Finalize();
}
```

- What is the expected outcome?
Initializing MPI with thread support

• **MPI_INIT_THREAD** (required, &provided, ierr)
  • IN: required, desired level of thread support (integer).
  • OUT: provided, provided level of thread support (integer).
  • Beware: Returned provided maybe less than required.

• Thread support levels:
  • **MPI_THREAD_SINGLE**: Only one thread will execute.
  • **MPI_THREAD_FUNNELED**: Process may be multi-threaded, but only master thread will make MPI calls (all MPI calls are "funneled" to master thread)
  • **MPI_THREAD_SERIALIZED**: Process may be multi-threaded, multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are "serialized").
  • **MPI_THREAD_MULTIPLE**: Multiple threads may call MPI, with no restrictions.

**MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED < MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE**
OMP MASTER calls MPI

• The OMP master thread is the thread that entered main
  • In some OSes it might have specific properties and behaviors (signals, pid, ...)
• MPI_THREAD_FUNNELED is required
• Inside a parallel region there are no implicit synchronizations

```c
#pragma omp parallel
for(i = 0; i < BIG_NUMBER; i++)
  buf[i] = i;

#pragma omp master
MPI_Send(buf, ...);
```
OMP MASTER calls MPI

• The OMP master thread is the thread that entered main
  • In some OSes it might have specific properties and behaviors (signals, pid, ...)
• MPI_THREAD_FUNNELED is required
• Inside a parallel region there are no implicit synchronizations
  • An explicit barrier before the MPI call is needed to ensure correctness of the input data
  • An explicit barrier after the MPI call is needed to ensure correctness of the output data
  • It also implies that all the other threads are wasting time

```c
#pragma omp parallel
for(i = 0; i < BIG_NUMBER; i++)
  buf[i] = 1;

#pragma omp master
MPI_Send(buf, ...);
```
OMP SINGLE calls MPI

• The OMP single directive ensure the only one thread executes the corresponding block
• MPI_THREAD_SERIALIZED is required
• Inside a parallel region there are no implicit synchronizations
  • An explicit barrier before the MPI call is needed to ensure correctness of the input data
  • An explicit barrier after the MPI call is needed to ensure correctness of the output data
  • It also implies that all the other threads are wasting time

```c
#pragma omp parallel
for(i = 0; i < BIG_NUMBER; i++)
  buf[i] = i;

#pragma omp master
  MPI_Send(buf, ...);
```
No pain, no gain

• Enforcing barriers limit the performance
• Removing the barriers depends on the algorithm and on the other implicit synchronizations between parts of the algorithm
  • When was the data updated? Outside the parallel section?
  • When will be the data used? Outside this parallel section?
• Without the barrier automatic overlap between computations and communications become automatic
A word (or two) about affinity

- Single threaded MPI applications rarely raise affinity issues
- Unleashing multiple threads in the context of the same application is a different topic:
  - Thread affinity: floating vs. bound
    - Memory issues
  - Memory affinity: allocate memory as close as possible to the core that will use it most
    - Affinity is not decided during the allocation
    - The default policy is "first touch"
- Each MPI library has its own affinity settings (read the man/documentation...)

[Diagram of CPU and I/O controllers connected with memory storage]
More words about affinity

- Performance with and without correct data initialization
- **HWLOC** is the tool to use!

```c
#pragma omp parallel for
for( i = 0; i < MANY; i++ ) {
    a[i] = 1.0; b[i] = 2.0; c[i] = 0
}
```

```c
#pragma omp parallel for
For( i = 0; i < MANY; i++ ) {
    c[i] = a[i] * b[i];
}
```

Stream NUMA effects - Hopper

[Graph showing bandwidth over number of OpenMP threads]

Courtesy Hongzhang Shan
Hybrid Parallelization steps

• From sequential code, decompose with MPI first, then add OpenMP
• From OpenMP code, treat as serial code.
• From MPI code, add OpenMP.
• Simplest and least error-prone way is to use MPI outside parallel region, and allow only master thread to communicate between MPI tasks. MPI_THREAD_FUNNELED is usually the best choice.
  • Keep in mind the cost and implications of serializations
• Could use MPI inside parallel region with thread-safe MPI.
• MPI_THREAD_MULTIPLE comes with a performance cost. Inside the MPI library, thread synchronizations might be necessary, and this might show on the overheads of the MPI calls.