MPI + X programming

George Bosilca
CS462
MPI

Each programming paradigm only covers a particular spectrum of the hardware capabilities
- MPI is about moving data between distributed memory machines
- CUDA is about accessing the sheer computations power of a single GPU
- OpenMP is about taking advantage of the multicores architectures

What is involved in moving data between 2 machines
- Bus (PCI/PCI-X)
- Memory (pageable, pinned, virtual)
- OS (security)

Applications need to fully take advantage of all available hardware capabilities. It became imperative to combine different programming paradigms together!
PCI* performance

PCI - Peripheral Component Interconnect
PCI-X - Peripheral Component Interconnect eXtended
PCIe - Peripheral Component Interconnect Express

- split transactions (transactions with request and response separated by time)
- has a protocol and processing overhead due to the additional transfer robustness (line code below)
  - CRC and acknowledgements

---

### PCI Express link performance

<table>
<thead>
<tr>
<th>PCI Express version</th>
<th>Line code</th>
<th>Transfer rate[^i]</th>
<th>Throughput[^i]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>x1</td>
<td>x2</td>
</tr>
<tr>
<td>1.0</td>
<td>8b/10b</td>
<td>2.5 GT/s</td>
<td>250 MB/s</td>
</tr>
<tr>
<td>2.0</td>
<td>8b/10b</td>
<td>5.0 GT/s</td>
<td>500 MB/s</td>
</tr>
<tr>
<td>3.0</td>
<td>128b/130b</td>
<td>8.0 GT/s</td>
<td>984.6 MB/s</td>
</tr>
<tr>
<td>4.0</td>
<td>128b/130b</td>
<td>16.0 GT/s</td>
<td>1969 MB/s</td>
</tr>
<tr>
<td>5.0[^30][^31] (expected in Q2 2019)[^33]</td>
<td>128b/130b</td>
<td>32.0 or 25.0 GT/s[^ii]</td>
<td>3938 or 3077 MB/s</td>
</tr>
</tbody>
</table>

[^i]: In each direction (each lane is a dual simplex channel).
[^ii]: Both rates are being considered for technical feasibility.
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] = i;
#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, ...);
```
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 barrier
    #pragma omp single
    MPI_Send(buf, ...);
    #pragma omp barrier
}
```
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 it’s own affinity settings (read the man/documentation...)

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];
}
```

Courtesy Hongzhang Shan
Adding MPI to MPI

Hybrid programming: MPI + X
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.
• Special care should be taken regarding collective communications (where clearly only one thread per node should call the collective)
  • Multiple collective calls of the same type in the same communicator is explicitly prohibited by the MPI standard
MPI + MPI (2-level hybridization)

• MPI point-to-point used to exchange data between nodes, MPI-3.0 shared memory regions used inside the node to share content
  • Advantages
    • Lower communication overheads: No message passing inside of the SMP nodes
    • Simplicity: only one parallel programming standard
    • No thread-related data races (e.g., thread-safety isn’t an issue)
  • Problems
    • Application responsibility to split communicators into shared memory islands
    • To minimize shared memory communication overhead: the data accessed by the neighbors must be stored in MPI shared memory windows (memory regions visible to other processes where explicit synchronizations are necessary)
    • Load-balancing is as complicated as in pure MPI
Adding MPI to CUDA

Hybrid programming: MPI + X
• The CPU is the main driver, it launches kernels on the GPU that perform computations:
  \[ \text{sum}^{<<<1,1>>>}(2, 3, \text{device}_z) \]
  
  • Data must be moved between main memory and GPU prior to the computations.
  • And must be fetched back once the computation is completed.
  • In general these are explicit operations (cudaMemcpy).
MPI + CUDA

- MPI is handling main memory while CUDA kernels update the GPU memory. Explicit memory copy from the device to the CPU is necessary to ensure coherence.

```c
if( 0 == rank ) {
    cudaMemcpy(host_buffer, device_buffer, size, cudaMemcpyDeviceToHost);
    MPI_Send(host_buffer, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
} else { // assume MPI rank 1
    MPI_Recv(host_buffer, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    cudaMemcpy(device_buffer, host_buffer, size, cudaMemcpyHostToDevice);
}
```
Unified Virtual Addressing (UVA)

**No UVA: Multiple Memory Spaces**
- Devices have similar ranges of memory.
- Impossible to know where a memory range belongs to

**UVA: Single Address Space**
- Devices have continuous ranges of memory (managed by the hardware and OS).
- A memory address clearly identifies the hardware device hosting the memory

UVA: One address space for all CPU and GPU memory
- No need to alter libraries, they can how identify on which device the memory is located
Nvidia GPUDirect

- Allowed pinned pages to be shared between different users
- No need for multiple intermediary buffers to ready the data to be sent over the NIC

CUDA 3.1
Nvidia GPUDirect P2P

No GPUDirect P2P

GPUDirect P2P

- P2P (Peer-to-Peer) allows memory to be copied between devices on the same node without going through the main memory.
Nvidia GPUDirect RDMA

No GPUDirect RDMA

- Push the data out of the GPU directly into the NiC (or other hardware component).
  - Implement standard parts of the PCI-X protocol
if( 0 == rank ) {
    cudaMemcpy(host_buffer, device_buffer, size, cudaMemcpyDeviceToHost);
    MPI_Send(device_buffer, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
} else { // assume MPI rank 1
    MPI_Recv(device_buffer, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
    cudaMemcpy(device_buffer, host_buffer, size, cudaMemcpyHostToDevice);
}

• Explicit memory copy from the device to the CPU is **not** necessary to ensure coherence.

• Data now flows directly between the local and remote memory (independent on the location of the memory).
CUDA-aware MPI

```c
if( 0 == rank ) {
    MPI_Send(device_buffer, size, MPI_CHAR, 1, tag, MPI_COMM_WORLD);
} else { // assume MPI rank 1
    MPI_Recv(device_buffer, size, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status);
}
```

---

**OpenMPI 1.7.4 MLNX FDR IB (4X) Tesla K40**

![Graph showing bandwidth and latency](image)

- **CUDA-aware MPI with GPUDirect RDMA**
- **CUDA-aware MPI**
- **regular MPI**

**Latency (1 byte)**
- 19.04 us
- 16.91 us
- 5.52 us

**Versions:**
- MVAPICH2 1.8/1.9b
- OpenMPI 1.10
- CRAY MPI (MPT 5.6.2)
- IBM Platform MPI (8.3)
- SGI MPI (1.08)
Debugging

• Commercial tools (DDT, TV, ...)

• If possibility to export xterm:
  mpirun –np 2 xterm –e gdb –args <my app args>

• If not, add a sleep (or a loop around a sleep in your applications) and use "gdb –p <pid>" to attach to your process (once connected to the same node where the application is running)

• gdb can execute GDB commands from a FILE (with --command=FILE, -x)
Profiling

- Non-CUDA application: valgrind (free), or vtune (Intel), Score-P, Tau, Vampir
- CUDA application: nvprof from CUDA