MPI + MPI: Using MPI-3
Shared Memory As a Multicore Programming System
William Gropp
www.cs.illinois.edu/~wgropp
Likely Exascale Architectures


Figure 2.1: Abstract Machine Model of an exascale Node Architecture
Applications Still MPI- Everywhere

• Benefit of programmer-managed locality
  ♦ Memory performance nearly stagnant
  ♦ Parallelism for performance implies locality must be managed effectively

• Benefit of a single programming system
  ♦ Often stated as desirable but with little evidence
  ♦ Common to mix Fortran, C, Python, etc.
  ♦ But...Interface between systems must work well, and often don’t

• E.g., for MPI+OpenMP, who manages the cores and how is that negotiated?
Why Do Anything Else?

• Performance
  ♦ May avoid memory (though probably not cache) copies
• Easier load balance
  ♦ Shift work among cores with shared memory
• More efficient fine-grain algorithms
  ♦ Load/store rather than routine calls
  ♦ Option for algorithms that include races (asynchronous iteration, ILU approximations)
• Adapt to modern node architecture
Performance Bottlenecks with MPI Everywhere

• Classic Performance Model
  ♦ $T = s + r_n$
  ♦ Model combines overhead and network latency ($s$) and a single communication rate $1/r$
  ♦ Good fit to machines when it was introduced (esp. if adapted to eager and rendezvous regimes)
  ♦ But does it match modern SMP-based machines?
2.1 Overarching Abstract Machine Model

We begin with a single model that highlights the anticipated key hardware architectural features that may support exascale computing. Figure 2.1 pictorially presents this as a single model, while the next subsections describe several emerging technology themes that characterize more specific hardware design choices by commercial vendors. In Section 2.2, we describe the most plausible set of realizations of the single model that are viable candidates for future supercomputing architectures.

2.1.1 Processor

It is likely that future exascale machines will feature heterogeneous nodes composed of a collection of more than a single type of processing element. The so-called fat cores that are found in many contemporary desktop and server processors characterized by deep pipelines, multiple levels of the memory hierarchy, instruction-level parallelism and other architectural features that prioritize serial performance and tolerate expensive memory accesses. This class of core is often optimized to run a small number of hardware threads with an emphasis on efficient execution of system services, system runtime, or an operating system.

The alternative type of core that we expect to see in future processors is a thin core that features a less complex design in order to use less power and physical die space. By utilizing a much higher count of the thinner cores a processor will be able to provide high performance if a greater degree of parallelism is available in the algorithm being executed. Application programmers will therefore need to consider the uses of each class of core; a fat core will provide the highest performance and energy efficiency for algorithms where little parallelism is available or the code features complex branching schemes leading to thread divergence, while a thin core will provide the highest aggregate processor performance and energy efficiency where parallelism can be exploited, branching is minimized and memory access patterns are coalesced.

2.1.2 On-Chip Memory

The need for more memory capacity and bandwidth is pushing node architectures to provide larger memories on or integrated into CPU packages. This memory can be formulated as a cache if it is fast enough or, alternatively, can be a new level of the memory system architecture. Additionally, scratchpad memories (SPMs) are an alternate way for cache to ensure a low latency access to data. SPMs have been shown to be more energy-efficient, have faster access time, and take up less area than traditional hardware cache [14]. Going forward, on-chip SPMs will be more prevalent and programmers will be able to configure the on-chip memory as cache.
Modeling the Communication

• Each link can support a rate $r_L$ of data

• Data is pipelined (Logp model)
  ♦ Store and forward analysis is different

• Overhead is completely parallel
  ♦ $k$ processes sending one short message each takes the same time as one process sending one short message
A Slightly Better Model

• Assume that the sustained communication rate is limited by
  ♦ The maximum rate along any shared link
    • The link between NICs
  ♦ The aggregate rate along parallel links
    • Each of the “links” from an MPI process to/from the NIC
A Slightly Better Model

• For k processes sending messages, the sustained rate is
  \[ \text{min}(R_{NIC-NIC}, kR_{CORE-NIC}) \]

• Thus
  \[ T = s + \frac{kn}{\text{Min}(R_{NIC-NIC}, kR_{CORE-NIC})} \]

• Note if \( R_{NIC-NIC} \) is very large (very fast network), this reduces to
  \[ T = s + \frac{kn}{kR_{CORE-NIC}} = s + \frac{n}{R_{CORE-NIC}} \]
Observed Rates for Large Messages

Not double single process rate

Reached maximum data rate
Time for PingPong with k Processes
Hybrid Programming with Shared Memory

- MPI-3 allows different processes to allocate shared memory through MPI
  - `MPI_Win_allocate_shared`
- Uses many of the concepts of one-sided communication
- Applications can do hybrid programming using MPI or load/store accesses on the shared memory window
- Other MPI functions can be used to synchronize access to shared memory regions
- Can be simpler to program than threads
Creating Shared Memory Regions in MPI

- **MPI_COMM_WORLD**
- `MPI_Comm_split_type` (COMM_TYPE_SHARED)
- **Shared memory communicator**
  - `MPI_Win_allocate_shared`
  - **Shared memory window**
  - **Shared memory window**
  - **Shared memory window**
Regular RMA windows vs. Shared memory windows

- Shared memory windows allow application processes to directly perform load/store accesses on all of the window memory
  - E.g., \( x[100] = 10 \)
- All of the existing RMA functions can also be used on such memory for more advanced semantics such as atomic operations
- Can be very useful when processes want to use threads only to get access to all of the memory on the node
  - You can create a shared memory window and put your shared data
int main(int argc, char ** argv)
{
    int buf[100];

    MPI_Init(&argc, &argv);
    MPI_Comm_split_type(..., MPI_COMM_TYPE_SHARED, ..., &comm);
    MPI_Win_allocate_shared(comm, ..., &win);

    MPI_Win_lockall(win);

    /* copy data to local part of shared memory */
    MPI_Win_sync(win);

    /* use shared memory */

    MPI_Win_unlock_all(win);

    MPI_Win_free(&win);
    MPI_Finalize();

    return 0;
}
Example: Using Shared Memory with Threads

- Regular grid exchange test case
  - 3D regular grid is divided into subcubes along the xy-plane, 1D partitioning
  - Halo exchange of xy-planes: P0 -> P1 -> P2 -> P3...
  - Three versions:
    - MPI only
    - Hybrid OpenMP/MPI model with loop parallelism, no explicit communication: "hybrid naïve"
    - Coarse grain hybrid OpenMP/MPI model, explicit halo exchange within shared memory: "hybrid task", threads essentially treated as MPI processes, similar to MPI SM

- A simple 7-point stencil operation is used as a test SPMV
Internode Halo Performance

![Graph showing Internode Halo Performance with various configurations and their corresponding time in seconds. The graph includes bars and lines indicating different performance metrics such as time spent in communication, naivety, and total wallclock time. The x-axis represents different configurations (1T OMP task, 8T OMP task, 4T OMP task, 2T OMP task, 1T OMP task, 16T OMP naive, 8T OMP naive, 4T OMP naive, 2T OMP naive, 1T repl_only). The y-axis shows time in seconds, ranging from 0.0 to 1.0. The legend explains the bars as blue for time spent in communication, red for naivety, and black for total wallclock time. The graph indicates varying performance outcomes across different configurations.]
Summary

- Unbalanced interconnect resources require new thinking about performance
- Shared memory, used directly either by threads or MPI processes, can improve performance by reducing memory motion and footprint
- MPI-3 shared memory provides an option for MPI-everywhere codes
- Shared memory programming is hard
  - There are good reasons to use data parallel abstractions and let the compiler handle shared memory synchronization
Thanks!

- Philipp Samfass
- Luke Olson
- Pavan Balaji, Rajeev Thakur, Torsten Hoefler
- ExxonMobile
- Blue Waters Sustained Petascale Project