Today's Class

Coping with Intra-Node Data Motion
- Sources of Overhead
- MPI+KLT Model (Shared Memory)

Coping with Network Communication
- Sources of Overhead
- MPI+ULT Model (Overdecomposition)

Hierarchical Model
- Communication-Tolerant Programming
High-Performance Computing

HPC is an essential tool in science and technology

Supported by Supercomputers

Compute Nodes
- Multi-core Processors
- Memory Hierarchy (RAM/L3/L2/L1)
- Many-Core Devices (NVIDIA GPUs)

Interconnect
- Network Interfaces (NIC)
- Routers
- Links (Circuit Board, Copper, Optical)

Aerospace Engineering

Computational Biology

Earth Sciences

Astrophysics
Big Challenge: Cost of Communication

- Communication comprises a large part of the running time.
  - E.g., up to 68% in genome assembly codes\(^1\).

- Increasing towards Exascale:
  - Increasing processor core density.
  - Memory performance / capacity not keeping up.
  - Larger supercomputer interconnects.

- Challenge: Develop Communication-Tolerant Applications.
  - Reduce Network Communication Costs
  - Reduce Intra-Node Communication Costs

\(^1\)“HipMer: an extreme-scale de novo genome assembler”, E. Georganas et al, SC12, 2012
Intra-Node Data Motion
Structured Grid Stencil Solver

- Uses a 5-point Stencil.
- Iteratively approaches a solution.
Most HPC applications are programmed under the *Bulk-Synchronous Model.*
- Iterates among separate *computation* and *communication* phases.
- Communication is agnostic about node locality.
Anatomy of a Supercomputer: Node

Example: NERSC Cori Phase I (*Haswell*) Node

**System Memory (126GB)**

**NUMA Domain 0 (63 GB)**

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>L3 (40MB)</td>
<td>L3 (40MB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L1d (32KB)</td>
<td>L1d (32KB)</td>
</tr>
<tr>
<td>L1i (32KB)</td>
<td>L1i (32KB)</td>
</tr>
<tr>
<td>Core P#0</td>
<td>Core P#0</td>
</tr>
<tr>
<td>Core P#1</td>
<td>Core P#1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Core P#15</td>
<td>Core P#15</td>
</tr>
</tbody>
</table>

**NUMA Domain 1 (63 GB)**

<table>
<thead>
<tr>
<th>Processor 0</th>
<th>Processor 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>L3 (40MB)</td>
<td>L3 (40MB)</td>
</tr>
<tr>
<td>L2 (256KB)</td>
<td>L2 (256KB)</td>
</tr>
<tr>
<td>L1d (32KB)</td>
<td>L1d (32KB)</td>
</tr>
<tr>
<td>L1i (32KB)</td>
<td>L1i (32KB)</td>
</tr>
<tr>
<td>Core P#0</td>
<td>Core P#0</td>
</tr>
<tr>
<td>Core P#1</td>
<td>Core P#1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Core P#15</td>
<td>Core P#15</td>
</tr>
</tbody>
</table>

**Intra-Node Data Motion**
Figure 2.13: Memory/Performance ratio of supercomputers since 2001. Data source: [114]
Roofline Model: Predicts the performance of an application on a given multi-core node.

Example Applications:
A) Matrix Multiplication (High Arithmetic Intensity)
B) Structured Grid Solver (Low Arithmetic Intensity)

Optimizations
- Optimize for NUMA/Cache Locality (e.g. Cache Blocking)
- Increase Arithmetic Intensity (e.g. Prevent unnecessary data motion)

Parallel Programming Models play a key role in this!

Read: S. Williams et al. “Roofline: An Insightful Visual Performance Model for Multicore Architectures”
Kernel-Level vs. User-Level Threads

**Keypoints:**
- Thread Execution is Preemptive by the OS (interrupt-based)
- Allows Multi-core Parallelism
  - E.g.: OpenMP, Pthreads

**Kernel Level Threads**

- Process
- Code
- Memory
- Segments
- Main Routine
- Thread 1
- Thread 2
- Thread n
- Core 1
- Core 2
- Core n
- Operating System
  - Scheduled
- Architecture

**User Level Threads**

- Process
- Code
- Memory
- Segments
- Routine Scheduler
- Routine 1
- Routine 2
- Routine n
- Main Thread
- Core 1
- Core 2
- Core n
- Operating System
  - Scheduled
- Architecture

**Keypoints:**
- Thread Execution is non-preemptive
- Concurrent (not parallel) Execution
- E.g.: Boost Coroutines
Hybrid MPI+X Models

**MPI+KLT:** Use MPI for communication across Nodes / NUMA Domains

Use a Kernel-Level Thread (KLT) library to share the same memory address space.

**Example:** MPI + OpenMP - Uses #pragma to automatically distribute the workload among KLTs.
MPI+MPI-3: Use Two-sided MPI for communication across Nodes / NUMA Domains

Use MPI-3 embedded RMA operations for communication inside nodes
Coping with Intra-Node Data Motion

Traditional MPI Application:

OS Processes (Private Address Space)

<table>
<thead>
<tr>
<th>MPI Rank</th>
<th>Core</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Problem: Intra-node data motion causes significant overheads in the running time.

Hybrid (MPI+X) Approach:

OS Process

<table>
<thead>
<tr>
<th>MPI Rank 0</th>
</tr>
</thead>
</table>

Threading Library e.g., OpenMP (Shared Address Space)

<table>
<thead>
<tr>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

Core 0.Core 1.Node 0

KLTs can share data without buffering → reduces in-node communication data motion.
**Hybrid Model: Approach I**

**Benefit:** MPI and Threading Sections are well-separated

**Drawback:** Requires process-wide synchronization -> Nobody can start computing until all boundaries arrive

```c
main()
{
    grid = initialize();

    for (Iterations)
    {
        #pragma omp master
        {
            MPI_Isends(Entire Boundary)...
            MPI_Irecvvs(Entire Boundary)...
            MPI_Waitall();
        }

        #pragma omp parallel for
        for (i in NY)
            for (j in NX)
                Res[i][j] = compute(grid,i,j);

        #pragma omp barrier
    }
}
```
Hybrid Model: Approach II

**Drawback:** Requires a careful interaction between two programming interfaces.

**Benefit:** Threads compute/communicate independently -> Some can start computing while others still wait

```c
main()
{
    grid = initialize();

    #pragma omp parallel
    for (Iterations)
    {
        myBoundaryPartY = myThreadId...
        myBoundaryPartX = myThreadId...

        MPI_Isends(myBoundaryPartX...
        MPI_Irecvs(myBoundaryPartY...
        MPI_Waitall();

        for (i in myGridPartX)
            for (j in myGridPartY)
                Res[i][j] = compute(grid,i,j);
        #pragma omp barrier
    }
}
```
Thread Concurrency Problem

- Boundary threads use MPI as communication backend.
- MPI implements a process-wide lock, which limits communication concurrency.
- Non-Contiguous Datatypes are particularly problematic:

```plaintext
MPI_Isend(strideType)
1) Pack strided data into a hidden contiguous buffer.
2) Transmit buffered data to destination rank.

MPI_Irecv(strideType)
1) Receive incoming data into a hidden contiguous buffer.
2) Unpack data into destination with strides.
```
A way to cope with this problem is to split MPI operations:

- MPI_Isend $\Rightarrow$ MPI_Pack + MPI_Isend (contiguous)
- MPI_Irecv $\Rightarrow$ MPI_Irecv (contiguous) + MPI_Unpack
After splitting the operations, threads can perform concurrent packing/unpacking.

**MPI_Pack**(strideType)
- Pack strided data into an explicit contiguous buffer.
- Thread Safe

**MPI_Isend**(contiguous)
- Transmit buffered data to destination rank.
- MPI_Lock

**MPI_Irecv**(contiguous)
- Receive incoming data into an explicit contiguous buffer.
- MPI_Lock

**MPI_Unpack**(strideType)
- Unpack data into destination with strides.
- Thread Safe
**Result Comparison**

Comparing 2 Variants:

**Hybrid** + Non-Contiguous Communication

**Hybrid** + Packing/Unpacking + Contiguous Communication
Network Overhead
Anatomy of a Supercomputer: Interconnect

- Latency

Network Communication

+ Latency
Network Topologies

$\textbf{High Radix Routers } \sim 2\sqrt{N}$

$> \text{Max 4 hops per data packet} <$

Sources: NERSC.gov
J. Kim et al. “Technology-Driven, Highly-Scalable Dragonfly Topology”
S. Ashby et al. “The opportunities and challenges of exascale computing”
Equation: Time to send a message from node A to node B (No Traffic)

\[ T_{a,b} = L_{a,b} + S_m / B_{max} \]

Max interconnect bandwidth
Message Size
Latency between nodes A and B

Observation I: The message size is typically small (Kb to Mb range) in scientific algorithms.

Observation II: The bandwidth of interconnects keeps increasing steadily.

Conclusion: It is unlikely that the Sm/Bmax component will dominate in Exascale supercomputers.

Question: What about the cost of latency (L_{a,b})?
Cost of Network Communication (II)

Equation: Latency between nodes $A$ and $B$

$$L_{a,b} = H_{a,b} \times h + s$$

- Fixed (startup) time per message
- Average time per hop
- **Average # hops between any $A$ and $B$**

**Observation I:** The average number of hops will increase in Exascale experiments.

**Observation II:** The time per hop is algorithm-independent.

**Conclusion:** It is likely that Latency will be the dominant cost of communication in Exascale.

**Concern:** Algorithms with *fine-grained* communication are especially susceptible to the cost of latency.

**Question:** What can be done to ameliorate this cost?
Network Communication

Traditional Approach: Iterative *Bulk-Synchronous* Model

```
For (iterations)
{
    -- Compute --  
    Issue Communication Requests
    -- Wait for Requests --
}
```

- **Problem:** Naive Bulk-Synchronous applications suffer from the full cost of network.
Coping Strategies

- **Communication Avoiding**: Performing less and/or more efficient communication\(^1\).
- **Communication Hiding**: Overlap communication with computation\(^2\).

For (iterations)
{
    -- Compute --
    Issue Communication Requests
    -- Wait for Requests --
}\nFor (iterations)
{
    -- Compute (Dependent) --
    Issue Communication Requests
    -- Compute (Independent) --
    -- Wait for Requests --
}\n
**Core Usage Timeline:**

**Better Core Usage**

**Shortfalls of manual refactoring:**
- Embeds policy decisions into the application code.
- Transformations are hard to maintain (architecture-dependent).
- For complex applications, these transformations are unviable.

\(^1\)"Communication Avoiding and Overlapping for Numerical Linear Algebra", Georganas, E et al. In SC’12
\(^2\)"Parallel programming in split-c", D. E. Culler, et al., In SC ’93
Strategy: Inner/Outer Halo Computation

1) Begin Boundary Cell Exchange
2) Compute Inner Part (boundary-independent)
3) Wait for boundary to arrive
4) Compute Outer Part to Complete Iteration
Strategy: Tiling

Divides a rank’s subgrid into tiles.

Steps:
1. For every Tile:
   - Compute
   - Issue Communication
2. Wait for all requests

Achieves overlap by operating at a Tile-level.
Strategy: Intermediate Buffering

Cannon: Solves a Dense Square $C = AB$ Matrix Multiplication

- Communicates $A_{i,j}$ and $B_{i,j}$ submatrices along columns/rows, respectively.
- Uses $dgemm$ (Intel MKL) to calculate partial $C += A_i \times B_i$ results.
- Overlap: Exchange next submatrix on additional buffers while computing the current one
Fact: Not all subranks incur the same communication cost.

Idea\(^1\): Prioritize subranks with higher communication cost to execute first.

Effect: Maximize computation while longer communication is performed.

---

Advanced: Rank Prioritization

Adaptive Algorithm:

- Higher Priority (mostly Node Boundary)
- Medium Priority (Mixed Boundaries)
- Low Priority (Inner Tasks)

We tested 3 prioritization schemes:

**Hybrid Default**
No priority scheme. Execution order is mostly random.

**Hybrid + Boundary**
Priority scheme as described in the previous slide.

**Hybrid + Center**
The opposite scheme than the one described in the previous slide.
General Approach: Overdecomposition

Dividing the workload into more ranks than processor cores.

- Overdecomposition is a more intuitive approach.
- Can hide the cost of network communication.
- However, it also increases intra-node data motion.
Granularity of MPI Applications

Observation: MPI programs typically reach optimal performance when $P = c$, where $c = \text{number of cores}$.

Equation: Granularity of an SPMD program. Defines how the workload is divided among processes.

$$G = \frac{N}{P}$$

Drawback: $P = c$ fixes granularity to the underlying architecture and does not allow oversubscription.
MPI+User-Level Threads

MPI+ULT: Use user-level threads to run more than one MPI processes per OS Process.

<table>
<thead>
<tr>
<th>MPiP0</th>
<th>MPiP1</th>
<th>ULT</th>
<th>OS Process</th>
<th>Core 0</th>
<th>NUMA Domain 0</th>
<th>Node 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPiP2</td>
<td>MPiP3</td>
<td>ULT</td>
<td>OS Process</td>
<td>Core 1</td>
<td>NUMA Domain 1</td>
<td></td>
</tr>
<tr>
<td>MPiP4</td>
<td>MPiP5</td>
<td>ULT</td>
<td>OS Process</td>
<td>Core 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPiP6</td>
<td>MPiP7</td>
<td>ULT</td>
<td>OS Process</td>
<td>Core 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPiP8</td>
<td>MPiP9</td>
<td>ULT</td>
<td>OS Process</td>
<td>Core 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPiP10</td>
<td>MPiP11</td>
<td>ULT</td>
<td>OS Process</td>
<td>Core 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPiP12</td>
<td>MPiP13</td>
<td>ULT</td>
<td>OS Process</td>
<td>Core 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MPiP14</td>
<td>MPiP15</td>
<td>ULT</td>
<td>OS Process</td>
<td>Core 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Equation: Granularity of a Threaded SPMD program.

\[ G = \frac{N}{(c \times V)} \]

Benefit: V can be configured, allowing the user to adjust granularity independently from architecture.
Problem: Surface-to-Volume Ratio

- Overdecomposition in MPI+ULT requires communicating extra ghost cells.
- Suffers from increased in-node communication.

There is an excess of in-node communication (e.g., ~1.30x more, in this case.)
Effect of Overdecomposition

1x
No Overdecomposition

2x
Oversubscription

4x
Oversubscription

No overlap

Partial Overlap. Better core usage.

Full Overlap, but Excessive overhead.
Hierarchical Decomposition
Overview of MPI+X Models

- MPI is to this day the most widely used model for programming scientific application.

- However, it presents two main limitations towards Exascale computing:
  - MPI is subjected to *data duplication*.
    - This can be solved by the Hybrid (MPI+KLT) Model through shared memory but...
    - Requires combining two programming interfaces and extensive synchronization.

  - MPI does not prescribe any means for overdecomposition.
    - This can be solved by the MPI+ULT Model but...
    - Cannot prevent the surface-to-volume problem.

- What if we integrate both (KLT) and (ULT) approaches in a single model?
  - Automatic support for both oversubscription and shared memory.
  - A single interface for communication inside and across nodes.
Hierarchical Model

1. A Hierarchical application executes as a set of processes.
2. Multiple Hierarchical Ranks live within each process.
3. A rank has a unique pair identifier: (Process ID, Local Rank ID)
4. Sibling ranks communicate through shared memory.
5. Ranks use MPI to communicate across processes.
Hierarchical Overdecomposition

Hierarchical Model:
- Benefits from the overlapping effects of overdecomposition.
- Reduces (and even avoids) intra-node data motion.
- It still requires process-wide synchronization → Region Graphs.
for (int i = 0; i < Iterations; i++)
{
    for (n in Neighbors) MPI_Irecv(recvBuf(n)←n);  Receive
    Compute();                                          Compute
    for (n in Neighbors) MPI_Pack(bCells→sendBuf(n)); Pack
    for (n in Neighbors) MPI_Isend(sendBuf(n)→n); }  Send
    MPI_Waitall(MPIRequests);
    for (n in Neighbors) MPI_Unpack(gCells←recvBuf(n)); Unpack

Observation: Ranks in traditional MPI applications operate sequentially.
**Dependency Graph Decomposition**

**Dependency-Driven Loop**

```c
#pragma mate graph
for (int i = 0; i < Iterations; i++)
{
    #pragma mate region(receive) depends(unpack*)
    for (n in Neighbors) if (!Mate_isLocal(n)) MPI_Irecv(recvBuf(n) ← n);

    #pragma mate region(compute) depends(pack*, unpack*)
    Compute();

    #pragma mate region(pack) depends(compute, receive)
    for (n in Neighbors) if (!Mate_isLocal(n)) MPI_Isend(sndBuf(n) → n);

    #pragma mate region(send) depends(pack)
    for (n in Neighbors) if (!Mate_isLocal(n)) MPI_Isend(sndBuf(n) → n);

    #pragma mate region(unpack) depends(compute, receive)
    for (n in Neighbors) if (!Mate_isLocal(n)) MPI_Unpack(gCells ← rcvBuf(n));
}
```

- Integrates MPI operations into the Dependency Graph.
- Expose parallelism across iterations increasing overlap potential.

Read: S. Martin & S. Baden. "MATE, a Unified Model for Communication-Tolerant Scientific Applications"
Let's Take a look...

Observations:
- Rank (0,1) requires boundary data from its neighbor’s previous iteration.
- Requires a process-wide barrier (#pragma omp barrier, MPI_Barrier)

Idea:
- Enable dependencies to cross rank boundaries.
Hierarchical Model Execution

Hierarchical Overdecomposition

- Provide light-weight synchronization.
- Enable ranks to depend only on their actual neighbors.
- Maximize process-wide concurrency.

+ Core Regions
Workers execute ranks opportunistically, maximizing overlap potential.
Test Case: 3D Stencil Solver

Solves the 3D Poisson equation using a 13-point stencil.

Cori Phase I Strong Scaling Study:
- Test at multiple communication intensities.
- 256 to 1024 nodes (8k to 32k cores)
- 43% of Cori Phase I

Variants:
- Basic-MPI
- Overlapping MPI (Olap-MPI)
- MPI+OpenMP (MPI_THREAD_MULTIPLE)
- Hierarchical Model
  - Overdecomposition = 8 Ranks per Core
Results: Cori Phase I (Haswell)
Less is Better

Basic-MPI spends 47% of time in communication.

Olap-MPI achieves overlap but hardly achieves any benefits.

MPI+OpenMP does reduce 64% of buffering but does not produce a speedup.

1. Olap-MPI and H-Model suffer from loss in cache-efficiency.

2. MPI+OpenMP and H-Model suffer from MPI/Threading limitations.

H-Model Reduces 74% of Network Communication and 39% of Buffering Time
Core Usage Timelines: 32k Cores

Basic-MPI
(8 Ranks)

Hierarchical Model
(64 Ranks)
Results: Cori Phase II (KNL)

![Bar chart showing performance (TFlop/s) for Basic-MPI, MPI+OpenMP, and Hierarchical Model across NERSC Cori Phase II Cores. The chart highlights a 52% Communication Time Reduction and a 52% Intra-Node Data Motion Reduction.]
Hierarchical Cannon

Uses two-levels of the Cannon's Algorithm.

- Submatrices are divided at a process level but compute/communicate at a rank level.
- Uses pointer exchanges at a local rank level.
- Uses row/column inter-rank dependencies.
Matrix Size Scaling Study

For small matrices, fixed overheads dominate running time. For large matrices, computation time dominates. Middle-range matrices are optimal test cases for communication-tolerance.
Results: Cori Phase II (KNL)

Weak Scaling Study ($n_0 = 24576^2$)

![Graph showing performance with different scaling factors.](image)

- **Speedup:** 1.18
- **48% Communication Time Reduction**

Performance (TFlop/s)

NERSC Cori Phase II Cores

<table>
<thead>
<tr>
<th>NERSC Cori Phase II Cores</th>
<th>Basic-MPI</th>
<th>Olap-MPI</th>
<th>Hierarchical Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>4096</td>
<td>1.00</td>
<td>1.00</td>
<td>1.04</td>
</tr>
<tr>
<td>16384</td>
<td>1.00</td>
<td>1.02</td>
<td>1.08</td>
</tr>
<tr>
<td>65536</td>
<td>1.00</td>
<td>1.02</td>
<td>1.08</td>
</tr>
</tbody>
</table>
Test Case #3: Cloverleaf3D

Finite-volume hydrodynamics benchmark.
- Employs a stencil operator over a 14-variable staggered grid.
- Each rank exchanges field boundaries with 26 neighbors in a 3-stage communication approach.
Load Imbalance

Occurs due to an uneven workload distribution across ranks.

- Manifests during communication/synchronization operations.
- Can become a large part of the application’s running time.
Hierarchical Balancing

1. Re-assigns rank distribution at a node-level.
2. Uses hierarchical overdecomposition to scatter workload at a process-level.
3. Shuffles ranks inside each node to smooth out peaks.
mpix_flowCart is a high-fidelity analysis package for aerodynamic design.

- Uses a multigrid with irregular meshes.
- Production code with hundreds of users.
- Load imbalance on 2048 Cori Haswell cores:

![Graph showing running time, imbalance reduction, and speedup](image)
Hierarchical Variant

1. Relies on overdecomposition to achieve overlap.
2. Avoids most of intra-node data motion by employing direct-copies.
Results: Cori Phase I (Haswell)

Strong Scaling Study (1024³ Cells)

Speedup: 1.19

34% Communication Time Reduction