The MATE Model

Rationale & Preliminary Results

Sergio Martin
PhD Candidate @ UC San Diego
GSRA @ Berkeley Lab

Argonne, IL 03/05/2018

UC San Diego
JACOBS SCHOOL OF ENGINEERING
Computer Science and Engineering

BERKELEY LAB

Challenges in Extreme Scale Computing

Big Challenge[1,2]: Exploit Massive Parallelism

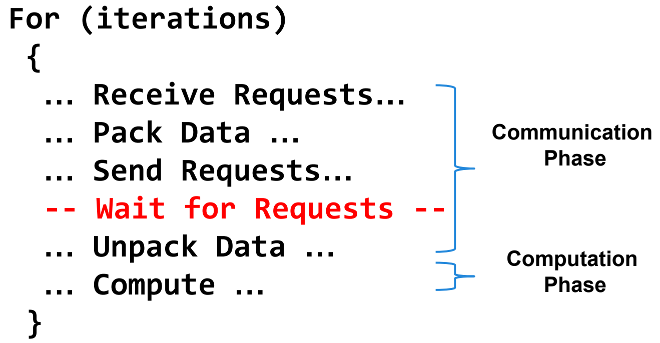
- Develop efficient multi-core and memory hierarchy-aware algorithms.
- Provide an adaptive response to load imbalance.
- **Mitigate the ever-growing cost of communication.**
    - Intranode Data Motion
    - Network Communication
    - Packing/Unpacking of Non-Contiguous Data

[1] **"The opportunities and challenges of exascale computing",** S. Ashby et al, Summary Report of the US DOE ASCR, 2010

[2] **"Algorithmic Challenges of Exascale Computing",** K. Yelick, Presentation, Lawrence Berkeley National Laboratory, 2012

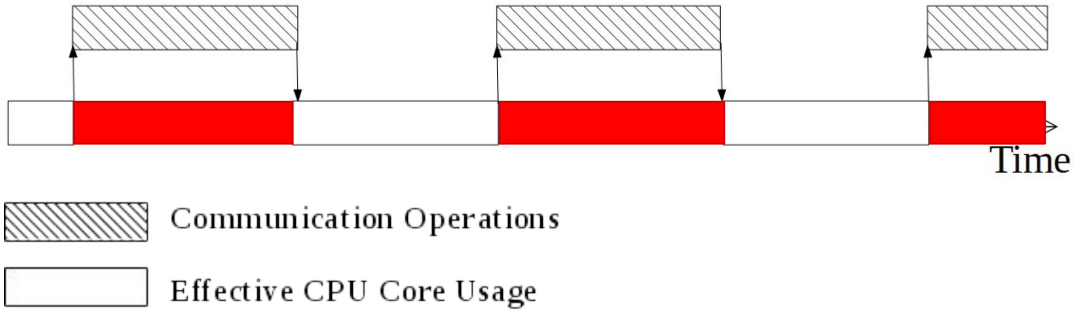
Anatomy of a Naive SPMD Application

Main Loop:



```
For (iterations)
 {
  ... Receive Requests...
  ... Pack Data ...
  ... Send Requests...
  -- Wait for Requests --
  ... Unpack Data ...
  ... Compute ...
 }
```

Core Usage Timeline:



- **Problem:** Naive SPMD MPI applications suffer from the full cost of communication.
- **Coping strategies:**
  - Communication Hiding Strategy: Overlap communication with computation[1,2].
  - Communication Avoiding Strategy: Performing less and/or more efficient communication[3].

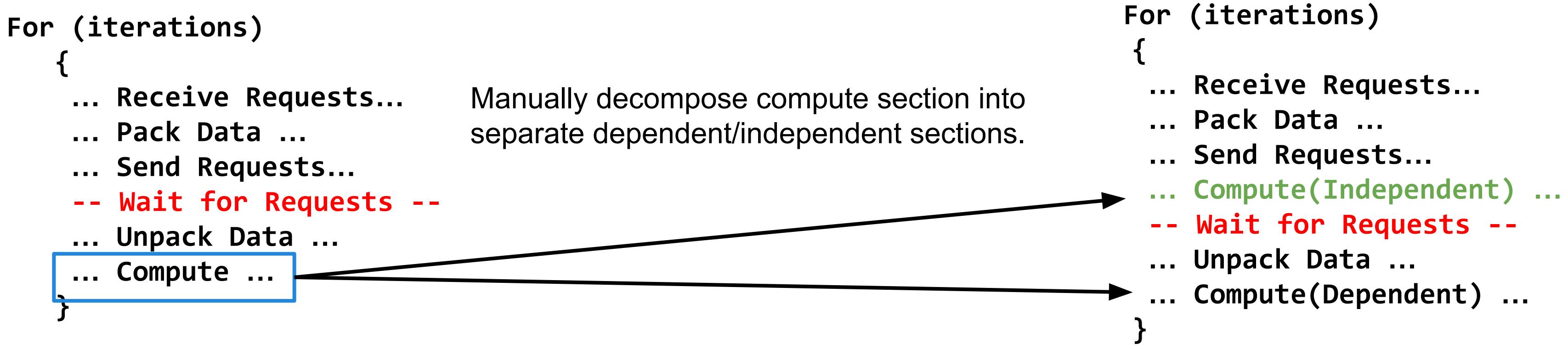
---

[1] **"A Programming Model for Block-Structured Scientific Calculations on SMP Clusters ",** Ph. D. Dissertation, '98

[2] **"Latency Hiding and Performance Tuning with Graph-Based Execution",** P. Cicotti and S. Baden. In DFM'11

[3] **"Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms",** E. Solomonik and J. Demmel. In EuroPar'01

Manual Optimization



```
For (iterations)
  {
   ... Receive Requests...
   ... Pack Data ...
   ... Send Requests...
   -- Wait for Requests --
   ... Unpack Data ...
   ... Compute ...
  }
```

Manually decompose compute section into separate dependent/independent sections.

```
For (iterations)
{
  ... Receive Requests...
  ... Pack Data ...
  ... Send Requests...
  ... Compute(Independent) ...
  -- Wait for Requests --
  ... Unpack Data ...
  ... Compute(Dependent) ...
}
```

What it entails to perform a manual optimization of a code:

- Requires embedding foreign logic into the solver part of the code.
- Transformations are hard to maintain (some are even architecture-dependent).

There are alternative ways to reduce communication cost.

Current PhD Project

MATE

Unified Model for Communication-Tolerant Scientific Applications

- **Employs a combination of 4 mechanisms to:**
    - Hide the cost of network communication.
    - Reduce the cost of on-node data motion.

- **It is comprised of:**
    - An annotation model (C/C++ #pragma) for dependency-driven execution.
    - A source-to-source code translator (ROSE Compiler Framework).
    - A runtime system between the application communication layer (MPI/CUDA/etc).

Mechanism I:
Task Overdecomposition

Task Overdecomposition

Observation:

- ❑ Typical execution of SPMD MPI applications instantiate one process per core.
- ❑ Instantiating more processes would only introduce additional scheduling overhead.

Idea:

- ❑ Interpret MPI ranks as reentrant functions (*virtualization*), not OS processes[1,2].
- ❑ Develop a user-level scheduler / runtime system.
- ❑ Instantiate more tasks than cores. Schedule them based on readiness[3].

Expected results:

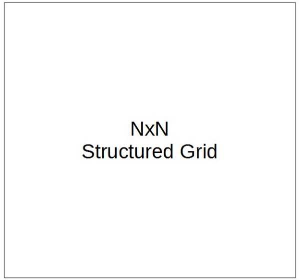
- ❑ A rank starts communication earlier while another performs computation.
- ❑ Realize communication and computation overlap.

[1] **"The Virtualization Model of Parallel Programming: Runtime Optimizations and the State of Art",** Laxmikant V. Kalé. In: LACSI'02.

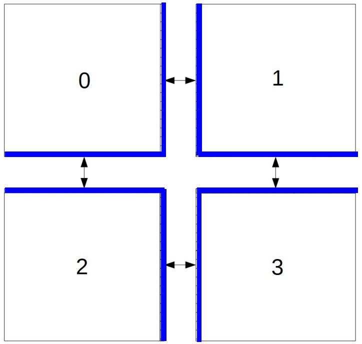
[2] **"FG-MPI: Fine-grain MPI for multicore and clusters",** H. Kamal and A. Wagner. In: IPDPSW'10.

[3] **"Asynchronous programming with Tarragon",** P. Ciccotti, S. Baden. In: HPDC'06.

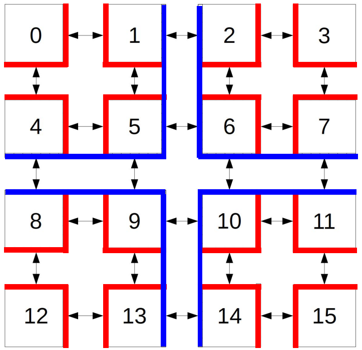
Task Overdecomposition



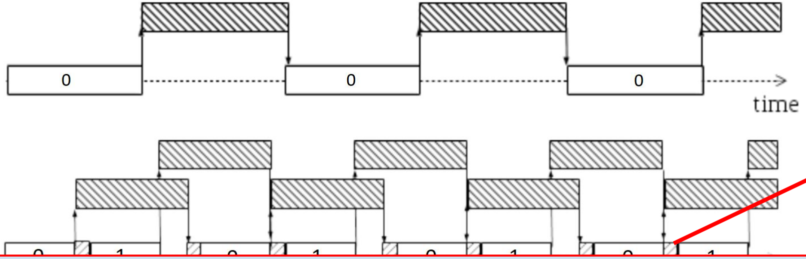
4 available cores



Typical MPI Decomposition
1 Subdomain / Core



Overdecomposed Grid
4 Subdomains / Core



No overlap

Additional Data Motion

Better core usage with

Observation: Overdecomposition refines task granularity but requires additional ***data motion***.

Let’s evaulate these effects experimentally.

Hardware Testbed: Cori KNL @ NERSC

NERSC Cori Phase II (KNL) Supercomputer:
9,688 Computing Nodes

Processor: Single-socket Intel "Knights Landing" with 68 cores per node @ 1.4 GHz

Memory: 96 GB DDR4 2400 MHz memory per node (8M page size).

Software:

- Cray-MPICH/7.6.2
- Intel icc compiler 18.0.1 (-O3)

Test Case: 13-Point Stencil Solver

Solves a 3D Poisson equation using the Jacobi Method.

3D Grid - 4096^3 Cells

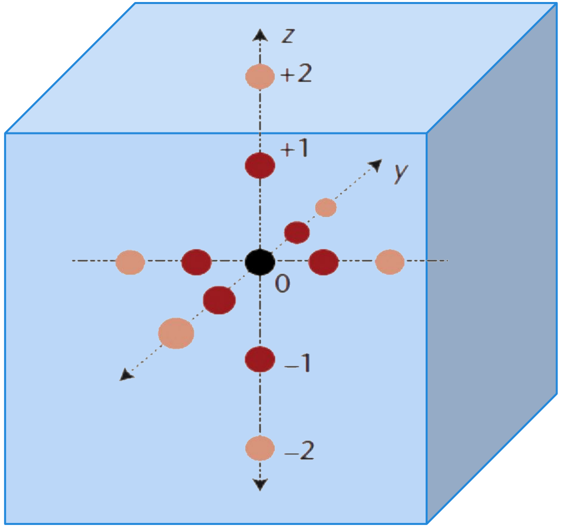
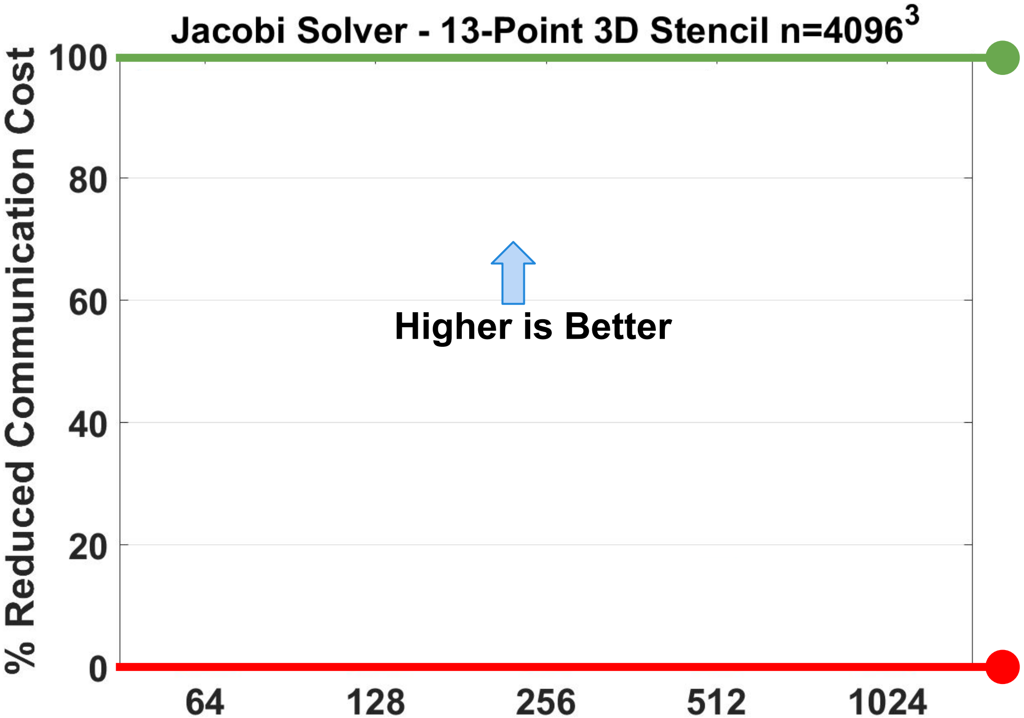


Image Source: *"Accelerating a 3D Finite-Difference Earthquake Simulation with a C-to-CUDA Translator"*, D. Unat et al.

Experiment Details:

- 64 cores per Cori KNL node (4 unused)
- Benefits from Hyperthreading
  - 128 MPI ranks per node
- Strong Scaling Study (64 to 1024 nodes)
  Goal: Obtain benefits at all subgrid sizes.

Experimental Results (Cori KNL)



Control Variants:

- Baseline MPI (128 ranks/node)
- No Communication (Upper Bound)
  - No Packing / Unpacking
  - No MPI Messages

Formula for % Comm Reduction:

$$\%CR = \frac{t_{Baseline} - t_{Variant}}{t_{Baseline} - t_{NoComm}}$$

Observation: Overdecomposition-only barely yields any benefits. Can we do better?

Mechanism II:
Code Regions & Dependencies

Code Regions & Dependencies

Observation:

- Overdecomposition refines task granularity allowing C/C overlap, **but**...
- Penalizes performance due to higher intranode data motion.

Idea:

- Subdivide the source code into smaller regions of code.
- Have code regions execute as soon as their dependencies are satisfied[1,2].

Expected results:

- Further refine granularity to expose more potential for C/C overlap.
- No additional additional ghost cells are required.

---

[1] **“Bamboo: Translating MPI Applications to a Latency-tolerant, Data-driven Form”,** T. Nguyen et al. In: SC’12.

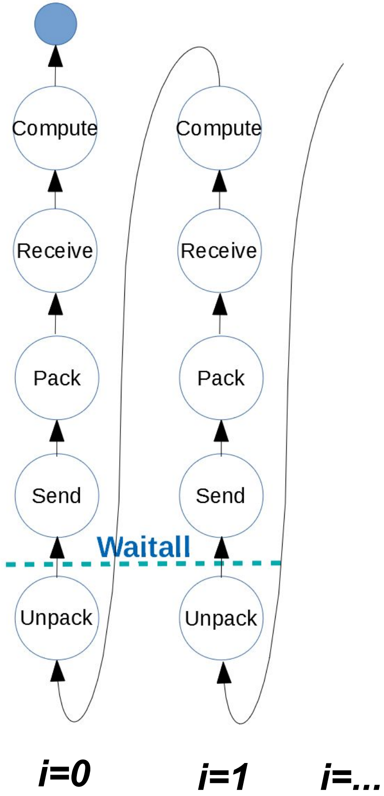
[2] **“Toucan - A Translator for Communication Tolerant MPI Applications”,** S. Martin, M. J. Berger, S. B. Baden. In: IPDPS’17.

Code Example: Stencil Solver

1D Stencil Solver

```
for (int i = 0; i < Iterations; i++)
{
  Compute();
  Swap(&U, &Un);

  MPI_Irecv(eastRecvBuffer, count_east, ...);
  MPI_Irecv(westRecvBuffer, count_west, ...);

  MPI_Pack(&Un[0],  count_east, eastSendBuffer, ...);
  MPI_Pack(&Un[nx], count_west, westSendBuffer, ...);

  MPI_Isend(eastSendBuffer, count_east, ...);
  MPI_Isend(westSendBuffer, count_west, ...);

  MPI_Waitall(requests);
  MPI_Unpack(&U[0],  eastRecvBuffer, EastRank);
  MPI_Unpack(&U[nx], westRecvBuffer, WestRank);
}
```

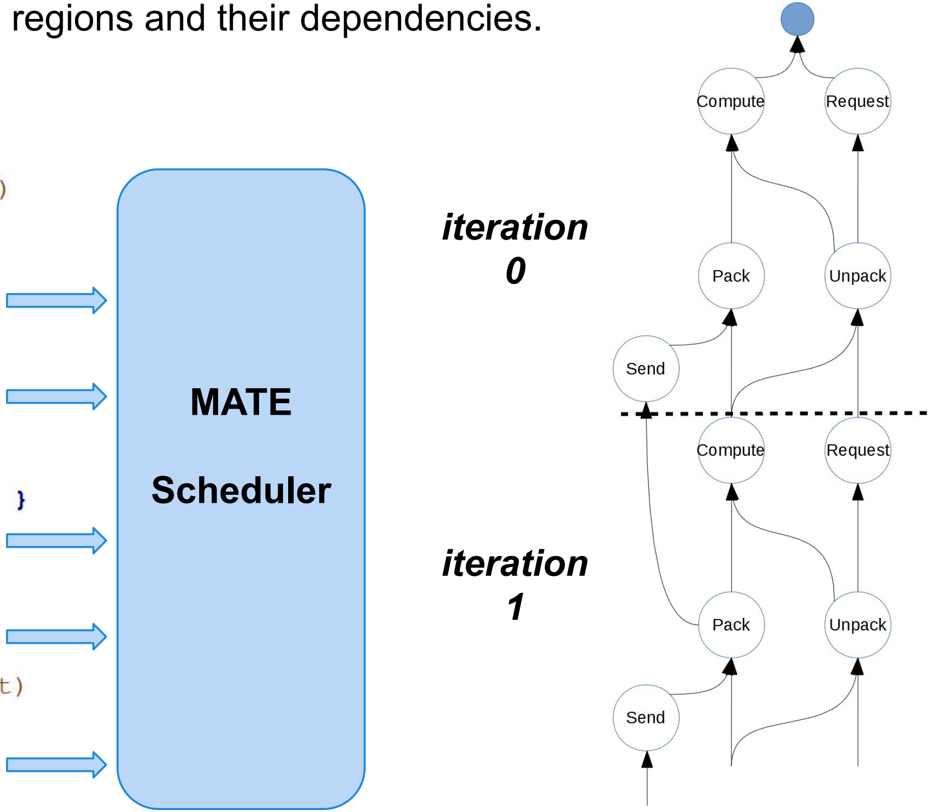
MPI Backend



MATE Dependency Graph

MATE provides a pragma-based syntax to delineate code regions and their dependencies.

```
#pragma mate graph
for (int i = 0; i < Iterations; i++)
{
    #pragma mate region (compute) depends (pack*, unpack*)
    { Compute();
      Swap(&U, &Un); }

    #pragma mate region (request) depends (unpack*)
    { MPI_Irecv(eastRecvBuffer, count_east, ...);
      MPI_Irecv(westRecvBuffer, count_west, ...); }

    #pragma mate region (pack) depends (compute, send*)
    { MPI_Pack(&Un[0],  count_east, eastSendBuffer, ...);
      MPI_Pack(&Un[nx], count_west, westSendBuffer, ...); }

    #pragma mate region (send) depends (pack)
    { MPI_Isend(eastSendBuffer, count_east, ...);
      MPI_Isend(westSendBuffer, count_west, ...); }

    #pragma mate region (unpack) depends (compute, request)
    { MPI_Unpack(&U[0],  eastRecvBuffer, EastRank);
      MPI_Unpack(&U[nx], westRecvBuffer, WestRank); }
}
```



MATE Translation Process

```
#pragma mate graph
for (int i = 0; i < Iterations; i++)
{
   #pragma mate region (compute) depends (pack*, unpack*)
   { Compute();
     Swap(&U, &Un); }

   #pragma mate region (request) depends (unpack*)
   { MPI_Irecv(eastRecvBuffer, count_east, ...);
     MPI_Irecv(westRecvBuffer, count_west, ...); }

   #pragma mate region (pack) depends (compute, send*)
   { MPI_Pack(&Un[0], count_east, eastSendBuffer, ...);
     MPI_Pack(&Un[nx], count_west, westSendBuffer, ...); }

   #pragma mate region (send) depends (pack)
   { MPI_Isend(eastSendBuffer, count_east, ...);
     MPI_Isend(westSendBuffer, count_west, ...); }

   #pragma mate region (unpack) depends (compute, request)
   { MPI_Unpack(&U[0],  eastRecvBuffer, EastRank);
     MPI_Unpack(&U[nx], westRecvBuffer, WestRank); }
}
```

```
int iCompute = 0, iRequest = 0, iPack = 0, iSend = 0, iUnpack = 0;

while(MATE_GetNextRegion(&regionId)) switch (regionId)
{
     case "compute":
      Compute();
      Swap(&U, &Un);
      if (++iCompute >= niterations) MATE_RemoveRegion("compute");
      break;

     case "request":
      MATE_Irecv(eastRecvBuffer, count_east, ...);
      MATE_Irecv(westRecvBuffer, count_west, ...);

     case "send":
      MATE_Isend(eastSendBuffer, count_east, ...);
      MATE_Isend(westSendBuffer, count_west, ...);
      if (++iSend >= niterations) MATE_RemoveRegion("send");
      break;

     case "unpack":
      MPI_Unpack(&U[0],  eastRecvBuffer, EastRank);
      MPI_Unpack(&U[nx], westRecvBuffer, WestRank);
      if (++iUnpack >= niterations) MATE_RemoveRegion("unpack");
      break;

     case __MATE_NOREGION:
      MATE_Yield(); break;
}
```

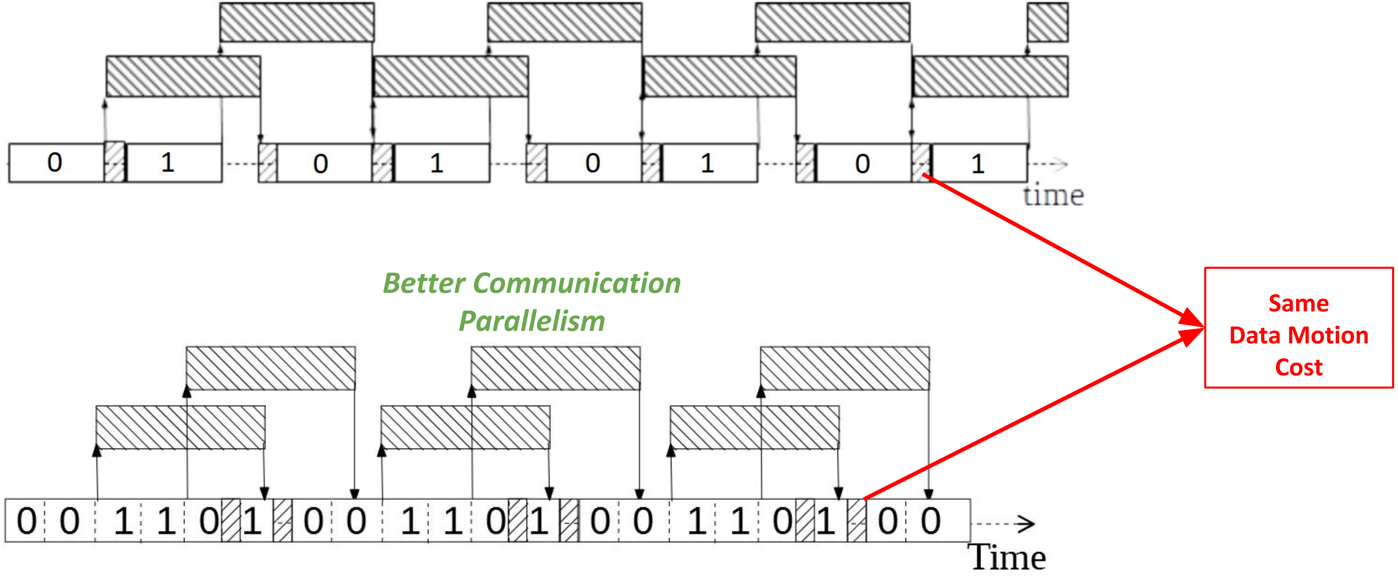
Prepended in Main():

```
MATE_AddRegions("compute", "request", "pack", "send", "unpack");
MATE_AddDependency("compute" → { "pack*", "unpack*"});
MATE_AddDependency("request" → "unpack*" );
MATE_AddDependency("pack" → { "compute", "send*" } );
MATE_AddDependency("send" → "pack" );
MATE_AddDependency("unpack" → { "compute", "request" } );
```

Core Usage Timeline

Overdecomposed
No Regions
2 Subdomains / Core

Overdecomposed
With Regions
2 Subdomains / Core
+ Code Regions

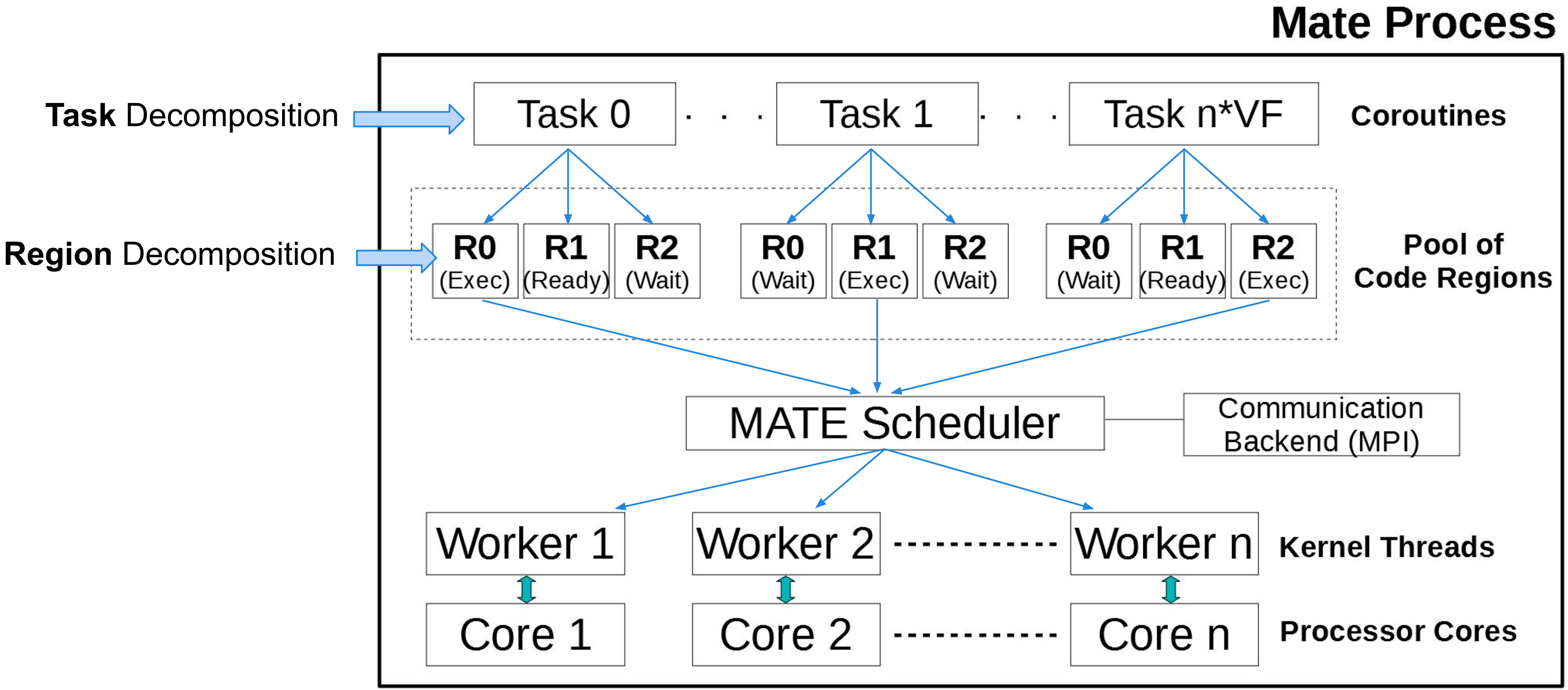


Better Communication Parallelism

Same
Data Motion
Cost

Observation: Code regions further refine granularity without additional ***data motion***.

MATE's Runtime System



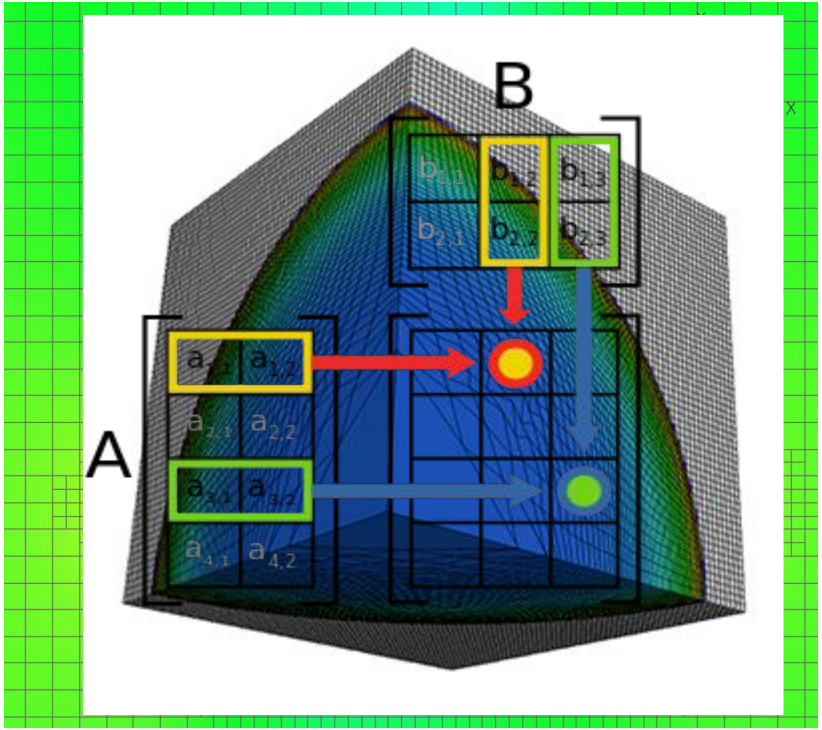
Toucan (IPDPS’17) Results

Platform:

- NERSC Edison (2x12-core)
- No Hyperthreading

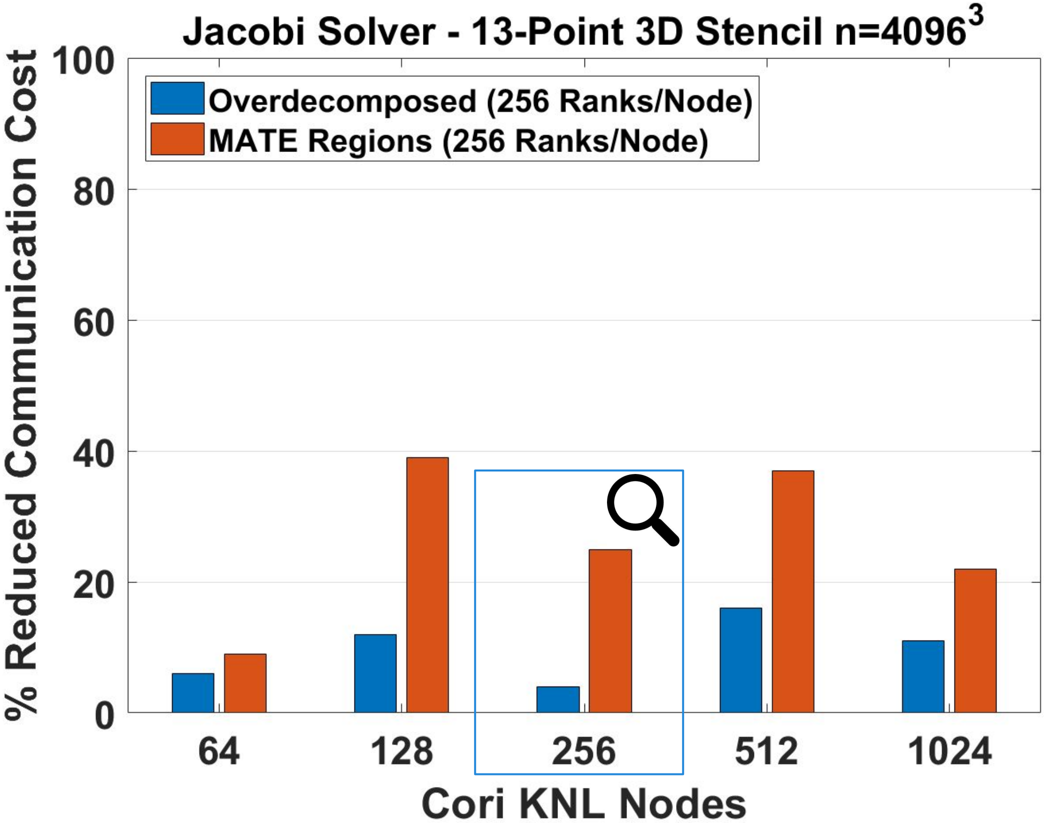
Test Cases:

- **Cannon 2D (Dense Linear Algebra)**
  (55% Comm Reduced @ 384 Nodes)
- **LULESH 2.0 (Unstructured Grid)**
  (72% Comm Reduced @ 576 Nodes)
- **Mpix_FlowCart (Unstructured MG)**
  (33% Comm Reduced @ 256 Nodes)

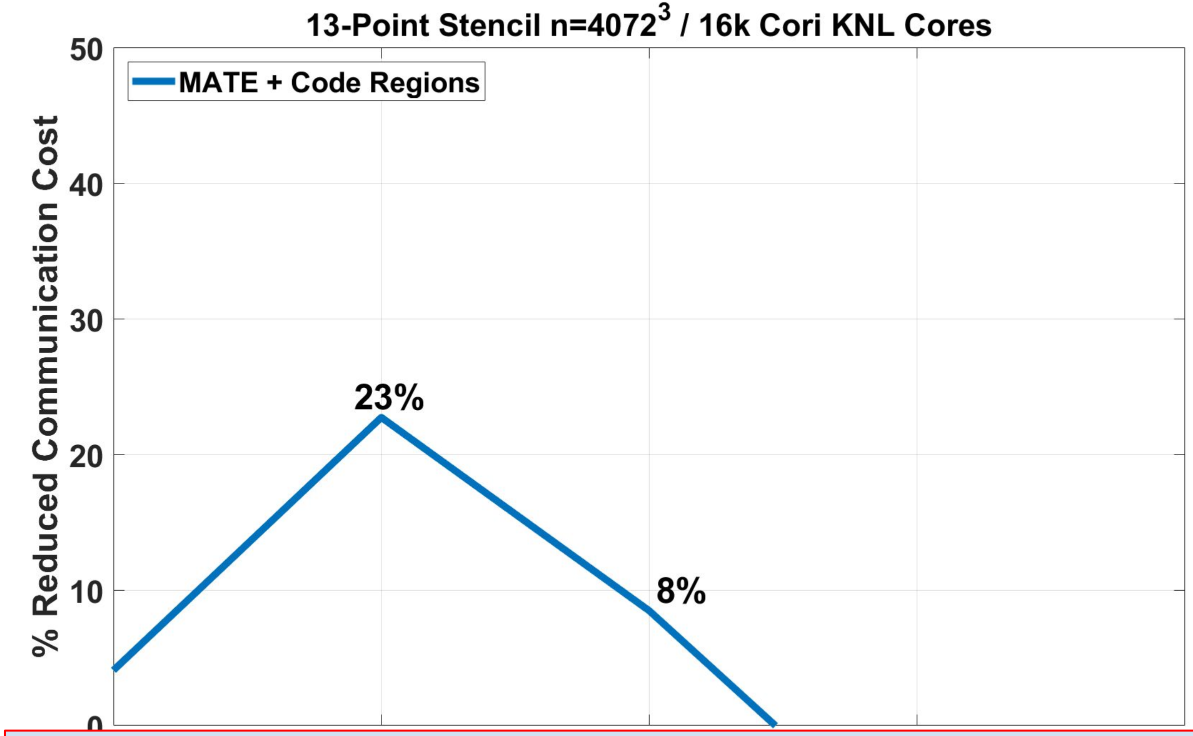


Source: NASA Ames Research Center

Experimental Results (Cori KNL)



Effect of Overdecomposition



Observation: We can only use a limited amount of overdecomposition.

Mechanism III: Hierarchical Decomposition

Hierarchical Decomposition

Observation:

- ❑ There are tasks living in the same node/process.
- ❑ Data is already present in memory. There’s no need for messaging.

Idea:

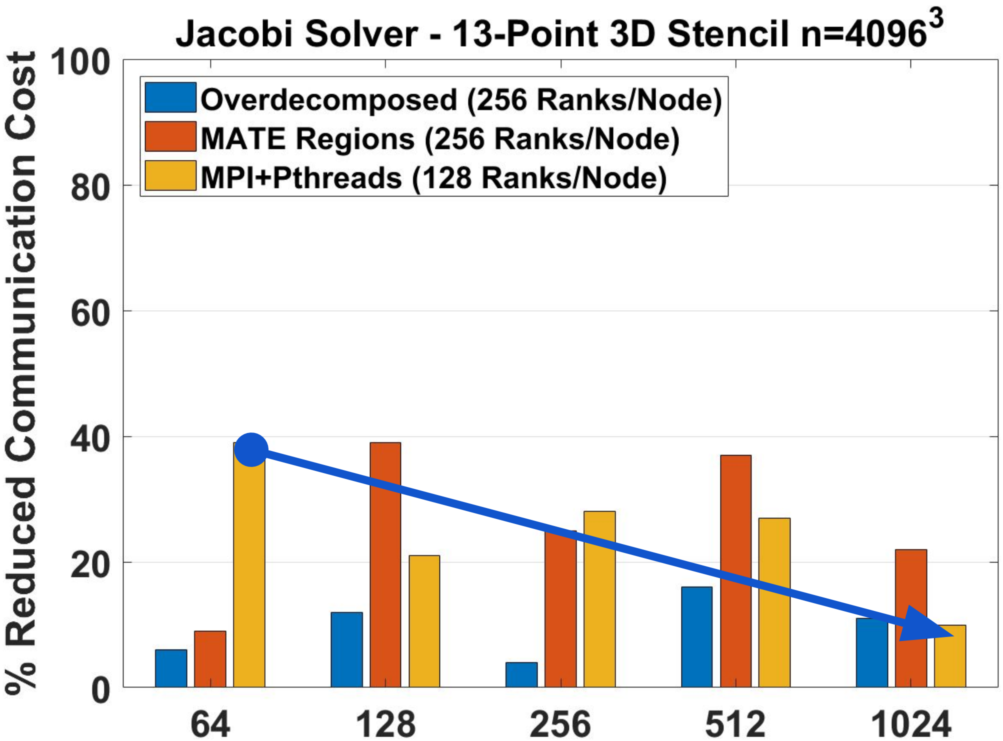
- ❑ Divide the problem grid once, one big subdomain per node/socket/process.
- ❑ Share subdomain among threads in the same address space[1,2].

Expected results:

- ❑ Local ranks can read boundary cells directly, without in-node communication.

Let’s evaluate the performance of such an MPI+X approach.

Experimental Results (Cori KNL)



MPI+Pthreads Configuration:
8 Threads per MPI Process
8 MPI Processes per Node

Observation: The benefits of MPI+X fades as we scale up.

Hierarchical Decomposition

Observation:

- ❑ Overdecomposition increases internal data motion.
- ❑ Data is already present in node.

Idea:

- ❑ Divide the problem grid once, one big subdomain per node (socket).
- ❑ Share subdomain among threads in the same address space[1,2].

Expected results:

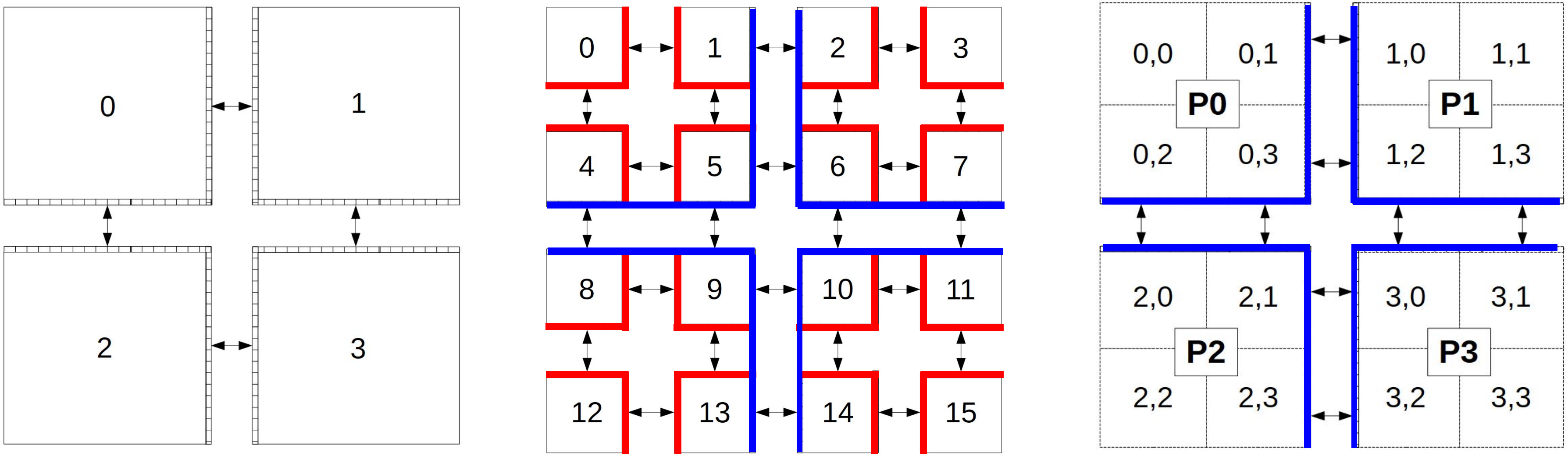
- ❑ Local ranks can read boundary cells directly, without in-node communication.

[1] **"Toward an Evolutionary Task Parallel Integrated MPI + X Programming Model",** R. Barrett et al. In: PMAM'15.

[2] **"MPI + MPI: A New Hybrid Approach to Parallel Programming with MPI Plus Shared Memory",** T. Hoefler et al. In: Computing 13.

MATE as a Unified Model

- **New Model:** Workload decomposed twice. Every subdomain is shared among multiple tasks.



Typical SPMD Decomposition
1 Subdomain / Core

Overdecomposed (x4)
4 Subdomains / Core

MATE Model
1 Subdomain / Core + 4 Tasks in Subdomain

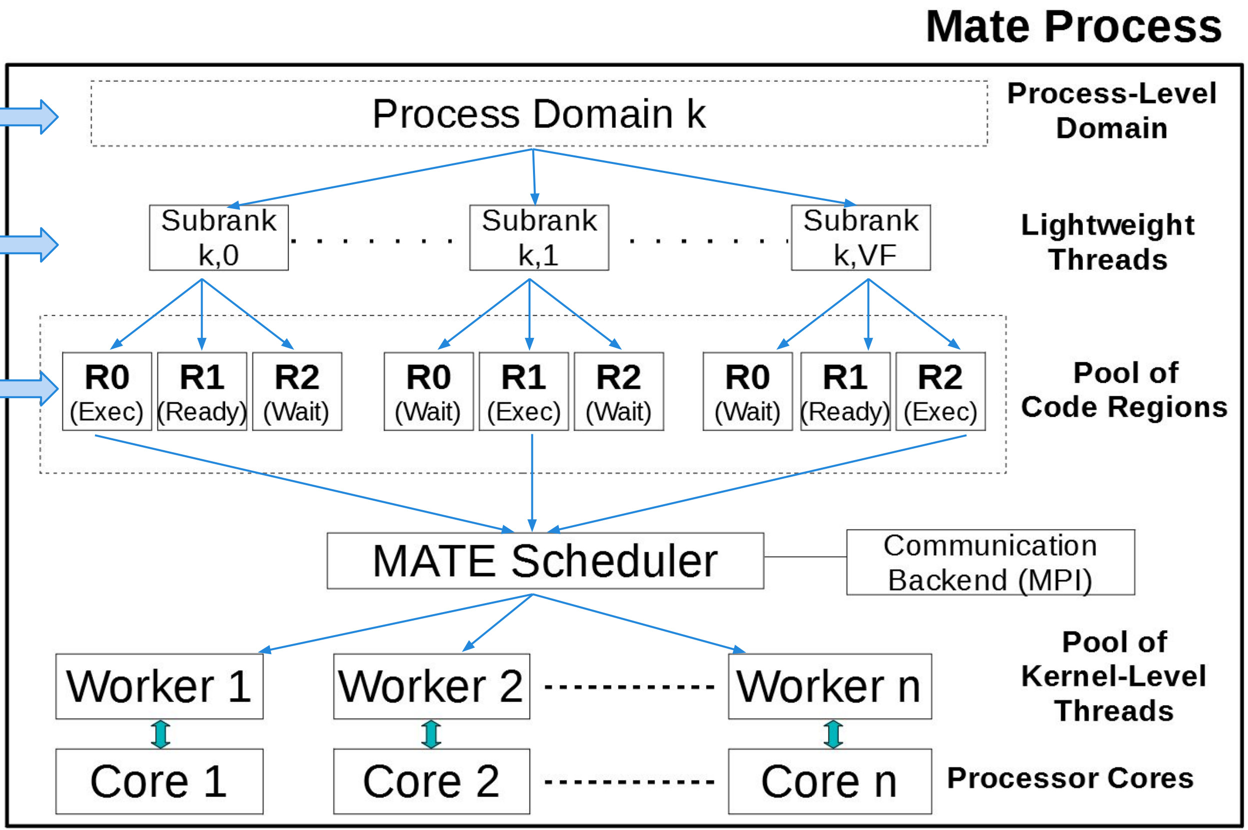
Observation I: Avoids in-node data motion as in **A+B models** (MPI+openMP, MPI+MPI, MPI+PThreads, etc).
Observation II: It does so in a single, **unified model**, instead of combining two agnostic models.

MATE Runtime System

Level 1 Decomposition

Level 2 Decomposition

Region Decomposition



Mate Process

Process Domain k

Process-Level Domain

Subrank k,0 . . . . Subrank k,1 . . . . Subrank k,VF

Lightweight Threads

R0 (Exec) **R1** (Ready) **R2** (Wait)

R0 (Wait) **R1** (Exec) **R2** (Wait)

R0 (Wait) **R1** (Ready) **R2** (Exec)

Pool of Code Regions

MATE Scheduler

Communication Backend (MPI)

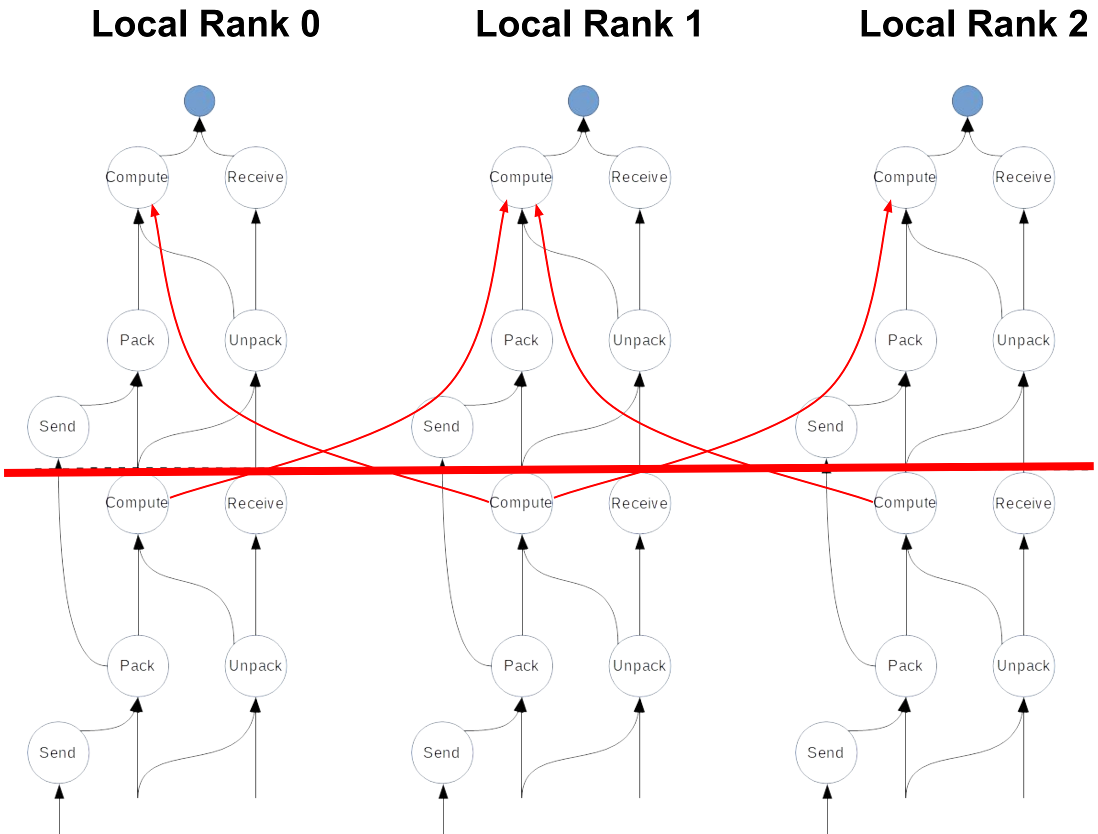
Worker 1 Worker 2 ------------ Worker n

Pool of Kernel-Level Threads

Core 1 Core 2 ------------ Core n

Processor Cores

MATE Local Synchronization Logic



Observations:

- Each task consumes its neighbor’s computations from previous iteration.
- Requires a process-wide barrier (#pragma omp barrier, MPI_Barrier)

Idea:

- Allow inter-task region dependencies
- Commutative, yet non-transitive.

Effect:

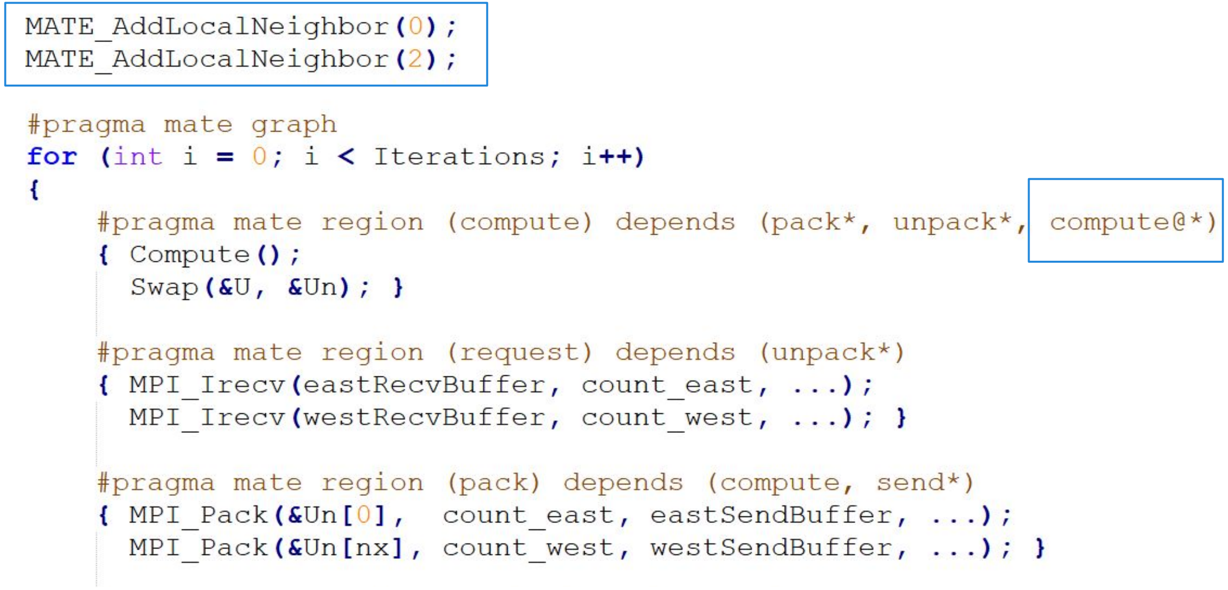
- Every local task is released **as soon as** its neighbors have finished their previous iteration’s *compute*.

MATE Local Synchronization Syntax

Syntax:

❑ Inform MATE of local neighbor ranks (*MATE_AddLocalNeighbor*)

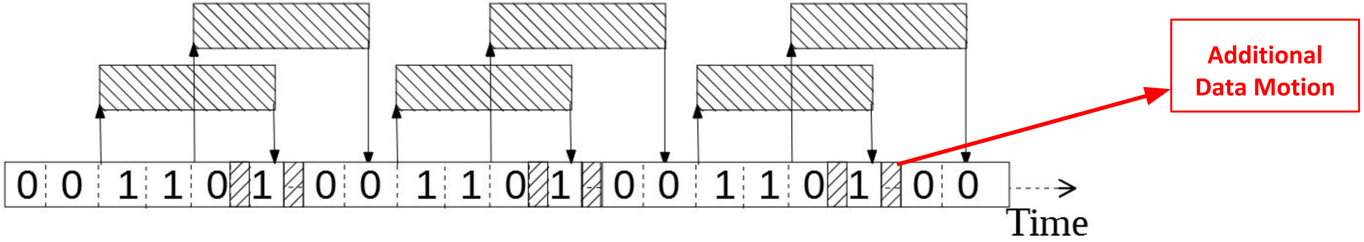
❑ Use ‘@’ to indicate that depended region belongs to neighbors.



```
MATE_AddLocalNeighbor(0);
MATE_AddLocalNeighbor(2);

#pragma mate graph
for (int i = 0; i < Iterations; i++)
{
    #pragma mate region (compute) depends (pack*, unpack*, compute@*)
    { Compute();
      Swap(&U, &Un); }

    #pragma mate region (request) depends (unpack*)
    { MPI_Irecv(eastRecvBuffer, count_east, ...);
      MPI_Irecv(westRecvBuffer, count_west, ...); }

    #pragma mate region (pack) depends (compute, send*)
    { MPI_Pack(&Un[0],  count_east, eastSendBuffer, ...);
      MPI_Pack(&Un[nx], count_west, westSendBuffer, ...); }
```

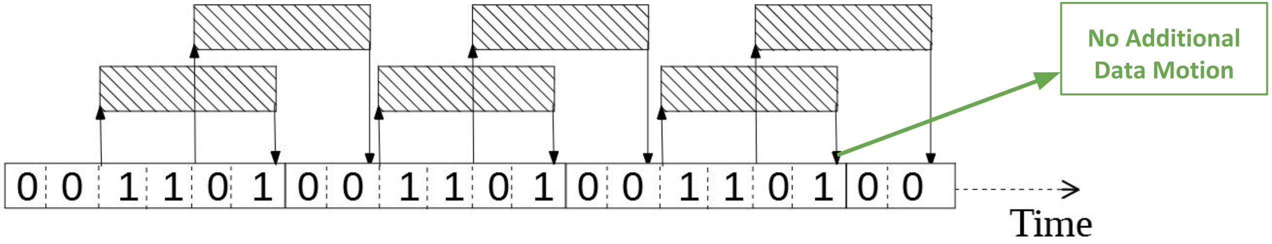
Core Usage Timeline

Overdecomposed
With Code Regions
2 Subdomains / Core
+ Dependencies



Additional Data Motion

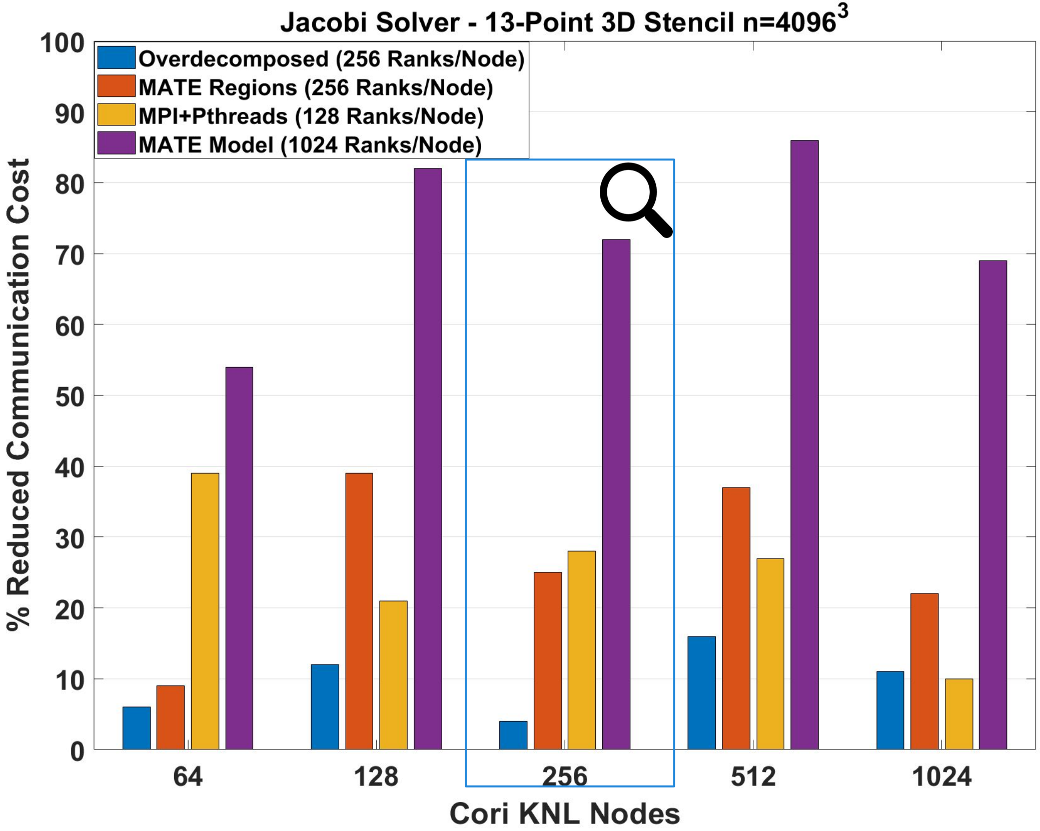
Overdecomposed
MATE Model
2 Subdomains / Core
+ Dependencies



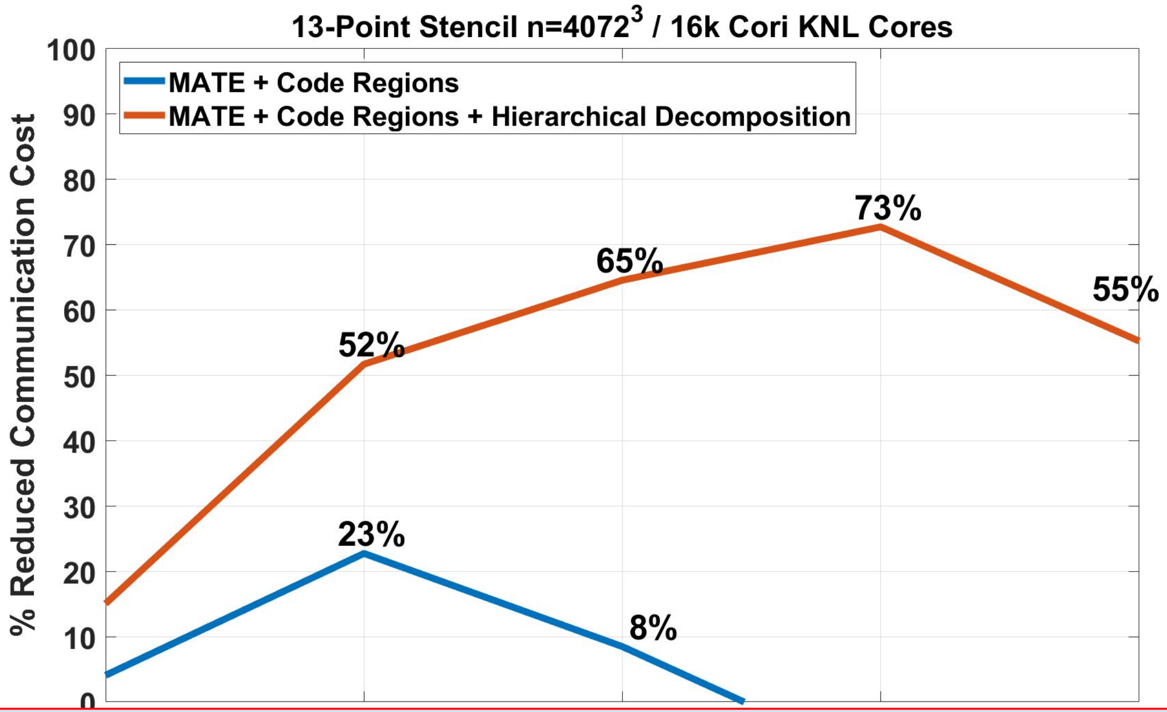
No Additional Data Motion

Observation: Using a hierarchical decomposition mitigates in-node ***data motion*** due to overdecomposition.

Experimental Results (Cori KNL)



Overdecomposition in MATE

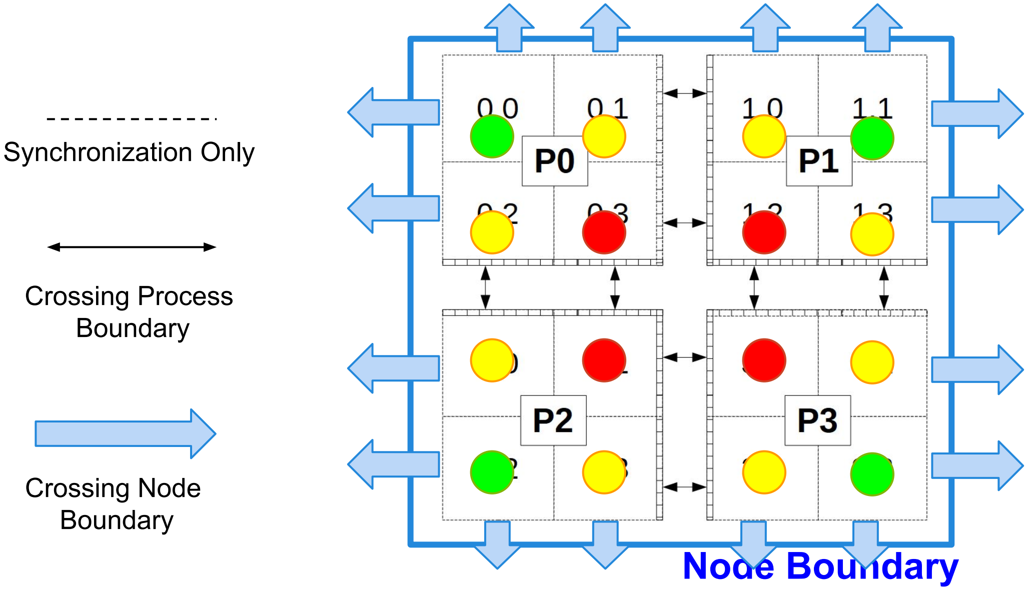


Observation: There is a **synergistic** effect in using Hierarchical Overdecomposition.

Mechanism IV: Communication-Based Prioritization

Communication-Based Prioritization

- **Fact:** Not all subranks incur the same communication cost.
- **Idea[1]:** Prioritize subranks with higher communication cost to execute first.
- **Effect:** Initialize costly communication first.



Adaptive Algorithm in MATE:

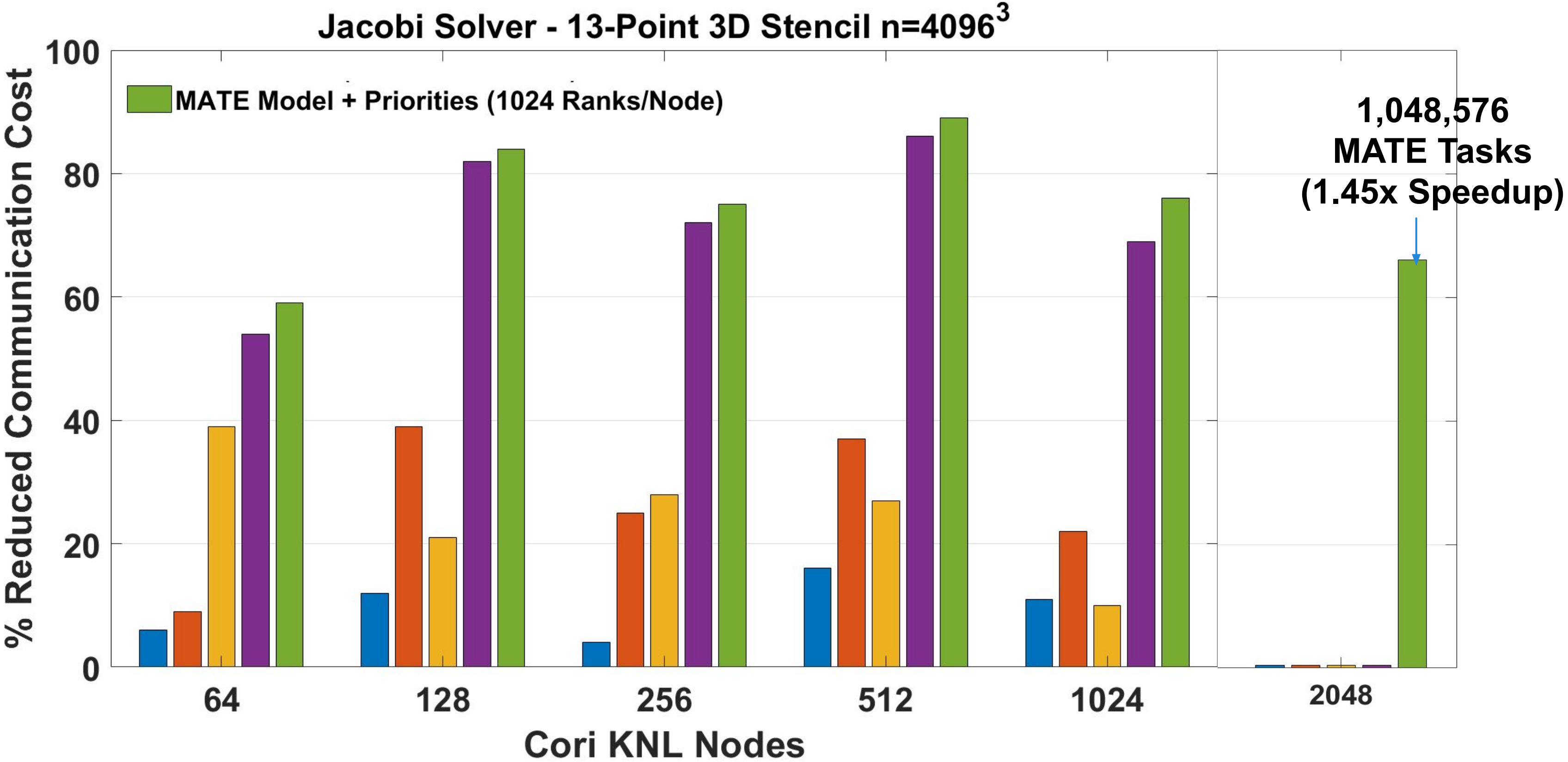
🟢 Higher Priority (mostly Node Boundary)

🟡 Medium Priority (Mixed Boundaries)

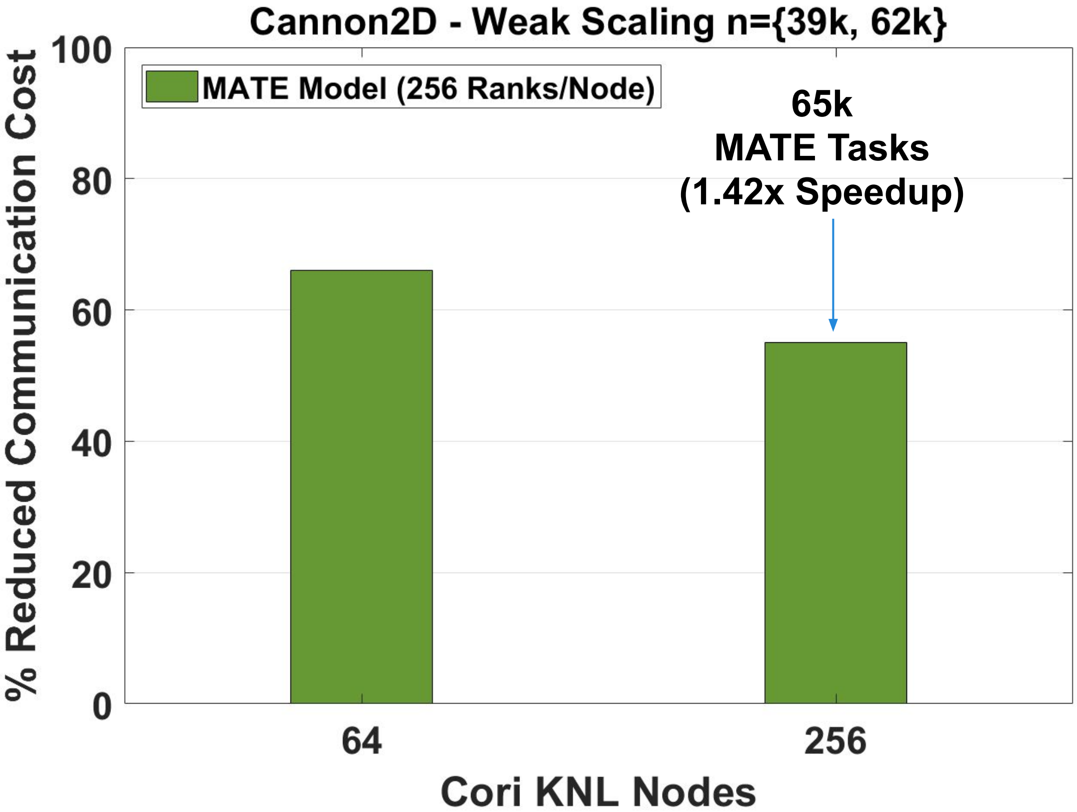
🔴 Low Priority (Inner Tasks)

[1] **"Performance tradeoffs in multi-tier formulation of a finite difference method"** S. B. Baden and D. Shalit. In: ICCS 2001.

Experimental Results (Cori KNL)



Cannon2D Results



Conclusions and Next Steps

MATE Model Conclusions

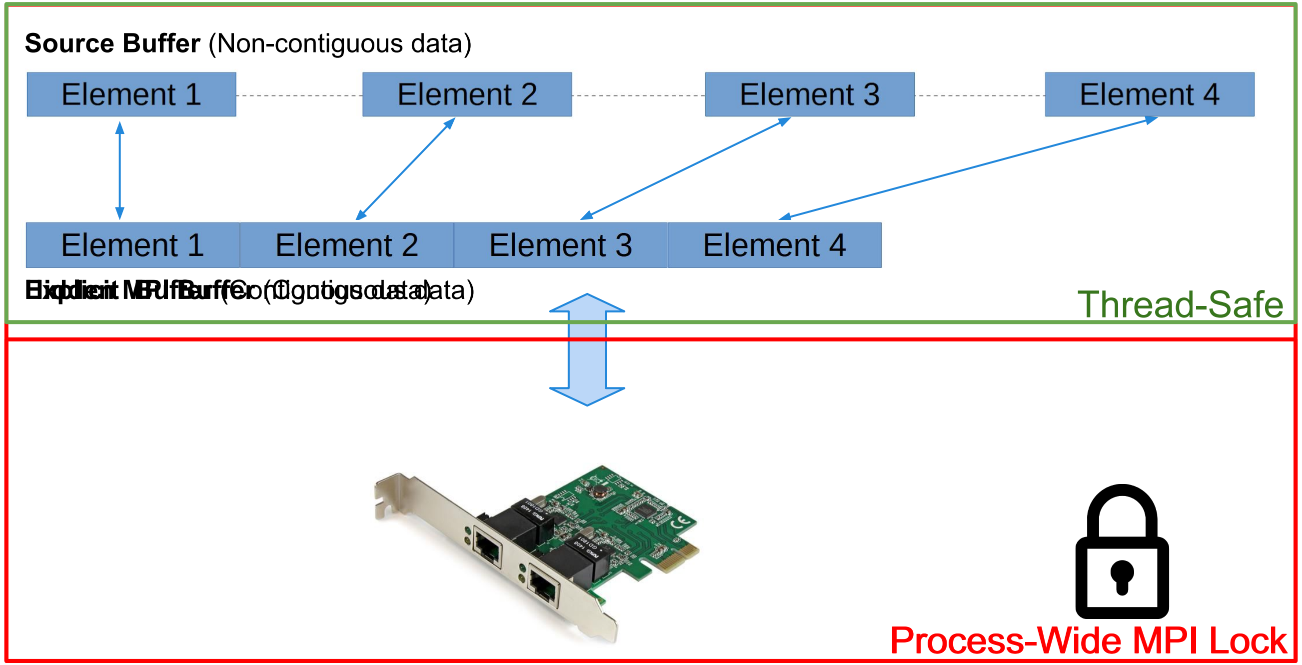
- **The overdecomposition-only approach is limited.**
    - It can realize communication/computation overlap, but...
    - Requires additional intranode data motion.
- **We can refine task granularity by splitting them into code regions.**
    - Regions can be independently scheduled based on their dependencies.
    - This does not introduce additional data motion.
- **Hierarchical decomposition solves the data motion problem.**
    - Enables higher levels of overdecomposition (x8 vs. x2) efficiently.
    - MATE’s inter-task dependencies enable efficient local synchronization.
- **Communication-based prioritization can improve performance.**
    - MATE can assign priorities adaptively during execution.

All of these mechanisms can be integrated into a single **unified** model

- **Limitations:**
    - Hierarchical decomposition requires **re-factoring** the work distribution logic.
    - Thread/MPI concurrency is still a limiting factor in MATE processes.

Hurdle: Thread Concurrency

- Non-contiguous data need to be packed before communicating.
- MPI implements a process-wide lock, which limits communication concurrency.
- **Partial Solution:** Perform thread-safe packing (MPI_Pack/Unpack) before issuing a send/recv.



Source Buffer (Non-contiguous data)

Element 1 Element 2 Element 3 Element 4

Element 1 Element 2 Element 3 Element 4

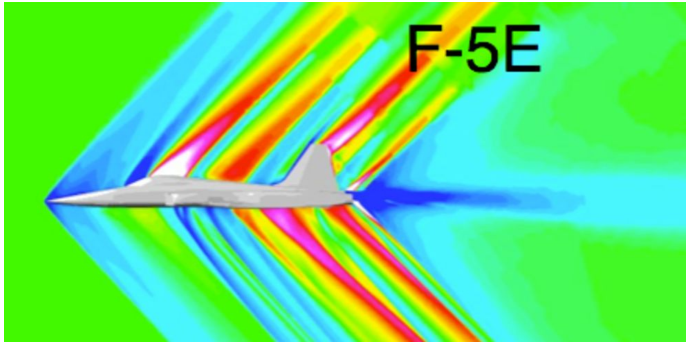
Explicit MPI Buffer (Contiguous data)

Thread-Safe

Process-Wide MPI Lock

MATE Workers

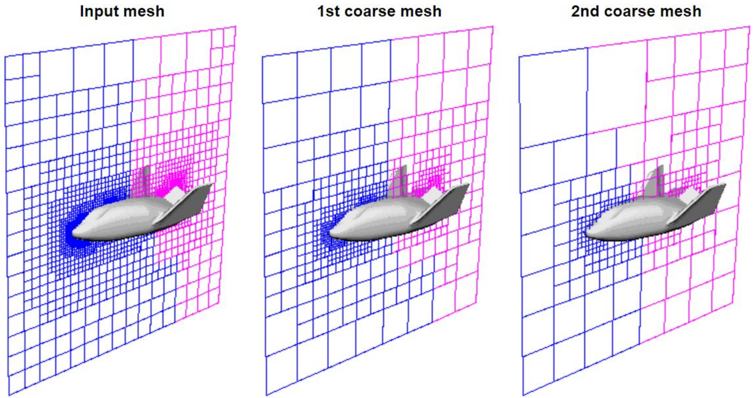
Next Step: Mpix_FlowCart



Mpix_FlowCart is an analysis package for aerodynamic design.
- Production code developed by NASA Ames and NYU.
- It has hundreds of users.

Mpix_FlowCart is particularly challenging:
- Uses a multigrid with irregular meshes.
- High volume of asymmetric communication.
- Benefits from hyperthreading (128 ranks), therefore
- Overdecomposition-only approach is too punishing.
- Performs reads/updates on the same grid (Gauss-like).

Applying the MATE model will require:
- Creating a two-level SFC that divides the grid so that:
- Virtualized ranks can compute without data hazards.
- Deal with worker thread/MPI concurrency.

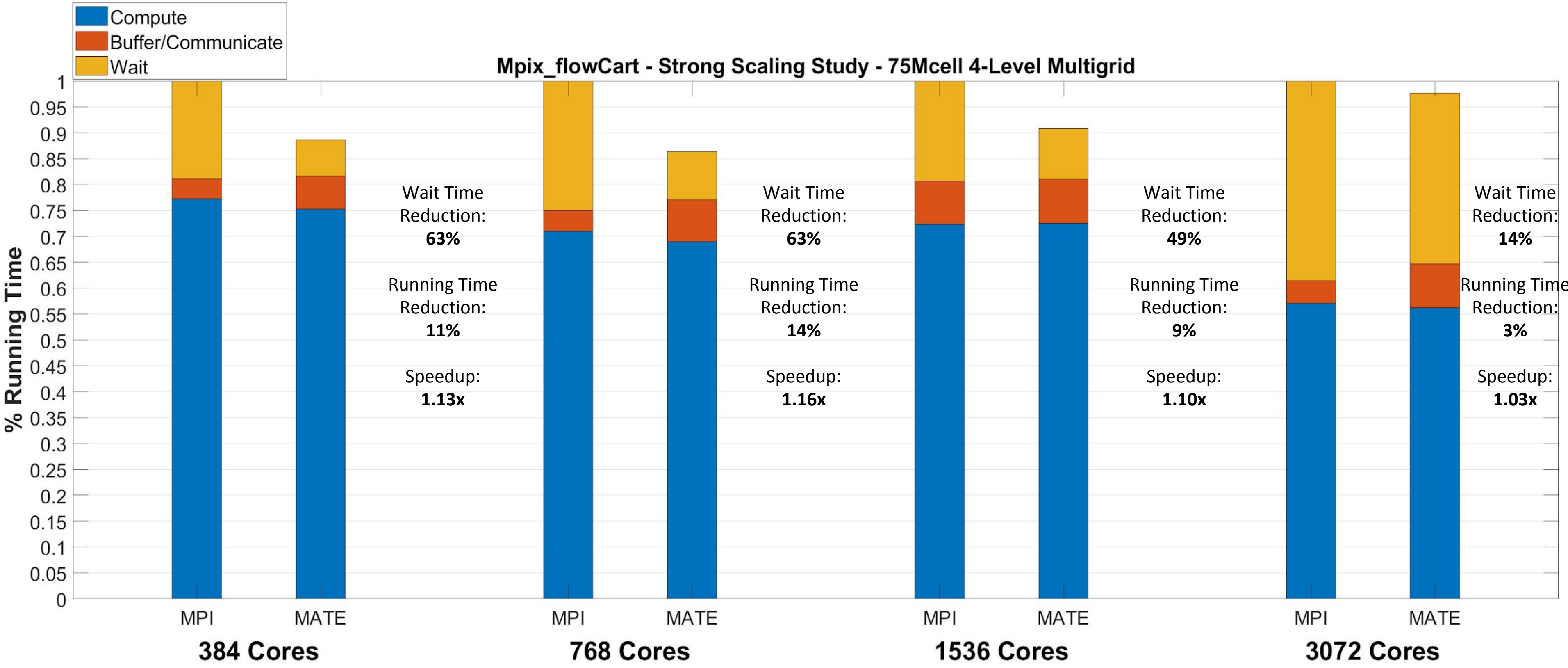


Source: https://www.nas.nasa.gov/publications/software/docs/cart3d/

Q&A

Contact:
sergiom@eng.ucsd.edu
mate.ucsd.edu

Edison Results - Strong Scaling



384 Cores

Wait Time Reduction: **63%**

Running Time Reduction: **11%**

Speedup: **1.13x**

768 Cores

Wait Time Reduction: **63%**

Running Time Reduction: **14%**

Speedup: **1.16x**

1536 Cores

Wait Time Reduction: **49%**

Running Time Reduction: **9%**

Speedup: **1.10x**

3072 Cores

Wait Time Reduction: **14%**

Running Time Reduction: **3%**

Speedup: **1.03x**