Translation and Runtime System-based Techniques for Hiding the Cost of Communication

Sergio Martin
University of California, San Diego
Berkeley Lab

Date: 8/8/2017

BERKELEY LAB

CRD

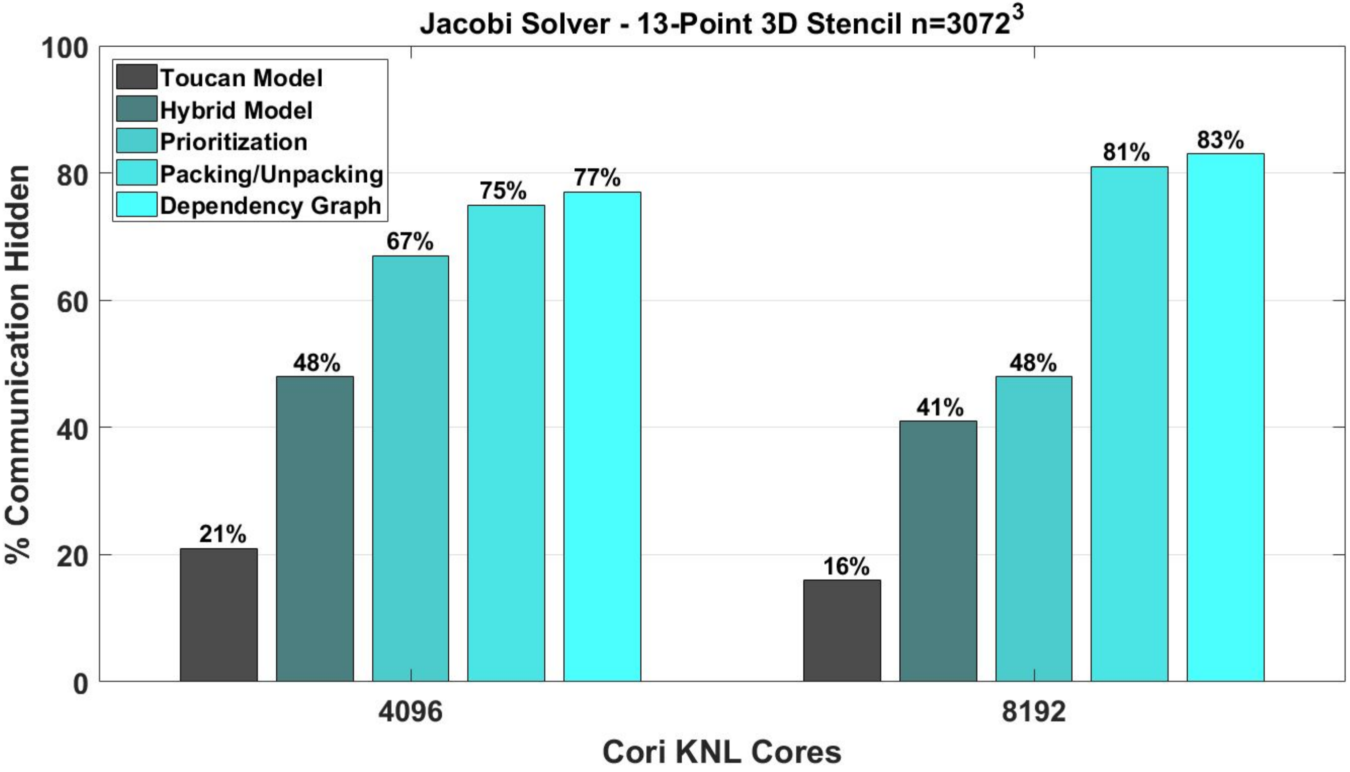
UCSDCSE
Computer Science and Engineering

Roadmap (2017)

Progression of the % of communication hidden.

Starting from January's version of our translator (21%) until today (83%).

Each bar represents an incremental optimization.



Motivation

- **Problem:** Communication costs are significant in large-scale parallel applications
    - Moreover, the overheads are continuing to grow towards the Exascale.
- **Coping strategies:**
    - Hiding Strategy: Overlap communication with computation
    - Avoiding Strategy: Performing less and/or more efficient communication

- **Shortfalls of implementing coping strategies manually:**
    - They may require algorithmic changes.
    - Entangles the coping strategy with the application logic.
    - For some large applications, these transformations are inviable.

Anatomy of a (Typical) MPI Program

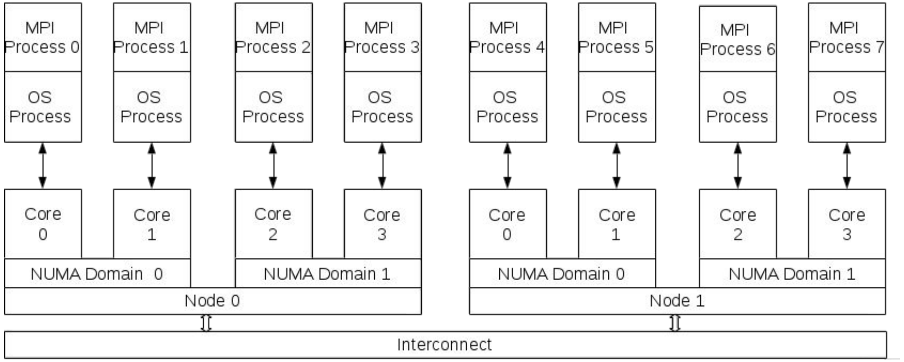
```
Begin

   Initialize Data

   Main Loop
    {
     … Receives …
     … Sends …
      -- Wait --
     … Compute …
    }

    Other Communication
    Output Result

End
```

Overlap Communication and Computation via Manual Transformation

```
Begin

   Initialize Data

   Main Loop
    {
     … Receives …
     … Sends …
     … Compute(Independent) …
         -- Wait --
     … Compute(Dependent) …
    }

   Other Communication
   Output Result

End
```

Granularity of MPI Programs

Observation I: MPI programs *typically* reach optimal performance when $P = c$, where c = number of cores.



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

$$G = N/P$$

Number of MPI Processes

Problem Size

Drawback: $P = c$ fixes granularity to the underlying architecture and does not allow *oversubscription*.

Toucan Model

Introducing **Toucan**

- **A Source-to-Source Translator of C/C++ MPI Applications.**
  - Automatically generates an overlapping version of the source code.
  - Built using the ROSE Compiler Framework (LLNL).
  - Uses Bamboo's[1] annotation scheme.

- **The translated code is linked to execute on our runtime system: MATE**.
  - MATE encapsulates all dynamic scheduling logic.
  - Avoids code bloating compared to Bamboo (static scheduling)
  - Supports recursive code.

MATE

(Pronounced 'Mah-tay')

- **Toucan/MATE rely on two mechanisms:**
  1. Oversubscription of processor cores.
  2. Splitting source code into code regions, scheduled individually.

---

[1] **"Bamboo - Translating MPI applications to a latency-tolerant, data-driven form"** Nguyen et al. In SC'12

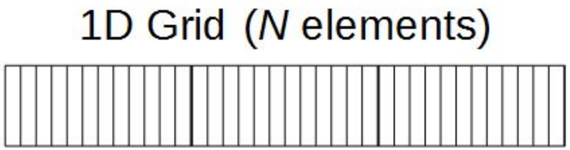
Definitions

- **Overdecomposition (SPMD):**
  - The problem domain is split into more partitions (Implies communication. i.e. ghost cells) than useful cores in the system.

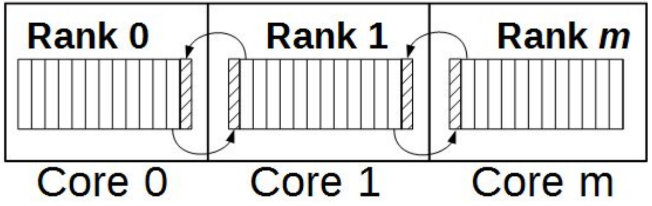
- **Oversubscription:**
  - Instantiating more autonomously executing tasks (ranks) than useful cores at all times during the execution.

- **Virtualization Factor:**
  - The integer multiplier by which oversubscription is achieved.
  - E.g. Instantiating 64 ranks in a 32-core run ⇒ **VF=2**

Toucan Decomposition

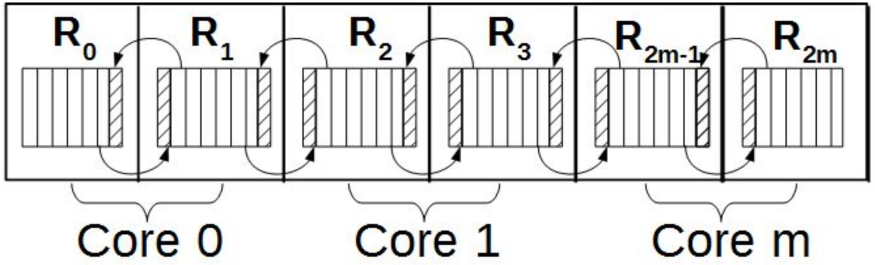


Typical Decomposition:
Virtualization Factor = 1
m Ranks, m Cores



N/m Elements per Rank
N/m Elements per Core

Toucan Decomposition:
Virtualization Factor = 2
2m Ranks, m Cores



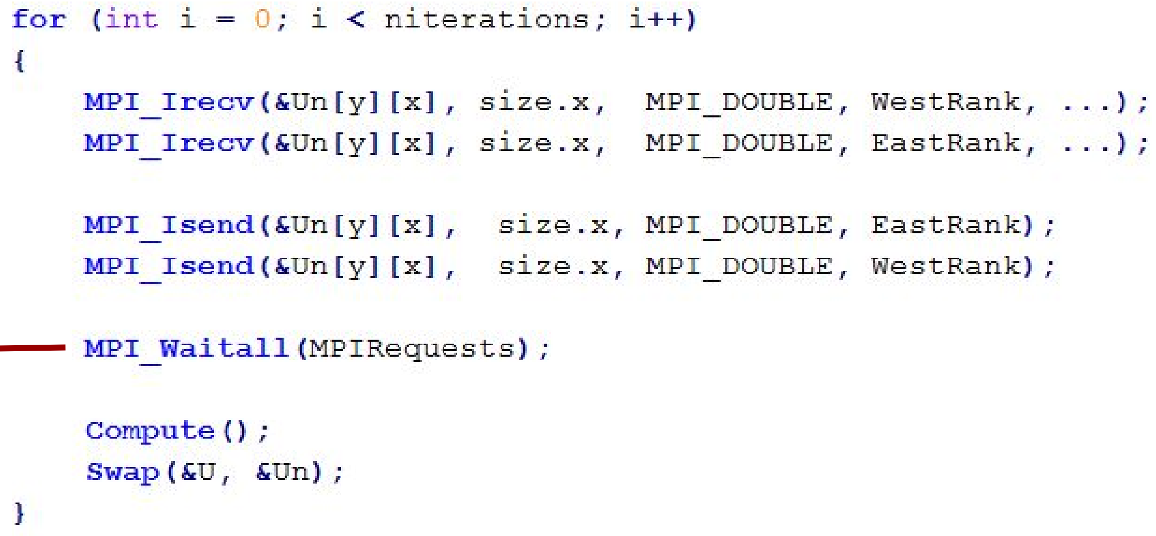
N/(2m) Elements per Rank
N/m Elements per Core

Toucan can only achieve **oversubscription** through **overdecomposition**.

Code Regions (1/2): A Basic MPI Code

All operations are waited for at MPI_Waitall().

1 Task Exit Point: **MPI_Waitall()**.



```
for (int i = 0; i < niterations; i++)
{
    MPI_Irecv(&Un[y][x], size.x,  MPI_DOUBLE, WestRank, ...);
    MPI_Irecv(&Un[y][x], size.x,  MPI_DOUBLE, EastRank, ...);

    MPI_Isend(&Un[y][x],  size.x, MPI_DOUBLE, EastRank);
    MPI_Isend(&Un[y][x],  size.x, MPI_DOUBLE, WestRank);

    MPI_Waitall(MPIRequests);

    Compute();
    Swap(&U, &Un);
}
```

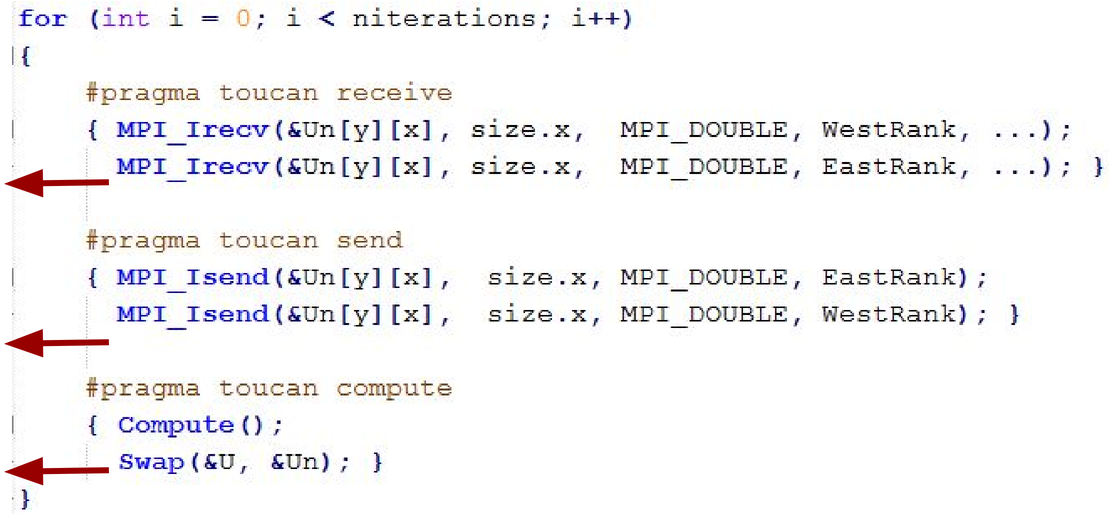
Code Regions (2/2): Using Toucan

3 pragma directives:
(Compute, Send, Receive)

Dependency Graph implicitly defined by the Bamboo/Toucan model:

- Receive depends on compute*
- Send depends on compute* and send*
- Compute depends on receive ops only

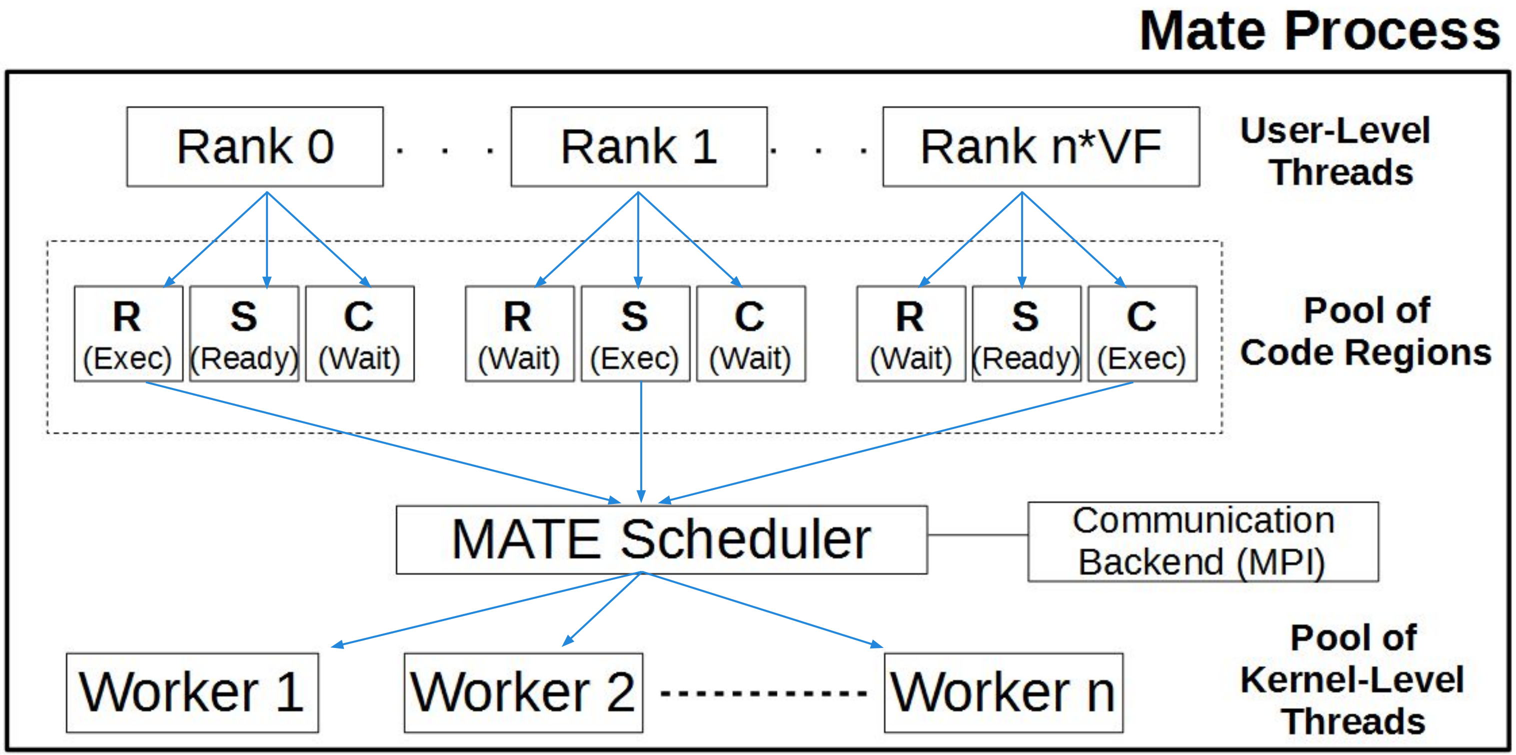
3 Task Exit Points:
One at the start/end of each region.



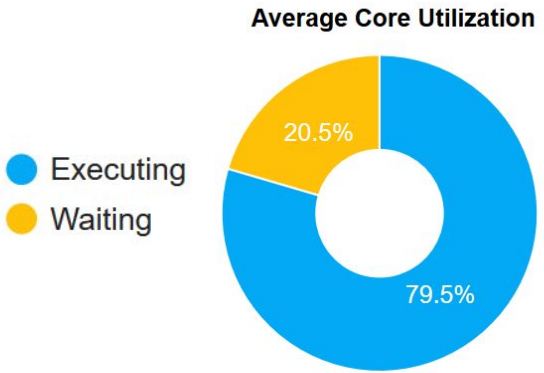
```
for (int i = 0; i < niterations; i++)
{
    #pragma toucan receive
    { MPI_Irecv(&Un[y][x], size.x,  MPI_DOUBLE, WestRank, ...);
      MPI_Irecv(&Un[y][x], size.x,  MPI_DOUBLE, EastRank, ...); }

    #pragma toucan send
    { MPI_Isend(&Un[y][x],  size.x, MPI_DOUBLE, EastRank);
      MPI_Isend(&Un[y][x],  size.x, MPI_DOUBLE, WestRank); }

    #pragma toucan compute
    { Compute();
      Swap(&U, &Un); }
}
```

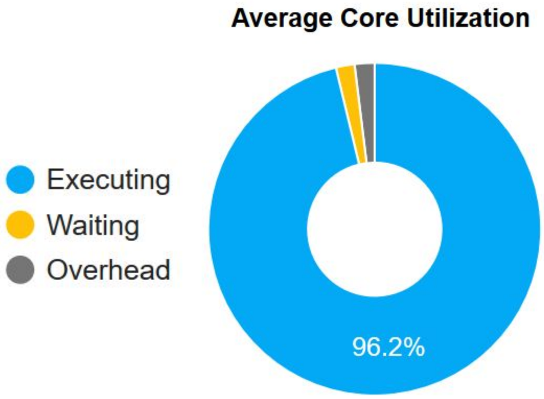
***From previous iteration**

Runtime System (MATE/Toucan)

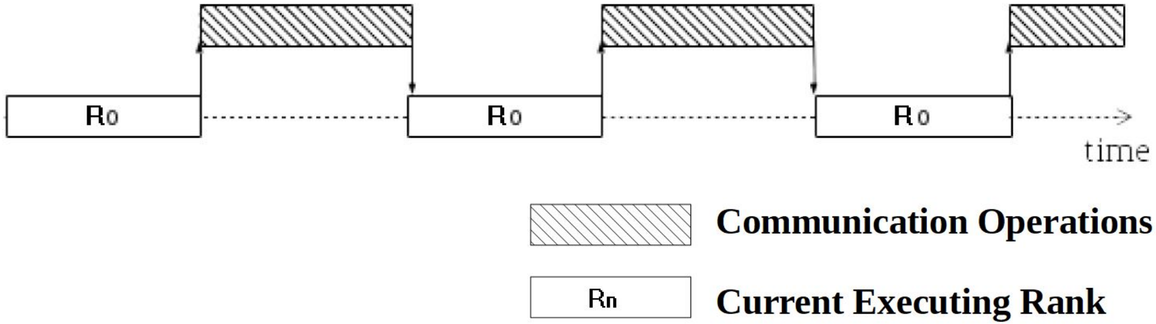


Core Usage Timeline

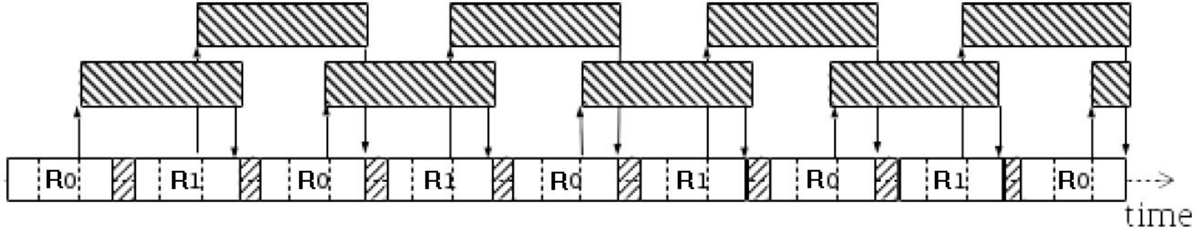




MPI Original - No Overlap



Toucan - VF=2 + Code Regions



Hardware Testbed: Cori KNL @ NERSC

Node Configuration:

- 1 x 68-core Intel KNL processor @ 1.4 Ghz (We only use 64 cores per Node)

Memory:

- 16 GB MCDRAM ~ 460 GB/s
- 96 GB DDR4 ~102 GB/s

Software:

- Cray-MPICH v7.6.0
- Intel icc compiler 17.0.2 (-O3)

Test Case:

- 13-Point Stencil 3D Jacobi Solver
- 2 Control Variants:
  - MPI Original
  - Upper Bound

MPI Original without communication,
just synchronization

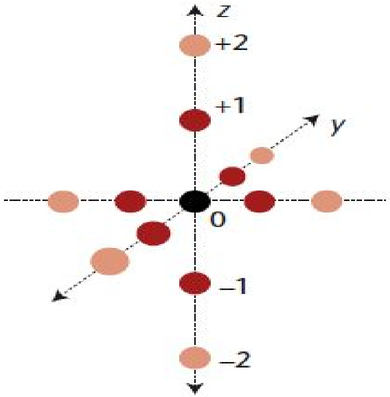


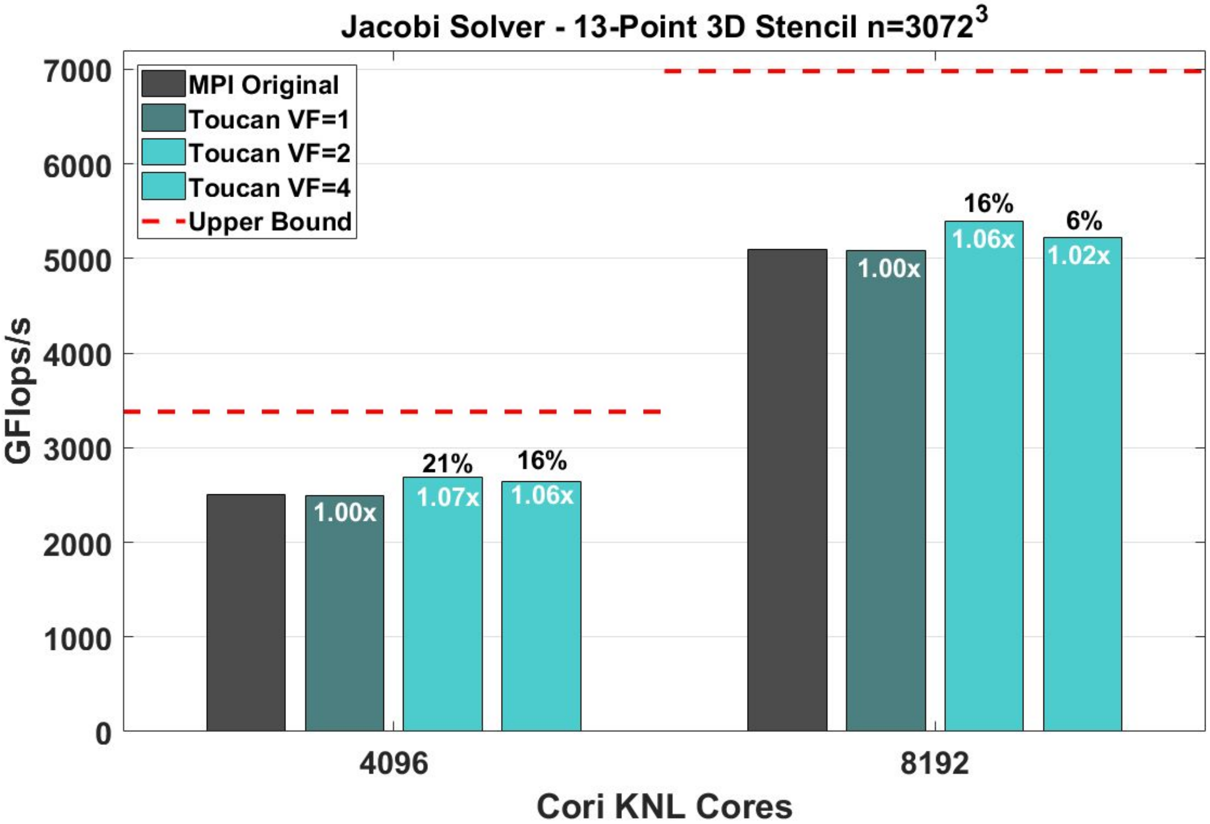
Image Source:
"Accelerating a 3D Finite-Difference Earthquake Simulation with a C-to-CUDA Translator", Cai et al.

Results: Toucan Variants

VF=1
Does not allow any overlap since cores cannot find ready ranks while other communicate.

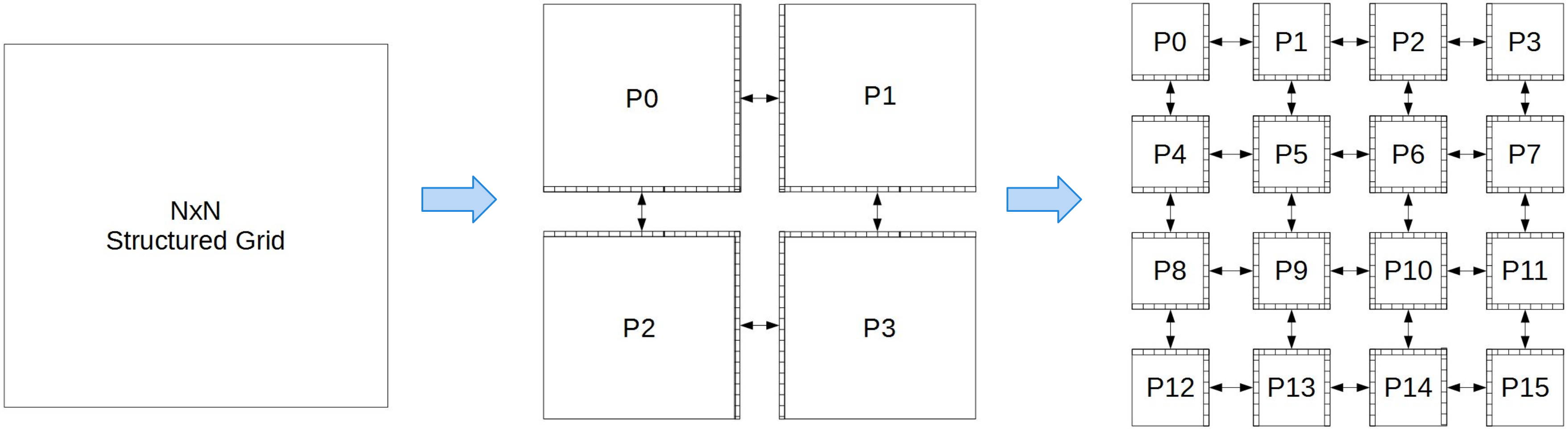
VF=2
Achieves optimal performance in both cases.

VF=4
Loses performance due to the **surface/volume ratio problem.**



Surface/Volume Ratio Problem

- Overdecomposition in Toucan requires communicating extra ghost cells.
- Suffers from increased in-node communication.



Typical MPI Decomposition
Virtualization Factor = 1

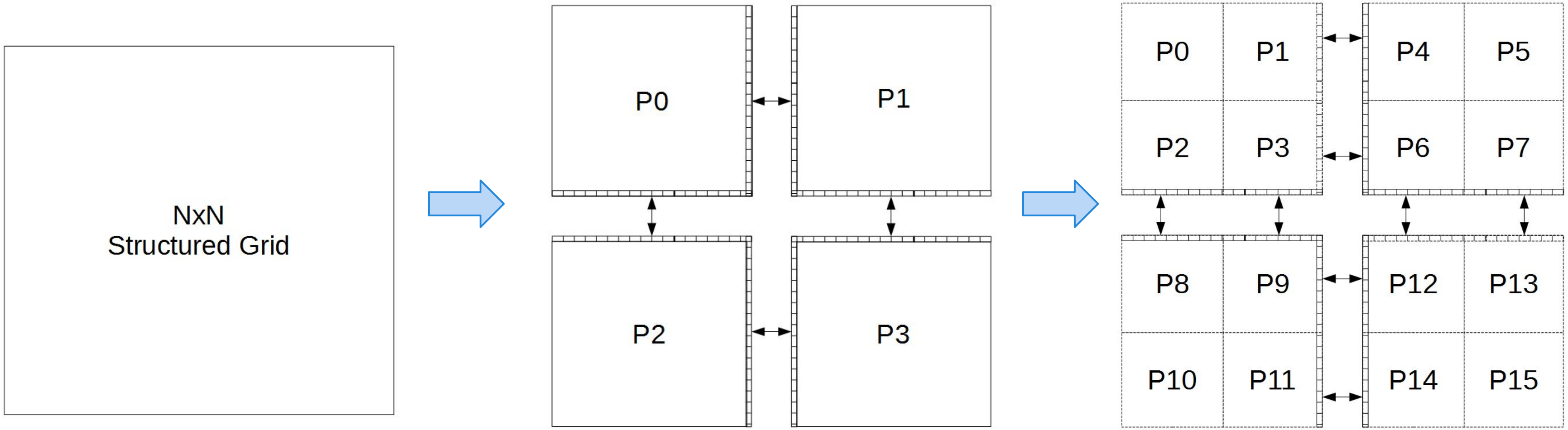
Toucan Overdecomposition
Virtualization Factor = 4

There is an excess of in-node communication: **1.30x** more, in our Edison experiments.

Hybrid Model

MATE Hybrid Model

- Problem domain is not overdecomposed.
- Instead, oversubscribed tasks share the same process-wide subdomain.

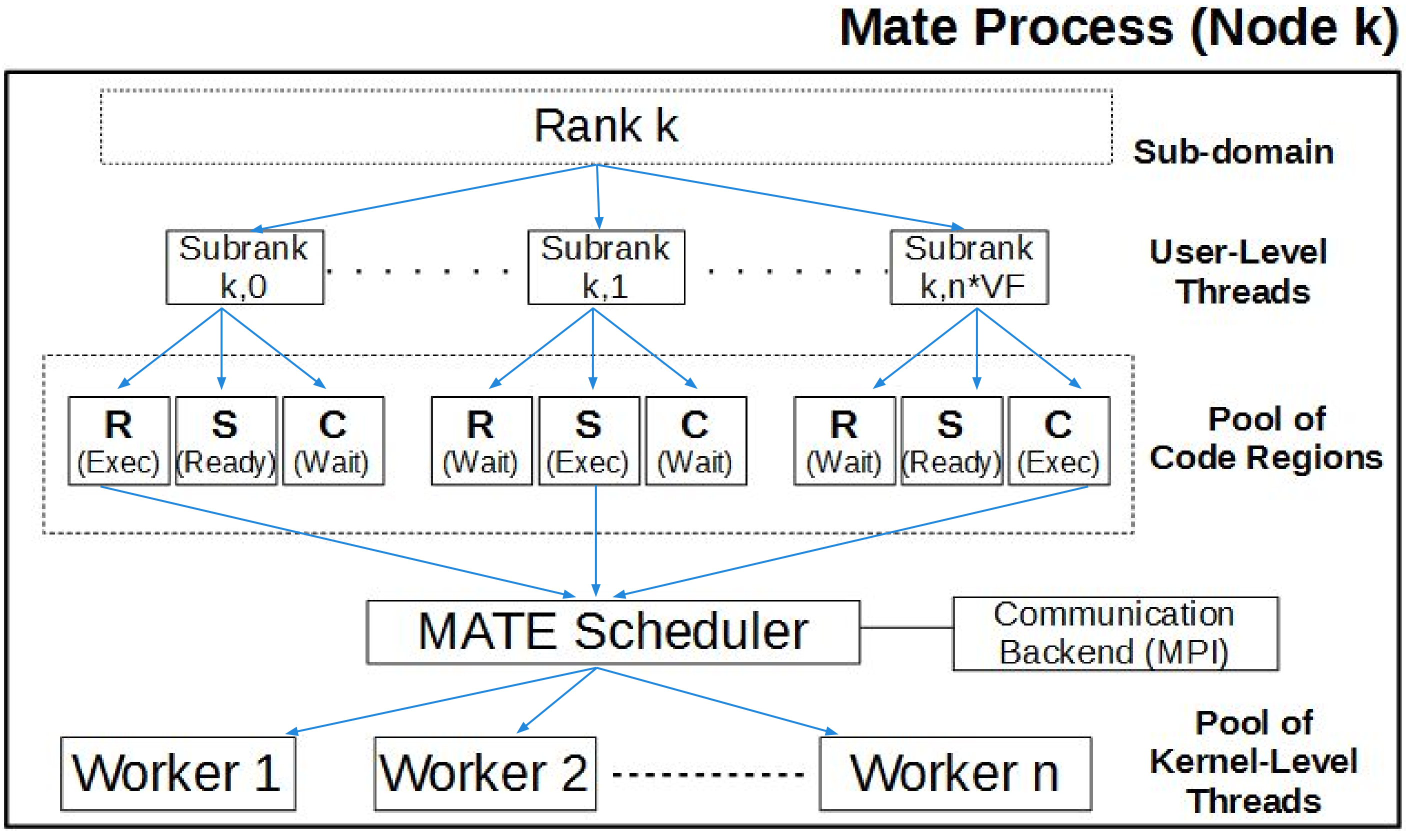


Typical MPI Decomposition
Virtualization Factor = 1

H-Model (No Overdecomposition)
Virtualization Factor = 4

Synchronization is required for tasks in the same subdomain only.

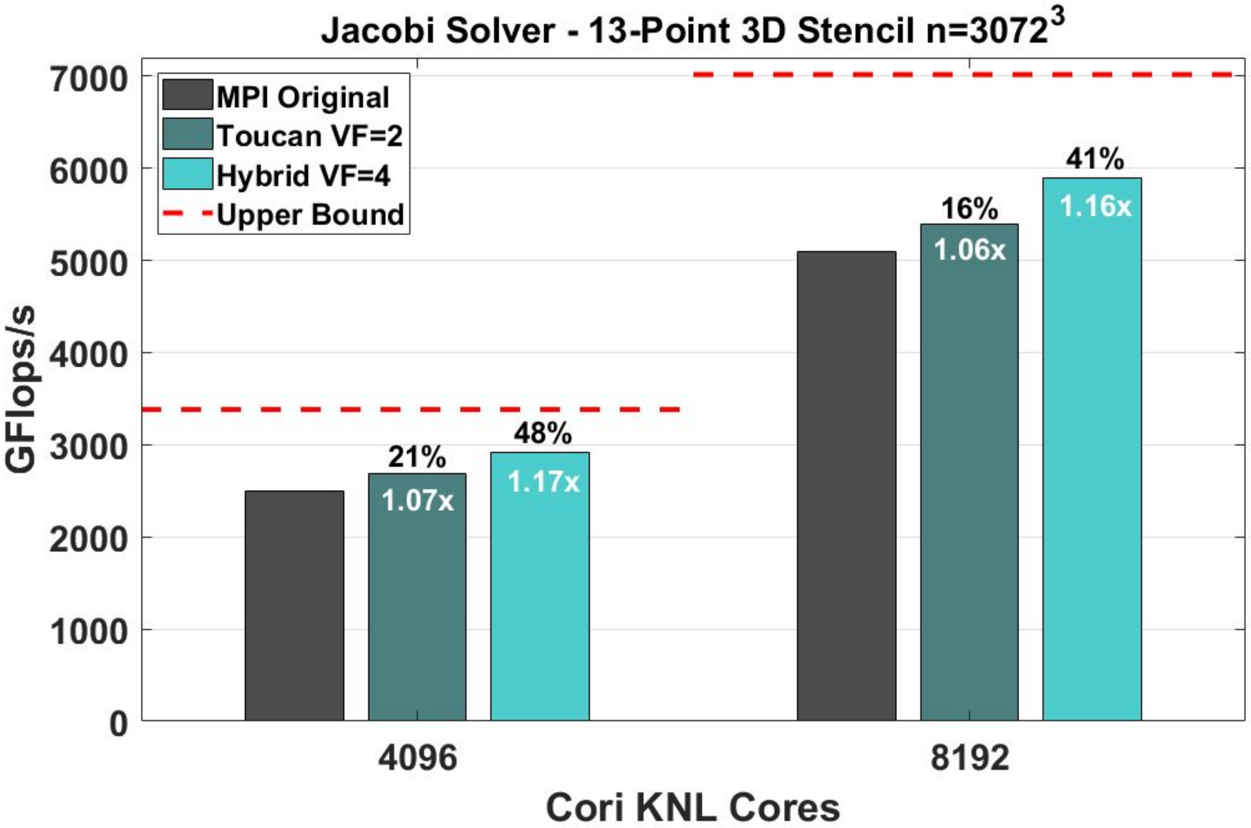
Runtime System (MATE/Hybrid)



Results: Hybrid vs Toucan

Since the Hybrid model does not increase the amount of communication inside a node, **higher Virtualization Factors can be used**.

In this case, **VF=4** is the ideal setting for the Hybrid Variant, achieving more than double the overlap yield of the Toucan variant.

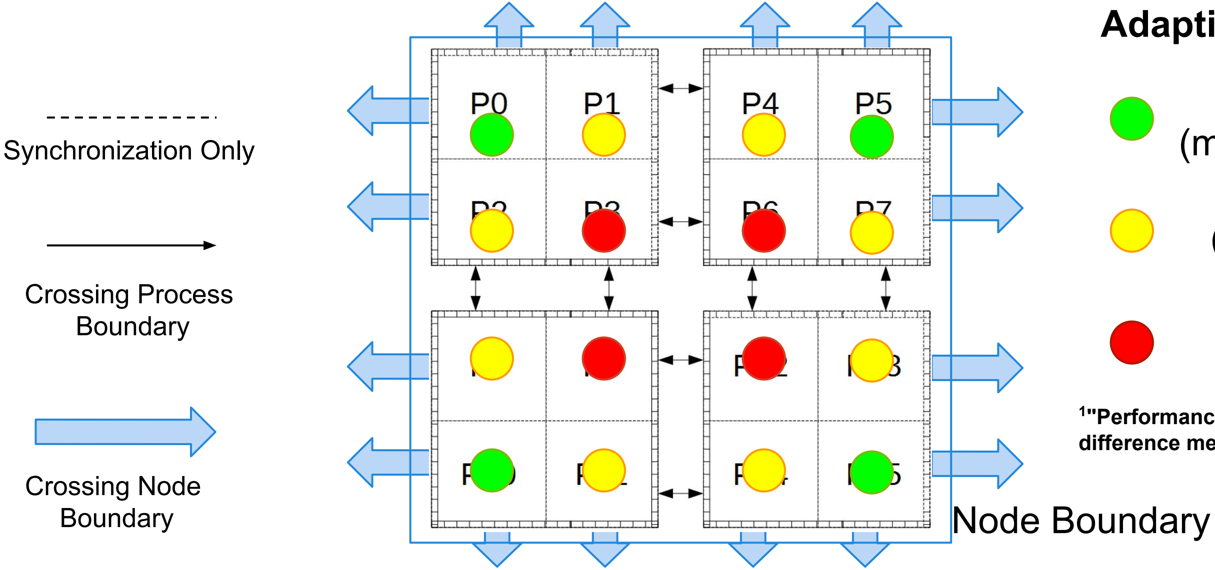


Optimization #1:

Subrank Prioritization

Subrank Prioritization

- 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.



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.

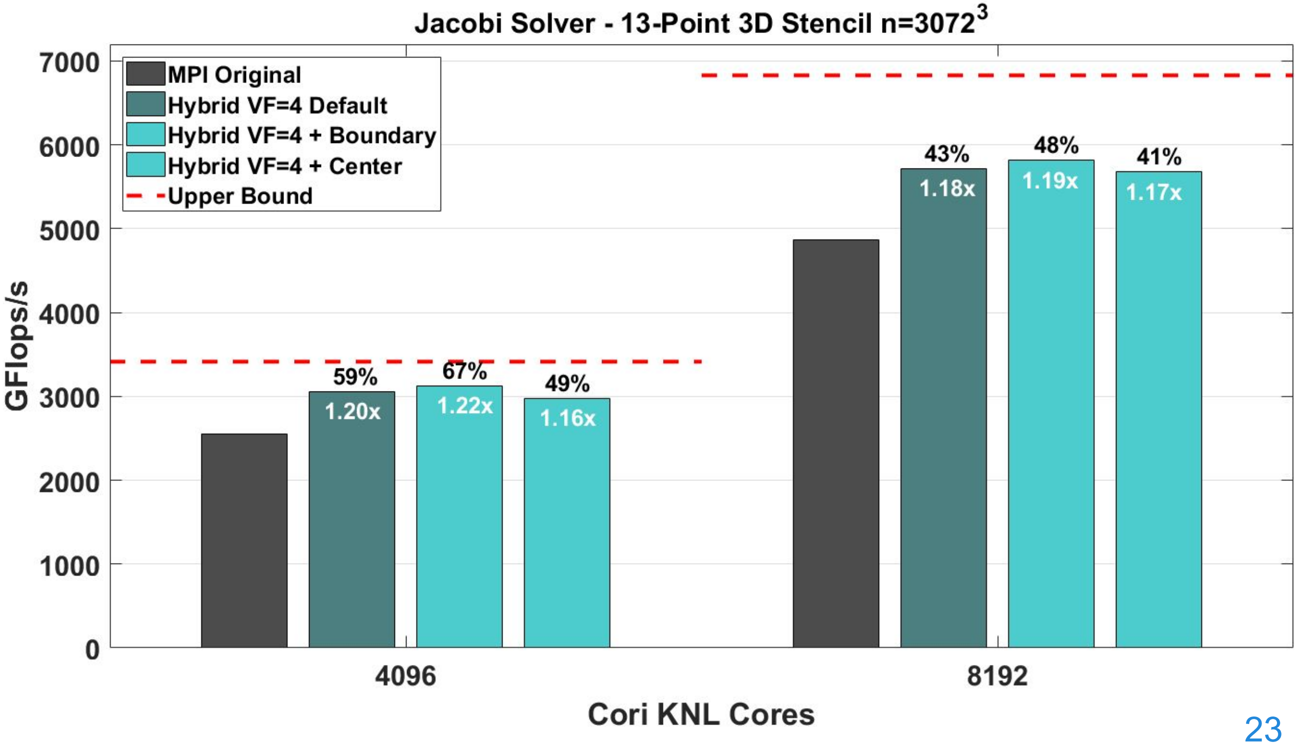
Results: Prioritization

We tested 3 prioritization schemes:

Hybrid VF=4 Default
No priority scheme. Execution order is mostly random.

Hybrid VF=4 + Boundary
Priority scheme as described in the previous slide.

Hybrid VF=4 + Center
The opposite scheme than the one described in the previous slide.



Optimization #2:

Thread Concurrency

Thread Concurrency Problem

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

`MPI_Isend(strided_type)`

1) Pack strided data into a hidden contiguous buffer.
2) Transmit buffered data to destination rank.

MPI_Lock

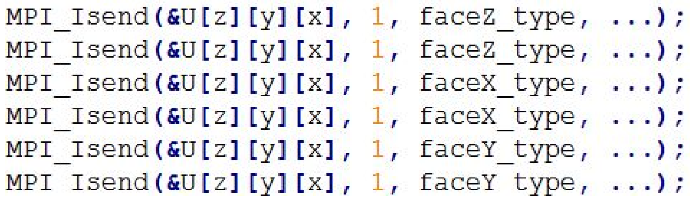
`MPI_Irecv(strided_type)`

1) Receive incoming data into a hidden contiguous buffer.
2) Unpack data into destination with strides.

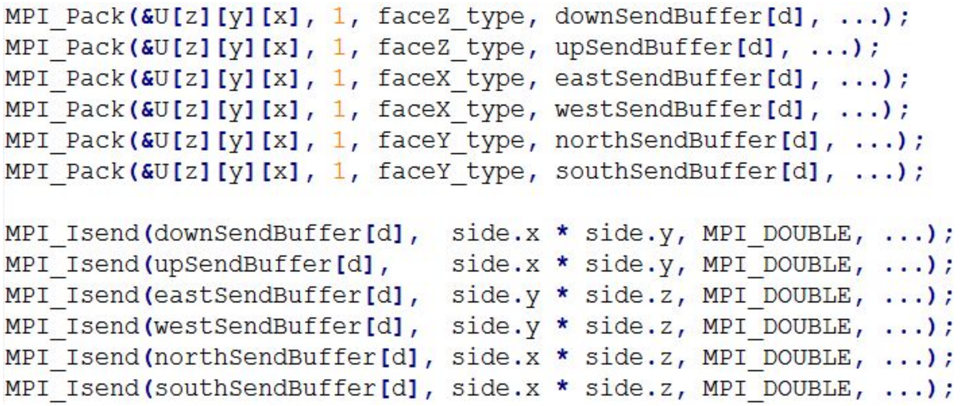
MPI_Lock

Thread Concurrency Problem

- **A way to cope with this problem is to split MPI operations:**
  - MPI_Isend ⇒ MPI_Pack + MPI_Isend (contiguous)
  - MPI_Irecv ⇒ MPI_Irecv (contiguous) + MPI_Unpack



```
MPI_Isend(&U[z][y][x], 1, faceZ_type, ...);
MPI_Isend(&U[z][y][x], 1, faceZ_type, ...);
MPI_Isend(&U[z][y][x], 1, faceX_type, ...);
MPI_Isend(&U[z][y][x], 1, faceX_type, ...);
MPI_Isend(&U[z][y][x], 1, faceY_type, ...);
MPI_Isend(&U[z][y][x], 1, faceY_type, ...);
```



```
MPI_Pack(&U[z][y][x], 1, faceZ_type, downSendBuffer[d], ...);
MPI_Pack(&U[z][y][x], 1, faceZ_type, upSendBuffer[d], ...);
MPI_Pack(&U[z][y][x], 1, faceX_type, eastSendBuffer[d], ...);
MPI_Pack(&U[z][y][x], 1, faceX_type, westSendBuffer[d], ...);
MPI_Pack(&U[z][y][x], 1, faceY_type, northSendBuffer[d], ...);
MPI_Pack(&U[z][y][x], 1, faceY_type, southSendBuffer[d], ...);

MPI_Isend(downSendBuffer[d],  side.x * side.y, MPI_DOUBLE, ...);
MPI_Isend(upSendBuffer[d],      side.x * side.y, MPI_DOUBLE, ...);
MPI_Isend(eastSendBuffer[d],  side.y * side.z, MPI_DOUBLE, ...);
MPI_Isend(westSendBuffer[d],  side.y * side.z, MPI_DOUBLE, ...);
MPI_Isend(northSendBuffer[d], side.x * side.z, MPI_DOUBLE, ...);
MPI_Isend(southSendBuffer[d], side.x * side.z, MPI_DOUBLE, ...);
```

Thread Concurrency Problem

After splitting the operations, threads can perform concurrent packing/unpacking.

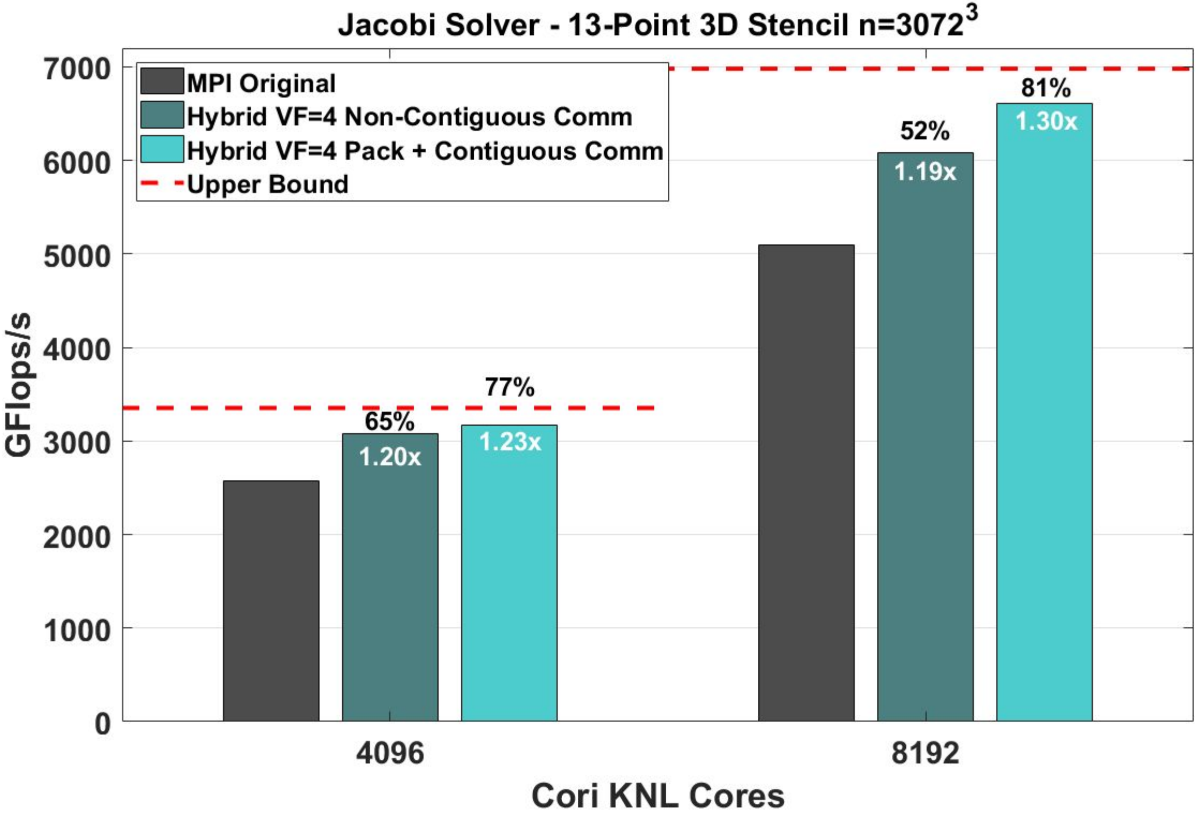
`MPI_Pack(strided_type)`	Pack strided data into an explicit contiguous buffer.	Thread Safe
`MPI_Isend(contiguous)`	Transmit buffered data to destination rank.	MPI_Lock
`MPI_Irecv(contiguous)`	Receive incoming data into an explicit contiguous buffer.	MPI_Lock
`MPI_Unpack(strided_type)`	Unpack data into destination with strides.	Thread Safe

Results: Contiguous vs Non-Contiguous

Comparing 2 Variants:

Hybrid VF=4 + Non-Contiguous Communication

Hybrid VF=4 + Packing/Unpacking + Contiguous Communication



Optimization #3:
Dependency Graph Refinement

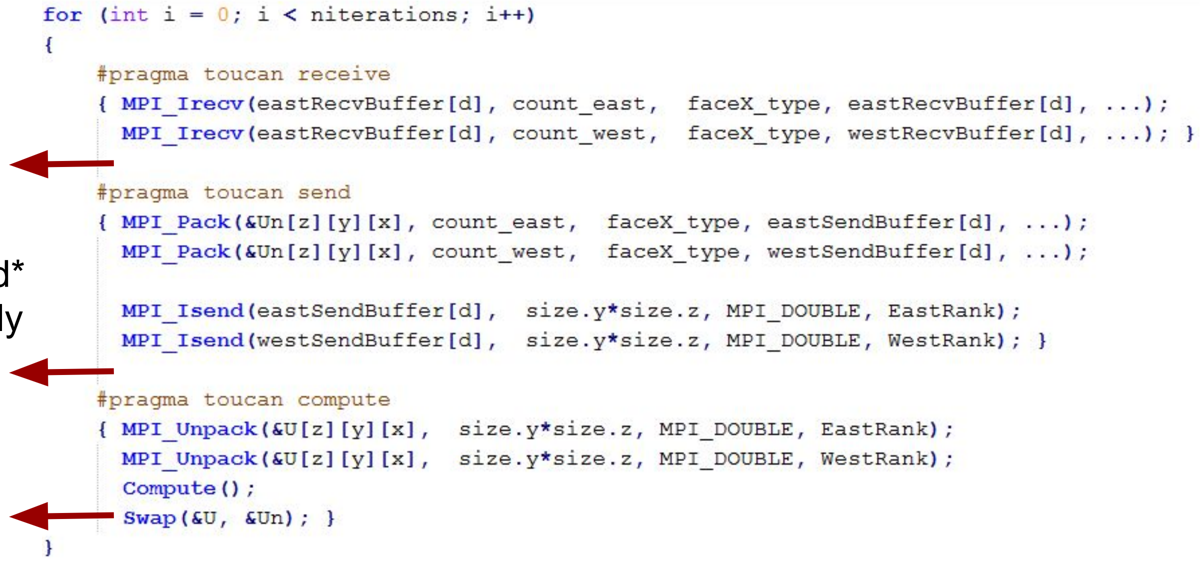
Toucan-Annotated Program

3 pragma annotations:
(Compute, Send, Receive)

Dependency Graph implicitly defined

- Receive depends on compute*
- Send depends on compute* and send*
- Compute depends on receive ops only

3 Task Exit Points.



```
for (int i = 0; i < niterations; i++)
{
    #pragma toucan receive
    { MPI_Irecv(eastRecvBuffer[d], count_east,  faceX_type, eastRecvBuffer[d], ...);
      MPI_Irecv(eastRecvBuffer[d], count_west,  faceX_type, westRecvBuffer[d], ...); }

    #pragma toucan send
    { MPI_Pack(&Un[z][y][x], count_east,  faceX_type, eastSendBuffer[d], ...);
      MPI_Pack(&Un[z][y][x], count_west,  faceX_type, westSendBuffer[d], ...);

      MPI_Isend(eastSendBuffer[d],  size.y*size.z, MPI_DOUBLE, EastRank);
      MPI_Isend(westSendBuffer[d],  size.y*size.z, MPI_DOUBLE, WestRank); }

    #pragma toucan compute
    { MPI_Unpack(&U[z][y][x],  size.y*size.z, MPI_DOUBLE, EastRank);
      MPI_Unpack(&U[z][y][x],  size.y*size.z, MPI_DOUBLE, WestRank);
      Compute();
      Swap(&U, &Un); }
}
```

From previous iteration

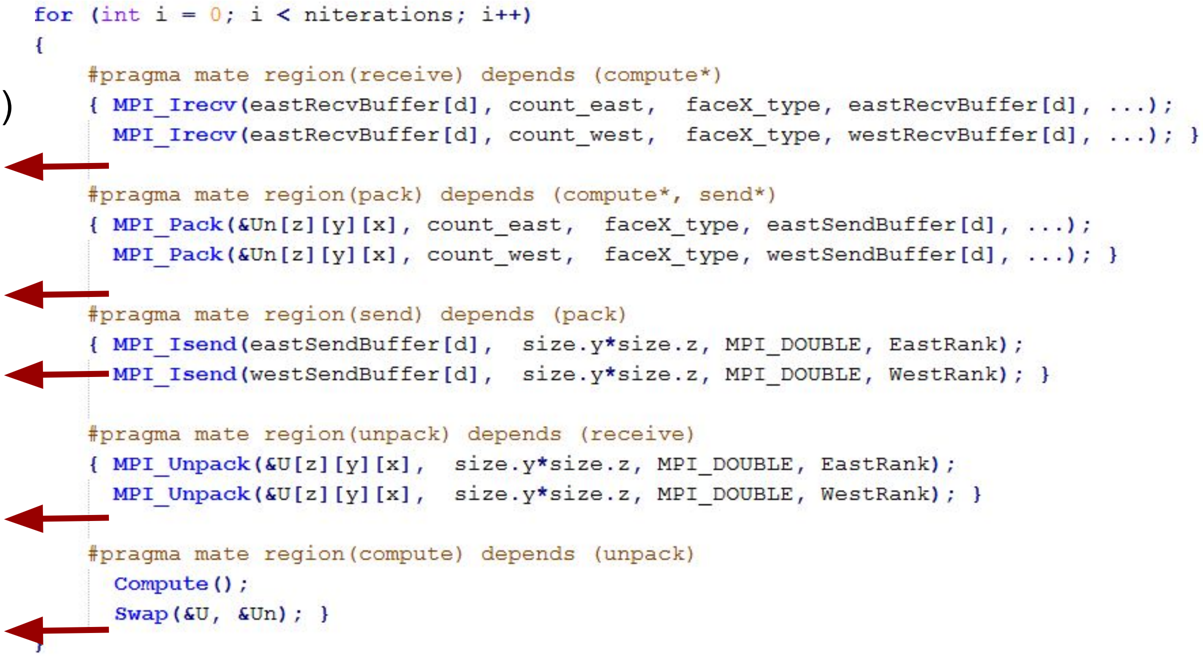
Explicit Graph in MATE

5 pragma annotations:
(Compute, Pack, Send, Receive, Unpack)

Dependency Graph explicitly defined

- Receive depends on Unpack*
- Pack depends on Compute*, Send*
- Send depends on Pack
- Unpack depends on Receive
- Compute depends on Unpack

5 Task Exit Points.



```
for (int i = 0; i < niterations; i++)
{
    #pragma mate region(receive) depends (compute*)
    { MPI_Irecv(eastRecvBuffer[d], count_east,  faceX_type, eastRecvBuffer[d], ...);
      MPI_Irecv(eastRecvBuffer[d], count_west,  faceX_type, westRecvBuffer[d], ...); }

    #pragma mate region(pack) depends (compute*, send*)
    { MPI_Pack(&Un[z][y][x], count_east,  faceX_type, eastSendBuffer[d], ...);
      MPI_Pack(&Un[z][y][x], count_west,  faceX_type, westSendBuffer[d], ...); }

    #pragma mate region(send) depends (pack)
    { MPI_Isend(eastSendBuffer[d],  size.y*size.z, MPI_DOUBLE, EastRank);
      MPI_Isend(westSendBuffer[d],  size.y*size.z, MPI_DOUBLE, WestRank); }

    #pragma mate region(unpack) depends (receive)
    { MPI_Unpack(&U[z][y][x],  size.y*size.z, MPI_DOUBLE, EastRank);
      MPI_Unpack(&U[z][y][x],  size.y*size.z, MPI_DOUBLE, WestRank); }

    #pragma mate region(compute) depends (unpack)
    { Compute();
      Swap(&U, &Un); }
}
```

From previous iteration

Directional Annotations

9 pragma annotations:
(Pack, Send, Receive, Unpack) x 2
Compute

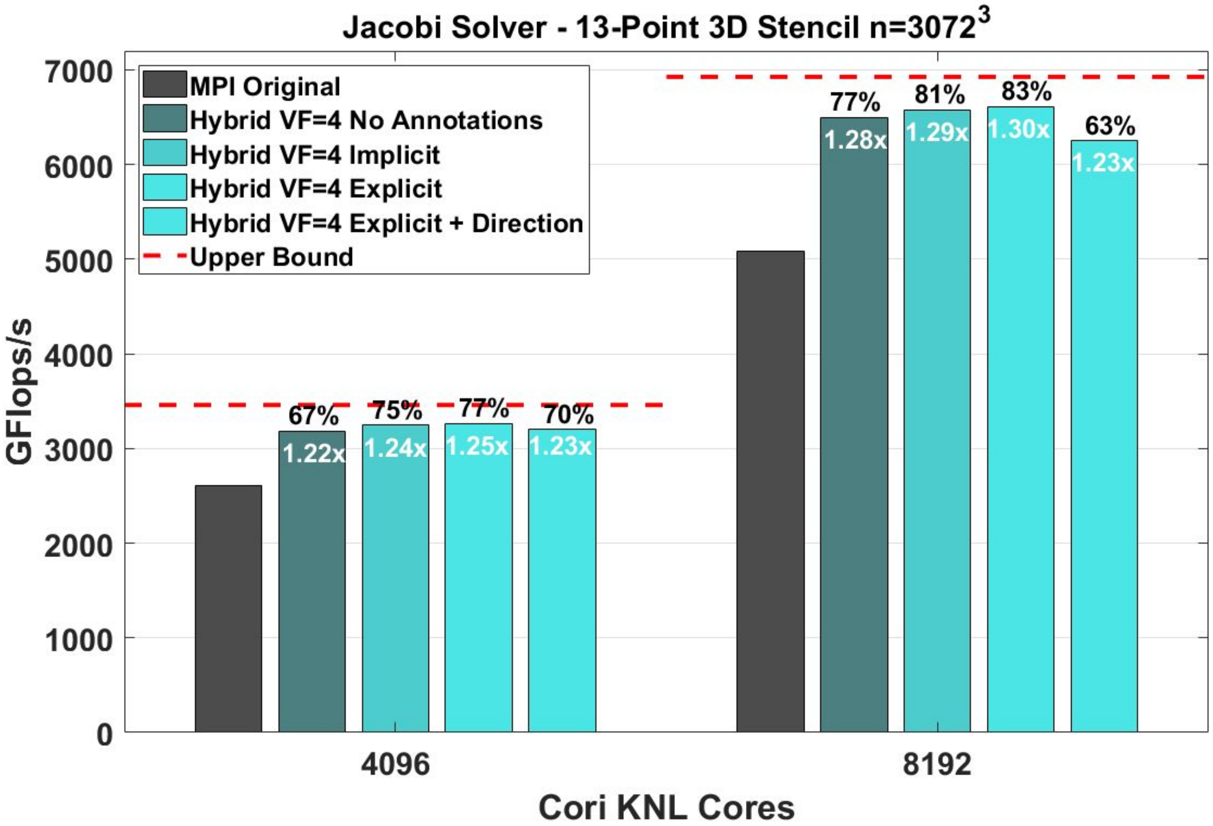
Dependency Graph explicitly defined
All regions and dependencies are split into their specific directions

9 Task Exit Points.

**From previous iteration*

```
for (int i = 0; i < niterations; i++)
{
  #pragma mate region(receive_east) depends (compute*)
   MPI_Irecv(eastRecvBuffer[d], count_east,  faceX_type, eastRecvBuffer[d], ...);

  #pragma mate region(receive_west) depends (compute*)
   MPI_Irecv(eastRecvBuffer[d], count_west,  faceX_type, westRecvBuffer[d], ...);

  #pragma mate region(pack_east) depends (compute*, send_east*)
   MPI_Pack(&Un[z][y][x], count_east,  faceX_type, eastSendBuffer[d], ...);

  #pragma mate region(pack_west) depends (compute*, send_west*)
   MPI_Pack(&Un[z][y][x], count_west,  faceX_type, westSendBuffer[d], ...);

  #pragma mate region(send_west) depends (pack_east)
   MPI_Isend(eastSendBuffer[d],  size.y*size.z, MPI_DOUBLE, EastRank);

  #pragma mate region(send_east) depends (pack_west)
   MPI_Isend(westSendBuffer[d],  size.y*size.z, MPI_DOUBLE, WestRank);

  #pragma mate region(unpack_east) depends (receive_east)
   MPI_Unpack(&U[z][y][x],  size.y*size.z, MPI_DOUBLE, EastRank);

  #pragma mate region(unpack_west) depends (receive_west)
   MPI_Unpack(&U[z][y][x],  size.y*size.z, MPI_DOUBLE, WestRank);

  #pragma mate region(compute) depends (unpack_east, unpack_west)
  { Compute();
   Swap(&U, &Un); }
}
```

Results: Dependency Graph Variants

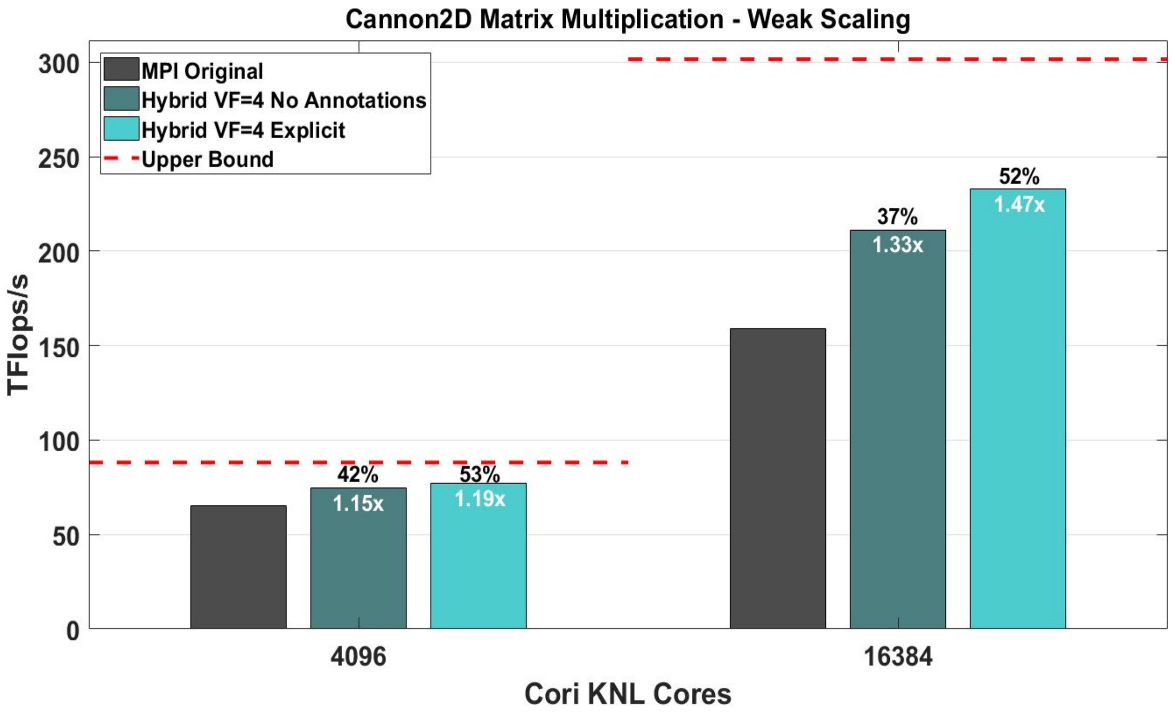


Results: Matrix Multiplication

Results for the Cannon2D Matrix Multiply algorithm

Comparing variants with and without annotations.

The variant with an explicitly defined graph hides 11-15% more of the communication cost.



Conclusions

Conclusions

- **The Hybrid Model exceeds the efficiency of the Toucan Model**
  - Hides communication by Oversubscription+Regions (like Toucan) but,
  - It does not require overdecomposition (zero added communication).

- **Subrank prioritization can have a substantial impact on performance**
  - Communication cost is not homogeneous, thus creating opportunities for further communication/computation overlap.
  - The MATE runtime can assign priorities automatically during execution.

- **Thread concurrency is still an important issue to be solved**
  - Even if packing can be performed concurrently, MPI still locks comm ops.

- **It is possible to refine dependency graphs explicitly**

Future Work

- **Apply the Hybrid Model on real-world applications / benchmarks**
  - E.g. Mpix_FlowCart (>20k LoC)

- **Replace MATE's communication layer.**
  - Use a thread-tolerant communication layer (e.g. GASNetEx, UPC++)
  - While keeping the current MPI-2 programming interface.

- **Use parallel profiling tools to examine the low-level effects of our models**
  - HPCToolkit

- **Explore the effect of refining code regions and dependencies**
  - Their impact on performance needs to be investigated further.

Questions?

For an explanation of Toucan's rationale and experimental results on the Edison supercomputer, check our IPDPS'17 paper:

"Toucan - A Translator for Communication Tolerant MPI Applications"
S. Martin, M. J. Berger, and S. B. Baden