A Hybrid Model for Reducing the Cost of Communication in Large-Scale Applications

Sergio Martin
Department of Computer Science and Engineering
University of California, San Diego
Date: 10/02/2017

Committee:
Prof. Scott Baden, Chair (CSE)
Prof. George Porter, Co-Chair (CSE)
Prof. Tajana Rosing (CSE)
Prof. Sutanu Sarkar (MAE)
Prof. John Weare (Chem & BioChem)
High-Performance Computing

HPC is an essential tool in developments in science and technology.
High-Performance Computing

Relies on the power of Supercomputers
(Hardware + Interconnect)

Requires a Parallel Programming Framework

- Parallel Programming Models
- Communication Libraries
- Runtime Systems
- Threading Libraries
- Compilers / Translators

That's me @ Berkeley Lab
Evolution of Supercomputers

I. Introduction
II. Related Work
III. Refactoring
IV. Toucan
V. Hybrid
VI. Remaining Work

“The mission and science opportunities in going to exascale are compelling”

Example: Design of advanced materials.
Challenges of Exascale Computing

3 Main Challenges¹:

- Reduce Energy Consumption (From 200MW estimated to 20MW²).
- Ensure Reliability and Fault-Tolerance.
- Exploit Massive Parallelism.
  - Provide an adaptive response to load imbalance.
  - Develop multi-core and memory hierarchy-aware algorithms.
  - Reduce the cost of communication.

Communication cost comprises a significant part of large-scale application running time\(^1\).

(Moreover, communication overheads are continuing to grow towards the *Exascale*.)

*For this reason*...

\(^{1}\)“[...] There is a need to investigate algorithms that reduce communication to a minimum.”\(^2\)

\(^{2}\)“The opportunities and challenges of exascale computing”, S. Ashby et al, Summary Report of the US DOE ASCR, 2010
\(^{1}\)“Communication Avoiding and Overlapping for Numerical Linear Algebra”, E. Georganas et al, SC12, 2012
<table>
<thead>
<tr>
<th>Models</th>
<th>Variants</th>
<th>Languages/Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threading Model</td>
<td>Kernel Threads</td>
<td>POSIX Threads, OpenMP</td>
</tr>
<tr>
<td>Message Passing Model</td>
<td>Flat MPI</td>
<td>MPI</td>
</tr>
<tr>
<td></td>
<td>Fine-Grained MPI</td>
<td>FP-MPI, TMPI, AzequiaMPI</td>
</tr>
<tr>
<td></td>
<td>MPI+X</td>
<td>MPI+OpenMP, MPI+PThreads, MPI+MPI</td>
</tr>
<tr>
<td>Dataflow Models</td>
<td>Concurrent Collections</td>
<td>Habanero CnC, DAGuE</td>
</tr>
<tr>
<td></td>
<td>Statement-Level Dataflow</td>
<td>Tarragon, SMPSs</td>
</tr>
<tr>
<td></td>
<td>Task-Level Dataflow</td>
<td></td>
</tr>
</tbody>
</table>

- Based on shared memory.
- Limited to a single node.
- Based on Message Passing.
- Enables inter-node communication.
Anatomy of a naive MPI Application

Problem: Naive MPI applications suffer from the full cost of communication.

Coping strategies:
- Hiding Strategy: Overlap communication with computation\(^1,2\).
- 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
For (iterations)
{
    ... Receive Requests...
    ... Send Requests...
    -- Wait for Requests --
    ... Compute ...
}

Manually decompose compute section into separate dependent/independent sections.

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

Shortfalls of re-factoring MPI applications manually:
- Embeds policy decisions into the application code.
- They may require non-trivial algorithmic changes.
- Transformations are hard to maintain (architecture-dependent).
- For some large applications, these transformations are unviable.
Communication cost comprises a significant part of large-scale application running time\textsuperscript{1}.

(Moreover, communication overheads are continuing to grow towards the \textit{Exascale}.)

\textit{For this reason}...

\textsuperscript{1}“[…] There is a need to investigate algorithms that reduce communication to a minimum.”\textsuperscript{2}

\textit{However}...

\textsuperscript{2}\textit{The opportunities and challenges of exascale computing}, S. Ashby et al, Summary Report of the US DOE ASCR, 2010

\textsuperscript{1}\textit{Communication Avoiding and Overlapping for Numerical Linear Algebra}, E. Georganas et al, SC12, 2012
## Alternative Parallel Programming Models

<table>
<thead>
<tr>
<th>Models</th>
<th>Variants</th>
<th>Languages/Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threading Model</td>
<td>Kernel Threads</td>
<td>POSIX Threads, OpenMP</td>
</tr>
<tr>
<td>Flat MPI</td>
<td></td>
<td>MPI</td>
</tr>
<tr>
<td>Fine-Grained MPI</td>
<td></td>
<td>FP-MPI, TMPI, AzequiaMPI</td>
</tr>
<tr>
<td>MPI+X</td>
<td></td>
<td>MPI+OpenMP, MPI+PThreads, MPI+MPI</td>
</tr>
<tr>
<td>Concurrent Collections</td>
<td></td>
<td>Habanero-CnC, DAGuE</td>
</tr>
<tr>
<td>Statement-Level Dataflow</td>
<td></td>
<td>SwiftT</td>
</tr>
<tr>
<td>Task-Level Dataflow</td>
<td></td>
<td>Tarragon, SMPSs</td>
</tr>
</tbody>
</table>

- Simple, intuitive, easy to use.
- Most widely used. Plenty of Legacy code.
- Hard to optimize for hiding communication cost.

- Data-dependency flow of execution.
- (Arguably) Less intuitive.
- Better suited to design communication-tolerant applications

Is automatic conversion possible?
Automatic Translation

Alternative approach to manual re-factoring:

- Use translation-based tools to achieve communication/computation overlap\(^1,2\).
- Idea first proposed by the authors of the **Bamboo Model**\(^3\).
- Convert a traditional MPI program into a dataflow-model program automatically.
- The semantics of the source code remain unaltered.

---


\(^2\)"Petal Tool for Analyzing and Transforming Legacy MPI Applications", H Ahmed et al. In: LCPC ‘15

\(^3\)"Bamboo - Translating MPI applications to a latency-tolerant, data-driven form"  Nguyen et al. In SC’12
Communication cost comprises a significant part of large-scale application running time\(^1\). (Moreover, communication overheads are continuing to grow towards the *Exascale.*)

*For this reason...*

1. “[...] There is a need to investigate algorithms that reduce communication to a minimum.”\(^2\)

*However...*

2. Manual re-factoring of legacy MPI applications is impractical.

*Therefore...*

3. Automatic translation can help towards communication-efficient Exascale computing.
Related Work

- Tarragon
- Bamboo
- PhD Start:
  - Fall 2013
  - Fall 2014
- Thesis Work Performed:
  - Fall 2015
  - Fall 2016
- Today:
  - Fall 2017
  - Fall 2018
- Remaining Thesis Work:
  - Spring 2018
For (iterations)
{
    ... Receive Requests...
    ... Send Requests...
    -- Wait for Requests --
    ... Compute ...
}

# pragma Bamboo Overlap
For (iterations)
{
    # pragma Bamboo Receive
    { ... Receive Requests... }
    # pragma Bamboo Send
    { ... Send Requests... }
    # pragma Bamboo Compute
    { ... Compute ... }
}
Tarragon Model

A Bamboo-Translated code runs as a Tarragon\(^1\) program.

\(^1\)“Latency Hiding and Performance Tuning with Graph-Based Execution” P. Cicotti and S. Baden. In DFM’11

- Tarragon is a parallel programming model for communication-tolerant algorithms.
- Bamboo converts each original MPI process into a set (>1) of Tarragon tasks.
- Tarragon tasks are not assigned resources until data dependencies are satisfied.
- Communication cost is hidden by executing ready tasks while others are communicating.

Core Usage Timeline

Observation: The optimal task number is dependent on both the application and the system.
Bamboo’s Limitations

Bamboo demonstrated automatic translation can be used to the cost of communication, however:

- **Bamboo and Tarragon were not co-designed.**
  - Bamboo’s translation logic was constrained to the Tarragon runtime system’s design.

- **Static Scheduling Problem**
  - Tarragon provides a single execution entry point per task (*Tarragon_Execute*).
  - Bamboo needs embed static scheduling logic into the translated code.
    - Code Bloating: **15x** increase. Difficult to debug.
    - **No support for recursive code.** Incompatible with some production applications.

- **Handling MPI↔Tarragon Communication**
  - A description of the communication graph layout is required by Tarragon.
    - This is a problem domain-specific setting.
  - All communication needs to be annotated (even initialization/finalization).
  - Buffering and header wrapping is required to translate Tarragon to MPI messages.
    - This requires additional CPU overhead (memcpy) and memory bandwidth.
Refactoring Bamboo/Tarragon
Refactoring Bamboo/Tarragon

Goal: Refactor Bamboo and Tarragon simultaneously to address their limitations.

- **Milestone 1: Learn how Bamboo & Tarragon operate**
  1. Examine translated codes.
  2. Examine the source code of Bamboo (15k LoC) and Tarragon (16k LoC).

- **Milestone 2: Transfer scheduling/communication functionality from Bamboo to Tarragon.**

- **Bittersweet results:**
  - Reduced code bloating by a factor of ~3x.
  - Recursion remained a problem due to Tarragon’s single entry point mechanism.
    - Further re-factoring was impractical due to Bamboo and Tarragon’s complexity.
    - Could not get Tarragon to run efficiently in new architectures, but:
  - **Gained the how-to for building both a Translator and a Runtime System.**
New Goal: Co-Design a new translator and a new runtime system simultaneously.

- **Design a new Translator**
  - Minimal intervention: no static scheduling embedded in the code.
    - Negligible Code Bloating.
    - Debuggable code.
  - Minimal annotation requirements
    - No problem domain-specific annotations.
    - Annotated/non-annotated communication can co-exist.

- **Co-Designed with a new Runtime-System**
  - Supports multiple entry points.
  - Manages all MPI message handling.
  - Supports Recursive Execution.
Introducing **Toucan/MATE**

- **Toucan: an improved MPI Translator**
  - Built using the ROSE Compiler Framework (LLNL).
  - Uses a reduced set of Bamboo’s annotations (4 directives).

- **MATE Runtime System**
  - Uses lightweight threads (Coroutines) instead of static scheduling.
  - Coroutines can exit/re-entry a function at any given point.
  - Creates and schedules the dependency graph dynamically.

- **Toucan/MATE rely on two mechanisms:**
  1. Oversubscription of processor cores.
Core Oversubscription in Toucan/MATE

Split the problem domain into more partitions than useful cores in the system.

Typical MPI Decomposition
1 Subdomain per Core

Toucan’s Overdecomposed Grid
4 Subdomains per Core
(Single Task Pool per Node)
#pragma toucan superblock
for (int i = 0; i < niterations; i++)
{
    #pragma toucan receive
    { MPI_Irecv(BufferGrid ← LeftNeighbor);
      MPI_Irecv(BufferGrid ← RightNeighbor); }

    #pragma toucan send
    { MPI_Isend(Grid ← LeftNeighbor);
      MPI_Isend(Grid ← LeftNeighbor); }

    #pragma toucan compute
    { Compute(); Swap(&Grid, &BufferGrid); }
}

Toucan defines 3 code region types:
(Compute, Send, Receive)

Loop is divided into 3 separate steps.

Coroutine yields to MATE Scheduler (instead of OS)

Dependency Graph defined implicitly:
- Compute depends only on receive
- Receive depends on compute*
- Send depends on compute* and send*

*From previous iteration
Runtime System (MATE/Toucan)
Example: 1D Stencil Jacobi Solver

```c
#pragma toucan superblock
for (int i = 0; i < niterations; i++)
{
    #pragma toucan receive
    { MPI_Irecv(BufferGrid ← LeftNeighbor);
      MPI_Irecv(BufferGrid ← RightNeighbor); }

    #pragma toucan send
    { MPI_Isend(Grid ← LeftNeighbor);
      MPI_Isend(Grid ← LeftNeighbor); }

    #pragma toucan compute
    { Compute(); Swap(&Grid, &BufferGrid); }
}
```

MATE_AddRegions("receive", "send", "compute");
MATE_AddDependency("compute" → "receive");
MATE_AddDependency("send" → { "compute", "send" } );
MATE_AddDependency("receive" → "compute");

```c
int iReceive = 0; iSend = 0; iCompute = 0;
while (MATE_GetNextRegion(&regionId)) switch (regionId)
{
    case: "receive"
      MATE_RequestData(BufferGrid ← LeftNeighbor);
      MATE_RequestData(BufferGrid ← RightNeighbor);
      if (++iReceive >= niterations) MATE_RemoveRegion("receive");
      break;

    case: "send"
      MATE_PushData(Grid ← LeftNeighbor);
      MATE_PushData(Grid ← LeftNeighbor);
      if (++iSend >= niterations) MATE_RemoveRegion("send");
      break;

    case: "compute"
      Compute(); Swap(&Grid, &BufferGrid);
      if (++iCompute >= niterations) MATE_RemoveRegion("compute");
      break;

    default:
      MATE_Yield();
}
```

385 LoC → 491 LoC (1.27x Increase)
NERSC Edison Supercomputer:
5586 Computing Nodes

Processor: 2x12-core Intel "Ivy Bridge" @2.4Ghz
Memory: 64 Gb DDR3 Total per Node

Software:
- Cray-MPICH v7.4.1
- Intel icc compiler 15.0.1 (-O3)
- Intel MKL Library (for dgemm)
Test Cases

We used 4 examples from 3 common scientific application motifs:

**Cannon 2D (Dense Linear Algebra)**
Com-putes the matrix product of two matrices.

**LULESH 2.0 (Unstructured Grid)**
Solves the Sedov blast problem. Developed at LLNL.

**Cart3D (Unstructured Grid)**
Multigrid solver of Euler equations. Relies on recursive code.

**Jacobi 3D (Structured Grid)**
Solves the Poisson equation for 3D PDEs.
Results @ Edison

Toucan Model Limitation

**Observation:** Overdecomposition requires additional *internal* communication.
MATE Hybrid Model

- **New Model**: Workload decomposed twice. (1) Process-wide and (2) Within Shared Domain

**Observation**: Hybrid model only requires synchronization for tasks sharing the same subdomain.
Programming with Hybrid Model

- Hybrid Model requires manual changes in the workload distribution part of the application.
- Two possible approaches:

Pros:  
- Single shared malloc  
- Contiguous Access  

Cons:  
- Requires two decompositions  
- Cache blocking sensitive

Pros:  
- Re-uses original decomp.  
- Cache efficient  

Cons:  
- Requires pointer passing  
- Non-contiguous access

Decompose by Process, Decompose by Subrank

Decompose by Subrank, Local Pointer Access
I. Introduction
II. Related Work
III. Refactoring
IV. Toucan
V. Hybrid
VI. Remaining Work

Runtime System (MATE/Hybrid)
Results @ Edison

- Toucan Model
- Hybrid Model

<table>
<thead>
<tr>
<th>% Communication Hidden</th>
<th>100</th>
<th>90</th>
<th>80</th>
<th>70</th>
<th>60</th>
<th>50</th>
<th>40</th>
<th>30</th>
<th>20</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6144 Cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jacobi3D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 7% Improvement
- ~57x Improvement
- 40% Performance Gain
(2017) Having shown promise with the Hybrid model, we decided to shift to a newer platform:

**NERSC Edison Supercomputer:**  
Operational since 2013

**Node Configuration:**  
- 2x12-core Intel "Ivy Bridge" @2.4Ghz  
- 460 Gflops/node

**NERSC Cori KNL Supercomputer:**  
Operational since 2017

**Node Configuration:**  
- 68-core Intel “Knights Landing” @1.4Ghz  
- 3000 GFlops/node
Results @ Cori KNL

-2.5x Improvement

8192 Cores

Jacobi3D

16%

48%

Toucan Model
Hybrid Model
Progression Roadmap (2017)

Progression since Winter 2017.

Starting from Toucan (16%) until today (84%).

Each bar represents an improvement.

~5.2x Improvement
Optimization 1: Subrank Prioritization

- **Fact:** Not all subranks incur the same communication cost.
- **Idea**: 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)

---

Results: Prioritization

Comparing 3 Variants:

**Hybrid (Default)**
No priority scheme.

**Hybrid (Boundary)**
+Priority to Boundary Subranks

**Hybrid (Center)**
-Priority to Boundary Subranks
Non-Contiguous buffers in MPI need to be packed before communicating.

MPI implements a process-wide lock, which limits communication concurrency.

**Solution:** Have MATE perform automatic thread-safe packing before calling MPI.
Results: MPI vs. Mate Packing

Comparing 2 Variants:

**Hybrid**
MPI-Only

**Hybrid**
Mate Packing + MPI
Optimization (3/3): Explicit Graph in MATE

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

Explicit Dependency Graph

5 Task Exit Points.

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

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

    #pragma mate region(compute) depends (unpack)
    {
        Compute();
        Swap(&U, &Un); }
```
Comparing 3 Variants:

**Hybrid** (Implicit)
Toucan Model

**Hybrid** (Explicit)
+Pack/Unpack Regions

Results: Dependency Graph Variants
Hybrid Model Conclusions

- The Hybrid Model exceeds the efficiency of the Toucan Model.
  - Hides communication by oversubscribing cores (like Toucan) but,
  - It does not require additional communication.

- Subrank prioritization can have a substantial impact on performance.
  - MATE can assign priorities adaptively during execution.

- Thread concurrency is still an important issue to be solved.
  - Packing can be performed concurrently, but MPI still locks comm ops.
  - Solution: Use a different communication layer? (GasNET / UPC++)

- It is possible to refine dependency graphs explicitly.

- Limitation: Require some manual changes to the workload distribution logic.
  - Possible topic for another thesis: make translation fully automatic.
Cart3D in Depth

Cart3D is a high-fidelity analysis package for aerodynamic design.

- Production code developed by NASA Ames and NYU Courant Institute of Mathematics.
- Has hundreds of users.
- Uses a multigrid with irregular meshes.
- Complex Code: 38k Lines of Code + Recursion.
- Non-trivial communication:
  - Irregular (asymmetric) communication.
  - Non-contiguous data types.
  - 4 communication regions in the main loop.

Source: https://www.nas.nasa.gov/publications/software/docs/cart3d/
Why is Cart3D Important?

Cart3D is an ideal test case:

- Demonstrates that translation can improve the performance of a production code at scale.
- We are working with developers at NYU / NASA.
- We have shown that Toucan can hide 33 to 41% of communication cost at scale in our Edison experiments (256 nodes).

Hybrid Model can outperform Cart3D on Toucan:

- Cart3D on Toucan suffers from added communication.
- We expect to achieve similar improvements as with Jacobi3D (~80% of communication hidden)

Next Milestones

- Apply Toucan to the latest version of Cart3D. (1~2 Months)

- Develop the Hybrid Model variant of Cart3D. This requires manual restructuring of Cart3D. (6~7 Months)
  - Run experiments at scale (>1024 Nodes) for Cart3D and other test cases.
  - Use parallel profiling tools (HPC Toolkit) to examine the low-level effects of our models. (2~3 Months)
  - Write and submit a paper to a main HPC conference (e.g. SC, IPDPS, EuroPar)

- Write and defend PhD Dissertation (~3 Months)

- Total: ~13-15 Months
Questions?