Toucan - A Translator for Communication Tolerant MPI Applications

Sergio M. Martin*
*Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093, USA
Email: sergiom@eng.ucsd.edu

Marsha J. Berger ‡
‡Computer Science Department, Courant Institute
New York University
New York, NY 10012, USA
Email: berger@cims.nyu.edu

Scott B. Baden*†
†Computational Research Division
Lawrence Berkeley National Laboratory
Berkeley, CA 94720, USA
Email: baden@lbl.gov

***Abstract*—We discuss early results with Toucan, a source-to-source translator that automatically restructures C/C++ MPI applications to overlap communication with computation. We co-designed the translator and runtime system to enable dynamic, dependence-driven execution of MPI applications, and require only a modest amount of programmer annotation. Co-design was essential to realizing overlap through dynamic code block reordering and avoiding the limitations of static code relocation and inlining. We demonstrate that Toucan hides significant communication in four representative applications running on up to 24K cores of NERSC's Edison platform. Using Toucan, we have hidden from 33% to 85% of the communication overhead, with performance meeting or exceeding that of painstakingly hand-written overlap variants.**

***Keywords*-Communication/Computation Overlap; Source-to-Source Translator; MPI; Data-Driven.**

I. Introduction

A major challenge in HPC is to reduce the growing cost of communication overhead. A well known coping strategy is to overlap communication and computation, keeping processors busy with "useful" work while waiting for data to arrive. The programmer needs to restructure the application to realize overlap, e.g. via split-phase coding. However, such hand-coding is complicated and increases software development costs.

We have built a source-to-source translator called *Toucan* to automate the process, reducing the programmer's burden considerably. Starting with legal C or C++ MPI source,[1] Toucan generates semantically equivalent code that runs under a data-driven execution model and overlaps communication with computation automatically. Toucan requires a modest amount of programmer annotation in the form of directives. Guided by these directives, it generates a communication-hiding, data-driven variant of the code.

A major innovation in this paper is our novel strategy of achieving overlap via dynamic out-of-order code execution. We also illustrate the larger point that a translator should be co-designed with its runtime system. The previous translator, Bamboo [20, 22, 21], used static translation-time code relocation and inlining to realize overlap via out-of-order execution of user-defined code regions. This was a consequence of built-in assumptions made by the runtime system, that had been developed separately. Static code relocation and inlining are not an appropriate way to handle out-of-order execution as they lead to excessive code bloat and do not support recursion. By co-designing the runtime system with the translator, we were able to avoid these limitations.

Toucan relies on two mechanisms to maximize overlap. First, overdecomposition is used to enhance pipelining of communication with computation by creating more tasks than cores [14]. We achieve this effect by instantiating multiple "subranks" of each original MPI rank. These sibling subranks run as a data-driven program and execute out-of-order according to the flow of data. Under this model, an application rarely blocks. If a subrank is not yet ready to execute, the scheduler can often find another one that is ready.

Second, Toucan refines the granularity of data-driven execution by subdividing a subrank's execution into code regions, phases that can be scheduled according to their dependencies. This subdivision enables tasks to suspend in order to let another task execute in its place, increasing the opportunities for overlap.

In this paper we discuss the early results obtained with Toucan, translating four applications representing three common HPC motifs on up to 24K cores of NERSC's Edison platform. Using Toucan, we have hidden from 33% up to 85% of the communication overhead. The performance of the code generated by Toucan meets or exceeds that of painstakingly hand-written, split-phase code variants. We achieve this result using a simple pragma-based interface and a runtime system that automatically manages data-driven execution.

The rest of this paper is organized as follows: §II introduces Toucan's annotations and principles of operation, §III describes the translation process and Toucan's runtime system, §IV presents experimental results, §V is a discussion including related work, and §VI presents conclusions, including future work.

[1] The strategy employed by Toucan is language-neutral, and could be applied to other MPI bindings, e.g. FORTRAN.

```
1  #pragma toucan superblock
2  for (int iter = 0; iter < nIter; iter++) {
3      #pragma toucan receive
4      { MPI_Irecv(recvBuffer ← [left  neighbor]);
5        MPI_Irecv(recvBuffer ← [right neighbor]); }
6
7      #pragma toucan send
8      { Pack(Uprev → leftSendBuffer);
9        Pack(Uprev → rightSendBuffer);
10       MPI_Isend(leftSendBuffer → [left  neighbor]);
11       MPI_Isend(rightSendBuffer → [right neighbor]); }
12
13     #pragma toucan compute
14     { MPI_Waitall();
15       Unpack(U ← leftSendBuffer);
16       Unpack(U ← rightSendBuffer);
17       for (int i = 0; i < N; i++)
18           U[i] = Uprev[i−1] − 2*Uprev[i] +  Uprev[i+1];
19       swap(&U, &Uprev); }
20 }
```

Figure 1: Annotated 1D, 3-point Jacobi kernel.

II. Toucan

A. Annotation Model

Toucan reinterprets C or C++ MPI source as a data-driven, task parallel program [1, 13]. This data-driven program executes tasks out-of-order, enabling the runtime system to overlap communication with computation. To achieve this, a programmer annotates a correctly written (e.g., free from deadlock) MPI program to split the code into regions employing Toucan's *pragma* directives. The translator uses this information to support out-of-order execution.

Fig. 1 lists the annotated pseudocode for a simple 1D Jacobi solver, a stencil method we will use to illustrate Toucan's principles of operation and code generation strategy. The code iteratively sweeps over a one dimensional grid of N elements partitioned uniformly across the MPI ranks as shown in Fig. 2. At the start of each iteration, MPI ranks exchange the boundary cells of their local partition to neighboring ranks (*left*, and/or *right*) and store them locally in an extended portion of the mesh called the *ghost cells*.

As illustrated in Fig. 1, Toucan realizes overlap over a pattern of communication and computation residing within an iteration. Each pattern is contained with a *superblock* (line 1). Any code residing outside a superblock is passed to the output source file unchanged and executes in-order. This strategy supports incremental development and code that is deemed not critical to performance need not be annotated.

Code within a superblock must reside within one of three kinds of regions that partition the superblock disjointly. There must be exactly one of each type of region in a superblock.

1) *Receive regions* (*#pragma toucan receive*) enclose MPI *recv* operations and any other statements required to set up the arguments for those operations (lines 3-5).
2) *Send regions* (*#pragma toucan send*) enclose non-MPI C/C++ statements required for buffer packing and setup and MPI *send* operations (7-11).

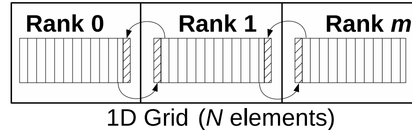

Figure 2: Grid decomposition of the 1D stencil solver. Ghost cells, marked in cross-hatch, reside on the edges of each local partition, except for the physical boundary.

3) *Compute regions* (*#pragma toucan compute*) (13-19) enclose *wait* operations, buffer unpacking, and any other code that depends on the communications issued in the *receive* region.

B. Execution Model

Superblocks execute in the order they appear in the program, and code regions execute in the order they appear in the enclosing superblock. The subdivision of a superblock into distinct send, receive and compute regions is fundamental to how Toucan handles out-order-execution. The effect is to enable Toucan to overlap communication with computation by pre-empting the current superblock with a superblock coming from another task assigned to the same processor. This effect is enhanced by *overdecomposition*, whereby a program invocation runs with more tasks than cores [16]. Overdecomposition increases the pool of superblocks that may be executed out-of-order, improving communication/computation overlap. When a superblock finishes executing a region, the scheduler will choose a different superblock with a region available for execution. When we overdecompose, we grow the number of ranks by a factor V. The value for V is a tuning parameter, and is an algorithm and system dependent quantity. The user sets the value at the time they launch the program.

By subdividing a superblock into regions, we allow a superblock to make progress incrementally, which increases the likelihood that there are a sufficient number of available regions that can execute, while overdecomposition ensures that a core will likely be able to substitute a runnable task. Of course, this assumes that there is sufficient computation to hide communication.

Once all data dependencies are satisfied, a region executes atomically. When the region completes, the runtime's scheduler will attempt to execute the next region in the superblock, if one exists. Otherwise, the scheduler will select a region from another task, whose dependencies have been satisfied.

At present, all of our applications use the same receive-send-compute structure contained within an iterative *for* loop. While other structures are possible, we will assume the R-S-C structure in this paper, where *R, S* and *C* stand for send, receive and compute. These 3 regions divide a superblock's execution into 3 pipeline stages. Toucan creates a data-driven loop to enable stages to execute in sequence, but with interlocks to ensure that a stage begins only when its dependencies have been satisfied. Our runtime

system manages the flow of data, enabling a superblock to execute when at least one of its regions have satisfied its dependencies, and maintains local state to ensure that the progression R, S, C executes sequentially inside a single superblock.

A C region will not be runnable until all the data dependencies registered in the current R and previous iteration S regions have been satisfied. In case an R region posts receives for multiple messages, execution in the dependent C region cannot begin until *all* the messages have arrived.

There are no explicit waiting operations in the translated code. All communication (send/recv) appearing within a S or R region is transformed into asynchronous non-blocking requests, while wait operations are outlined into a callback made by the runtime system. Only the detection of a message arrival within the runtime system's communication handler can enable a region within a superblock to begin executing. Thus, non-blocking communication requests return immediately and communication is handled by the runtime system, out of the critical path of the application.

A task will not execute a region unless all dependencies have been satisfied. If no regions are available, the task will be preempted and replaced by a runnable region from another task whose dependencies have been met. Task preemption is handled by the out-of-order logic in the runtime system.

III. Implementation

A. Code Translation

We built our translator using the ROSE compiler infrastructure [25]. ROSE relies on the Edison Design Group front-end [10] to parse and generate the abstract syntax tree (AST) of an MPI C/C++ program which is kept in memory during the translation process. ROSE provides tools for analyzing and modifying the generated AST.

A program translated by Toucan executes under the control of our runtime system, called *MATE*[2], which we describe in further detail later in this section. Toucan collects information about MPI call sites, and uses pragmas to guide its transformations. The translation process introduces a small number of statements to support out-of-order execution, including calls to the MATE API to manage dependencies at runtime.

After translation, we use ROSE's back-end to generate the resulting code from the modified AST. The resulting MATE-compatible code is still human-readable and can be easily debugged. The code listed in Fig. 3 shows the outcome of the translation process applied to the annotated code in Fig. 1. This code will be compiled by a C++ compiler and linked with the MATE runtime system into an executable.

Toucan processes the source code one superblock at a time, in program order. Upon finding a superblock, the

[2] A typical South American beverage, pronounced MAH-tay.

```
1  Mate_PushSuperblock();
2
3  // for (int iter = 0; iter < K; iter++) elided during translation
4  int iter_0 = 0, iter_1 = 0, iter_2 = 0, regionId;
5
6  while(MateGetNextRegion(&regionId))
7   switch (regionId) {
8    case NO_REGION_READY:
9           Mate_SuspendSubRank();  break;
10
11   case "receive":
12       Mate_RequestData(recvBuffer ← [left  neighbor]);
13       Mate_RequestData(recvBuffer ← [right neighbor]);
14       iter_0++; if (iter_0 >= K) Mate_FinishRegion("receive");
15       else Mate_AdvanceRegion("receive");
16       break;
17
18   case "send":
19       Pack(Uprev → leftSendBuffer);
20       Pack(Uprev → rightSendBuffer);
21       Mate_PushData(sendBuffer → [left  neighbor]);
22       Mate_PushData(sendBuffer → [right neighbor]);
23       iter_1++; if (iter_1 >= K) Mate_FinishRegion("send");
24       else Mate_AdvanceRegion("send");
25       break;
26
27   case "compute":
28       // MPI_Waitall(); call elided during translation
29       Unpack(U ← leftSendBuffer);
30       Unpack(U ← rightSendBuffer);
31       for (int i = 0; i < N; i++)
32           U[i] = U[i−1] + U[i+1] − 2*U[i];
33       swap(&U, &Uprev);
34       iter_2++; if (iter_2 >= K) Mate_FinishRegion("compute");
35       else Mate_AdvanceRegion("compute");
36       break;
37
38   default: error_handler(); break;
39  }
40 Mate_PopSuperblock();
```

Figure 3: Final generated code with calls to MATE runtime.

translator generates a call to *Mate_PushSuperblock* (line (1)), which creates a new superblock/region structure and places it on top of the superblock stack. MATE schedules regions that belong to the the superblock residing at the top of the stack only. Meanwhile, superblocks residing below the top of stack are suspended and none of their regions can be scheduled. This mechanism allows Toucan to support recursive code, which was used in one of our applications.

Toucan restructures the *for* loop (3), using a *while* loop (6), managing the iterations inside the *switch* statement (7-39) that supports support out-of-order execution across different superblocks. Each case of the switch statement corresponds to a region in the original (annotated) code. The increment and test at the end of each case/region (14, 23, 34) manages the iteration expressed in the original *for* loop[3].

Once a region completes, there are two possible outcomes. Either more iterations remain (iteration test evaluates to

[3] Toucan also evaluates the loop condition at the beginning of the region in case the condition is false in the first iteration. We omit this code to simplify the explanation.

false) or we are in the final iteration (test evaluates to true). In the former case, the *Mate_AdvanceRegion* function (15, 24, 35) is called, instructing the MATE scheduler to leave a continuation marker so that when the superblock next executes, control will pass to the following region in the sequence *receive, send, compute*. Otherwise, the *Mate_FinishRegion* function is called, preventing the region from executing again in the current invocation of its enclosing superblock.

After executing a region, a call to *Mate_GetNextRegion()* (6) queries the runtime scheduler for a region in the current superblock that is available for execution. The case statement dispatches on the three possible region types (send, receive, compute). If no region is ready for execution (at least one of their data dependencies is not satisfied), the subrank is preempted (*Mate_SuspendSubRank*), and a region from another subrank (if available) will execute in the current core instead. Otherwise, the loop will spin.

Toucan removes all calls to communication requests (send/receive) from the code during translation and replaces them with calls to *Mate_RequestData* or *Mate_PushData* (12-13, 21-22), which signals the runtime to handle data motion in the background. These calls return quickly and are asynchronous. Since execution is driven entirely by dependencies and requests handled by the MATE scheduler, all calls to blocking *wait* operations (e.g. *MPI_Wait*, *MPI_Waitall*) are eliminated.

The *while* iteration ensures that a superblock runs until all regions have completed. At this point, the *Mate_PopSuperblock* (40) function is called, removing the superblock from the top of the stack. If this was a recursive call, execution returns to the call, else execution continues to the next superblock. If the next line of code falls outside a superblock, it will not be translated by Toucan, and will be executed in-order.

B. Runtime Support

By co-designing MATE and Toucan, we were able to avoid overheads that were not apparent in our initial design of either software component. This decision was the result of lessons learned from the Bamboo translator, which *was not* co-designed with its runtime, leading to difficulties described in §V-A. To avoid these difficulties, we designed MATE to support data-driven execution at runtime, encapsulating all out-of-order scheduling logic behind its API.

Toucan restructures an application to run as a set of tasks. These tasks are mapped onto threads running within MATE processes distributed across separate address spaces. A MATE *process*, illustrated in Fig. 4, handles both runtime structures (e.g. the scheduler) and the application's *worker threads*. The MATE *scheduler* manages dependencies and assigns regions to workers for execution. In most of our experiments, we execute one MATE process per NUMA domain to maximize locality. The exception was

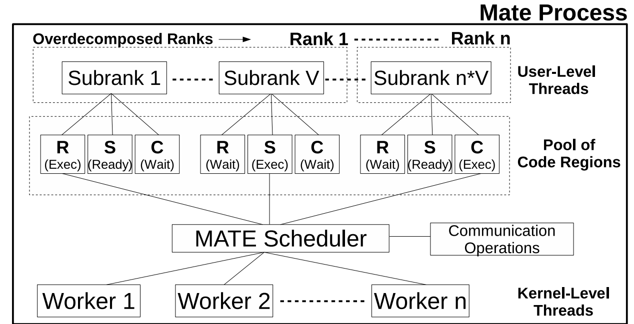

Figure 4: Structure of a MATE Process. The original *n* MPI ranks are overdecomposed by a factor of *V*, yielding *n*V* Toucan subranks. These are subsequently divided into regions, which are individually scheduled. Subranks 1 through V are sibling subranks and are cousins of Subrank *n*V*.

mpix_flowCart, in which 6 processes per NUMA domain (or 12 per node, which contains 2 NUMA domains) was optimal. The bookkeeping is distributed across the domains, and is not centralized. Thus, each MATE process stores only the metadata required to schedule its own subranks.

A *subrank* is the outcome of overdecomposition, a subdivision of MPI ranks. We implement subranks assigned the same process as user-level threads that can be suspended/resumed at any point of execution while preserving their execution stack.

When we launch a Toucan program, we specify the mapping of Toucan subranks onto MATE processes. It is important to make a distinction between *sibling* and *cousin* subranks living within the same MATE process. We consider two subranks to be *siblings* when they belong to the same overdecomposed MPI rank, while we consider them to be *cousins* when they belong to different MPI ranks assigned to the same MATE process. This distinction is important for scheduling. When MATE suspends an executing subrank (i.e. none of its regions are ready to execute), it is replaced by a sibling subrank. If no regions in sibling subranks are ready (i.e. contains at least one *ready* region), it replaces it with a cousin subrank. This prioritization of sibling subranks is made to preserve data locality.

Any subrank in the MATE process that is ready to execute, can be scheduled in any core, improving computation/communication overlap and core utilization. Our scheduler selects a candidate region across all subranks owned by the process using a round robin strategy.

We use Boost's C++ Coroutines[4] to create and manage subranks and our own scheduler to determine what regions within a subrank are ready to execute based on their dependencies. This lightweight design enables MATE to quickly schedule an available region after another one completes.

MATE processes are instantiated as OS processes, containing their own private address space. We currently use

[4]http://www.boost.org/doc/libs/release/libs/coroutine/

MPI to invoke MATE processes and handle their communication. For subranks that communicate within the same process, MATE uses a rendezvous strategy to transfer messages through a *memcpy* operation as soon as both *MATE_PushData*/*Mate_RequestData* requests are issued, eliminating the need for intermediate buffering.

MATE *workers* multiplex both the application and runtime services, including communication. They are kernel-level threads mapped statically 1-to-1 to processor cores. Workers allocated to the same process share a common pool of regions to be executed. To support parallel execution of multiple subranks per process, we instantiate multiple worker threads using the *pthreads* library. Although many worker threads may be executing regions at the same time, only one at a time can play the role of communicator. This is a limitation of our current design, and we are looking other approaches that permit many cores to inject data into the NIC simultaneously.

The *communicator worker* polls for messages that have arrived using *MPI_Testall()*. Since the runtime uses MPI to transport data among ranks, we need a way of enumerating tasks, which will be more numerous than MPI ranks due to the use of overdecomposition. To this end, we use part of the MPI tag field to store the task's ID, along with the MPI rank returned by *MPI_Testall()*. This scheme reduces the effective Toucan message tag space. On our experimental platform, MPI tags are 24 bits wide. Of these, we currently reserve 8 bits to resolve Toucan subrank IDs, leaving 16 bits for the tag. This is sufficient to handle a very high overdecomposition factor (256), far larger than we have encountered in practice.

At present, our test applications are not limited by the performance of collective communication, so our strategy is to prioritize functionality (in particular, so that it works correctly with overdecomposition) over efficiency. For this reason, we do not currently provide support for overlapping MPI collective operations, and they must be placed outside superblocks. There are, however, applications requiring a more comprehensive solution and we currently investigating ways to provide a performance-efficient support of collective operations.

IV. Experimental Evaluation

A. Overview

Our computational testbed is *Edison*[5], a Cray XC30 supercomputer located at the National Energy Research Scientific Computing Center (NERSC). Edison contains 5576 compute nodes, each with two 12-core Intel *Ivy Bridge* processors for a total of 24 cores per node. Although each core can execute 2 simultaneous threads, we observed no significant performance benefits from enabling hyperthreading in our experiments. We used Cray-MPICH v7.4.1 and Intel's *icc* Compiler version 15.0.1 with optimization level -O3. For double-precision dense matrix multiplication (*dgemm*), we use the Intel MKL library.

We tested Toucan on 4 applications coming from 3 application motifs: structured grid, dense linear (matrix) algebra and unstructured grid [7]. We tested each motif using the following variants:

MPI-Original. Does not attempt to overlap communication with computation.

Toucan. Annotated MPI-Original code translated using Toucan.

MPI-NoComm. A modified version of MPI-Original with communication disabled. Although it does not produce correct results, MPI-NoComm provides a loose upper bound on the performance benefit of overlap.

MPI-Overlap. A manually optimized version of MPI-Original that realizes overlap. Only available for some of the codes.

We ran all variants using the same combination of algorithm and # cores in the same job, to reduce variation in node allocations and to ensure fairness when comparing different variants. Each variant executed 3 to 5 times, with fewer times for applications with longer running times. Running times varied by not more than 1% for any given application variant.

B. 3D Jacobi

This code solves the 3D Poisson equation subject to Dirichlet boundary conditions using Jacobi's method. The computation iteratively sweeps the mesh using a 7-point central difference stencil that updates each element of the grid with the average of the six nearest values.

MPI-Original uses a 2D cache blocking scheme for the non-contiguous Y and Z axes in its solver kernel. We experimentally determined that the optimal blocking size for Edison's computing nodes is 64×64, that targets L2 cache locality. We observed that no further cache performance gain was obtained from overdecomposition, and thus any performance improvement realized by Toucan can be ascribed to communication overlap alone.

The *MPI-Overlap* variant is that used in [20]. A manually restructured variant of MPI-Original, it employs a split phase execution to overlap communication and computation. Each MPI rank divides its local mesh into a 3D $2 \times 2 \times 2$ geometry and executes as a pipeline, performing communication for each processed sub-mesh while computing the next one[6].

We conducted a strong scaling study to stress communication, running with a 3072^3-element mesh on 3072, 6144, 12288, 24576 cores. We tested the MPI-Original, MPI-Overlap, and MPI-NoComm variants using 24 MPI ranks

[5] http://www.nersc.gov/users/computational-systems/edison/configuration

[6] Another overlap technique for stencil methods is to defer computation on an inner annulus until ghost cells updates have completed. However, in previously published results [3], we have shown that this solution suffers from the poor cache locality in the deferred updates. We ran experiments with this variant, and confirmed there was no benefit.

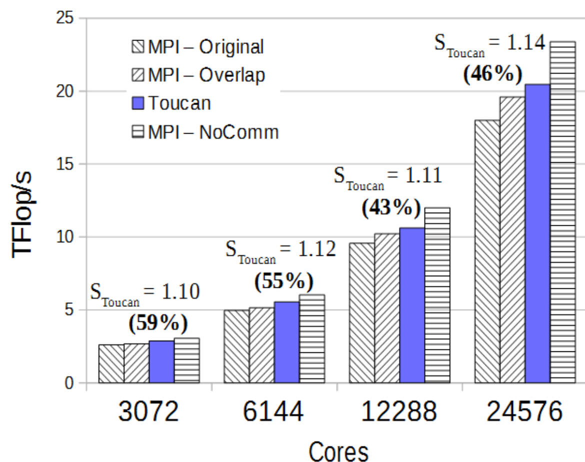

Figure 5: Performance of different variants of 3D Jacobi on 3K, 6K, 12K, 12K cores. S_{Toucan} is the speedup of Toucan over MPI-Original. The numbers in parentheses are % communication hidden by Toucan.

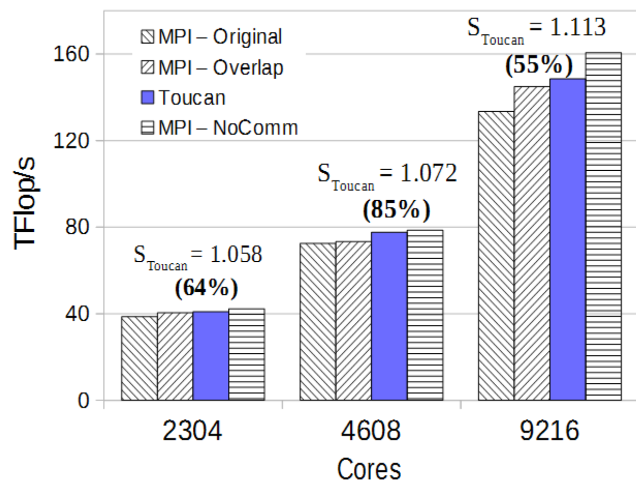

Figure 6: Performance of different variants of 2D Cannon's algorithm running on 2304, 4608, and 9216 cores.

per node (one per core). The Toucan variant is MPI-Original annotated with 4 directives (1 superblock). To run this variant, we allocated 2 MATE processes per node (one per NUMA domain), each running 12 MATE worker threads. We experimentally determined that using 2 subranks per core delivers optimal overlap in all cases.

Fig. 5 shows the results for 3D Jacobi. The Toucan-generated code improved performance by 10%, hiding 59% of the communication cost in the 3072 core run, while the 24576 core run improved performance by 14%, hiding 46% of communication.

The Toucan variant also outperformed the manually optimized variant in all cases: *MPI-Overlap* hid up to 30% of the communication, improving overall performance by up to 9%. We attribute the difference to two causes. First, the overlap strategy is hard coded into *MPI-Overlap* and the overdecomposition factor is fixed at 8, 2 in each of 3 coordinate directions. Though this factor is slightly sub-optimal on Edison (verified with experiments with the Toucan variant), it is not the main reason for the performance difference. The second cause is more significant: MATE schedules across cousin threads and this is not possible in *MPI-Overlap*. Toucan can realize overlap more effectively than *MPI-Overlap's* fixed strategy (and with a smaller overdecomposition factor) since each core can execute any cousin subrank residing in the same NUMA domain. *MPI-Overlap* is limited to overlapping within a single MPI rank.

We also ran 3D Jacobi under AMPI [12], which relies on overdecomposition and asynchronous communication. We used the latest version of AMPI provided within the *Charm++ 6.7.1* configured with the *gni-crayxc-smp* architecture which is optimized for the Cray XC90 computers. We employed AMPI to translate *MPI-Original* and performed tests on up to 24K cores. We verified experimentally that the best performance is obtained when AMPI executes exactly as we had configured Toucan: 2 processes per node, 12 threads per process, and an overdecomposition factor of 2. Our results show that AMPI was able to hide a small amount of the communication: 9% (1.02x speedup) with 12K cores, and 6% (1.02x) with 24K cores. By comparison, Toucan hid 43% (1.11x), and 46% (1.14x), respectively.

C. Matrix Multiplication: 2D Cannon's Algorithm

The *2D Cannon's* algorithm [6] computes the matrix product of two matrices $C = A \times B$ in a series of $\sqrt{p}$ steps, where $\sqrt{p}$ is the number of ranks. Each rank owns a square sub-block of C and also holds local sub-blocks of A and B. Each step rotates sub-blocks A and B along rows and columns of the 2D rank array and then computes a partial matrix product using *dgemm* to update its local portion of C ($C \mathrel{+}= A \times B$). 2D Cannon requires that the number of ranks form a perfect square. To accommodate Edison's nodes, we use perfect squares that are divisible by 24, e.g. 2304, 9216. We were also able to run tests on 4608 ($9216/2$) cores by using 48 MPI ranks per node for MPI variants.

MPI-Overlap, a manually restructured version of MPI-Original, employs additional buffers to communicate the matrix rotation for the next step while performing the *dgemm* computation of the current step.

The Toucan variant for 2D Cannon is MPI-Original annotated with 4 directives (1 superblock). We used 2 MATE processes per node, 12 worker threads per process, and overdecomposition factors of 2 or 4 subranks per core. We observed that increasing the overdecomposition factor did not affect cache performance of *dgemm* as it is highly optimized for the memory hierarchy. Thus, any performance improvement is due to communication overlap alone.

We performed a weak scaling study on 2304, 4608, and 9216 cores, with matrix sizes N = {67890, 85536, 107770}, respectively. Fig. 6 shows the results we obtained. Toucan was able to improve over the manually optimized overlap

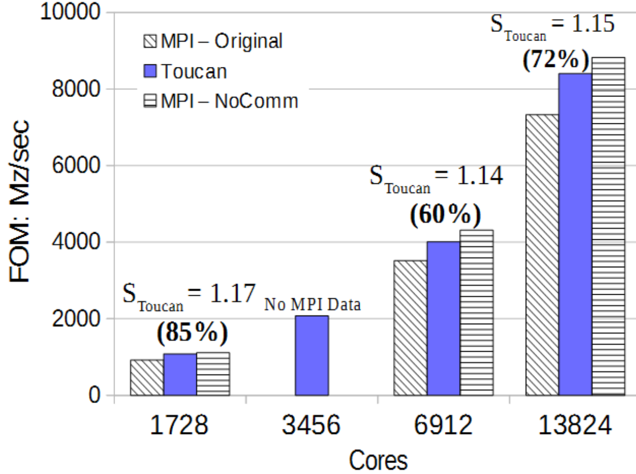

Figure 7: Performance of different variants of LULESH 2.0 on 1728, 3456, 6912, and 13824 cores, measured in million zones per second.

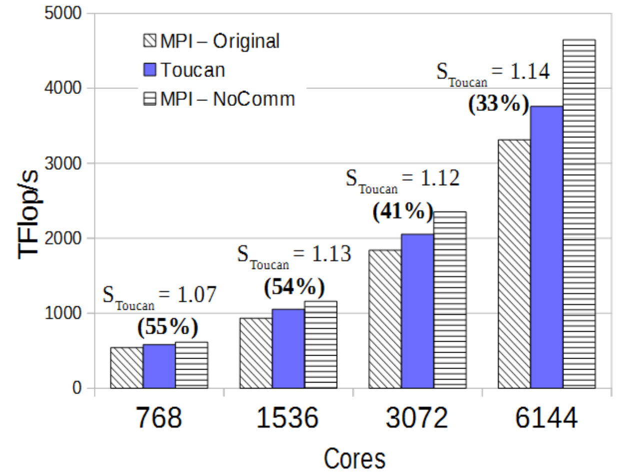

Figure 8: Performance of different variants of mpix_flowCart on 768, 1536, 3072, and 6144 cores.

variant in all cases. On 2304 and 4608 cores, Toucan was able to hide most of the communication cost (64% and 85%, respectively). On 9216 cores, the cost of communication increased abruptly. In this case, Toucan hid 55% of the communication and we also see a similar drop in the ability of the *MPI-Overlap* to hide communication. To explain the cause of this phenomenon, we hypothesized that communication across Edison compute cabinets may be causing an increase in communication imbalance. Indeed, every Edison cabinet contains 4608 cores each and the likelihood that more messages are crossing cabinet boundaries is expected to increase when jumping to 9216 cores. To test this idea, we repeated our tests using two variants of MPI-Original, ones that only communicate between (1) up/down and (2) left-/right neighbors. The result is that neither of these versions suffered the degradation observed in our results, suggesting that there is an interference pattern between both directions caused by an imbalance in message latency. To confirm this observation, we performed a manual rearrangement of task mapping to reduce the number of messages crossing node boundaries. This measure also realized a mitigation of communication imbalance. Based on this observation, we are considering developing ways to define custom rank mappings in Toucan-translated applications to improve overlap.

D. LULESH

LULESH [11] is an unstructured grid method, a proxy application developed at Lawrence Livermore National Lab [26]. As the starting point (*MPI-Original*), we used version 2.0 of the publicly available code [18]. There is currently no available *MPI-Overlap* version of the code that has been hand optimized for overlapping communication. The Toucan variant is MPI-Original annotated with 20 directives (5 superblocks).

LULESH requires a cubic processor geometry to divide the three dimensional space. We use a set of cubic numbers that are also divisible by 24, since there are 24 cores per Edison node (e.g. 1728, 13824). We were also able to run tests in 6912 cores $(13824/2)$ by using 48 MPI ranks per node for MPI variants.

We conducted a weak scaling study, keeping the number of grid elements (*zones*) fixed at 64^3 per core. While a strong scaling would stress communication more, we have found that it also dramatically reduces per-rank computation in LULESH, limiting any opportunity to hide the cost of network latency. LULESH provides a *Figure of Merit* for performance comparison calculated in $Mzones/s$. The results for our tests using are shown in Fig. 7. On 1728, 6912 and 13824 cores, Toucan was able to hide most of the communication overhead: 60% to 85%.

On 3456 cores, there is no compatible node/core configuration to satisfy a cubic rank distribution for the MPI variants, since that would require more ranks per node (96) than allowed by the *slurm* job scheduler. For this reason, we had no baseline data to calculate Toucan's speedup. We can nonetheless say that, in such case, the Toucan variant was able to scale efficiently (1.91x, compared to the 1728 core run) with an overdecomposition factor of 4. No changes to the *slurm* job configuration were needed for the Toucan variant, illustrating the flexibility offered by our model.

E. Mpix_flowCart

Our final application is an unstructured mesh code called mpix_flowCart [2]. This code, developed by NASA Ames and NYU, is used in aerospace design, and has hundreds of users. Our intent in studying this production code is to demonstrate that Toucan can handle a non-trivial application. The version we used (dated 7/30/2014) for the *MPI-Original* variant solves the compressible Euler equations to compute steady flow. We choose to model the flow for the *OneraM6* wing[7], a well-known standard test case.

[7] http://www.onera.fr/

There is no available hand-coded overlap variant for mpix_flowCart. Given the complexity of the code, developing such a variant is daunting and for this reason we selected mpix_flowCart to highlight Toucan's potential to hide communication in large and complex applications with modest effort. The Toucan variant is MPI-Original annotated with 16 directives (4 superblocks).

Mpix_flowCart uses a Cartesian multi-level embedded boundary mesh with cut cells located where the mesh intersects the geometry (the wing's surface), and represents the mesh hierarchy using a data structure that consists of an array of faces that point to an array of cells. It partitions the mesh hierarchy into one subdomain per rank at runtime, using a space-filling curve to order the meshes. This reordering maintains good cache locality and performance of MPI-NoComm (in TFlops/per core) is largely independent of the number of ranks used.

The solver combines an explicit Runge-Kutta time stepping method with a recursive multigrid acceleration scheme that iterates over the coarser meshes. The coarser meshes are also ordered with a space filling curve and are partitioned into subdomains and generally exhibit good locality with their parent meshes. The application uses the standard technique of adding a layer of ghost cells (1 cell deep) around the local subdomain, which are exchanged with neighboring subdomains after every step.

One challenge we encountered in mpix_flowCart was the use of global and static variables. Since Toucan maps several ranks into a single process, each such rank sees global and static variables as shared storage, which is not the case in a conventional MPI implementation. This is is a well-known problem, and was encountered by AMPI [27]. We have not used any of the solutions employed by AMPI since they are architecture-dependent and compiler support may not be available in all platforms. Our current approach is to perform a manual thread-wise privatization of global variables by moving them into a structure that is allocated individually by each subrank at the beginning of execution and is passed as argument between subroutines. A cleaner approach may be to use C++'s thread-local storage.

We configured mpix_flowCart to run for 10 iterations and 4 multigrid levels with a CFL number of 1.2. We conducted a strong scaling study using a 75M cell mesh and ran our tests on 768, 1536, 3072, and 6144 cores. We show our results in Fig. 8. On 6144 cores, Toucan obtained a 1.14x speedup and was able to hide 33% off communication cost.

Of the 4 applications we tested, mpix_flowCart benefited the least from overlapping communication with computation. We conjecture that this effect was due to the irregularity in mpix_flowCart's communication patterns. The other applications have uniform patterns of communication. By comparison, ranks in mpix_flowCart have varying numbers of communicating neighbors, and the message lengths vary among the neighbors. To understand the impact of non-uniform communication, we ran mpix_flowCart with just one mesh and no coarser multigrid meshes. Running with multiple levels increases the irregularity of the communication, since communication between levels involves asymmetric ghost cell update patterns. Indeed, our results showed an improvement in Toucan's capability to hide communication. We plan to investigate whether a different strategy for sending messages of varying lengths will help regularize communication traffic.

V. Discussion and Related Work

A. Discussion

In this paper, we presented a source-to-source translator, called Toucan, and showed that it reduced communication costs significantly in 4 applications representing 3 common HPC application motifs. Toucan hid 46% to 59% of the communication in a structured grid method (Jacobi3D), 55% to 85% in a dense linear algebra method (Cannon2D) 60% to 72% and 33% to 55% in two unstructured grid methods (LULESH and and mpix_flowCart, respectively). The ability to hide communication depends on a variety of factors, including the algorithm's floating point intensity (the ratio of the total number of flops executed to memory words accessed), data locality and the architecture. We compared Toucan against AMPI, and in preliminary results, found that while AMPI was able to hide small amounts of communication, Toucan was able to hide more. Unlike Toucan, AMPI requires no programmer annotations and it supports workload migration and checkpoint based resilience. Perhaps these capabilities are factors in the performance differences we observed, and a broader study on a wider range of applications and architectures may help illuminate the reasons why.

As mentioned previously, the design of Toucan was based on experiences with of our earlier Bamboo [20] translator. The principal difference was to replace Bamboo's static code relocation and inlining strategy for supporting data-driven execution with a dynamic approach that avoids static code reorganization. Our dynamic approach has 3 advantages over static code reorganization:

1) A dynamic strategy can handle recursion while an in-line expansion strategy cannot. We needed this capability to translate mpix_flowCart.
2) Compared with in-line expansion, Toucan incurs only modest code expansion. In-line expansion requires that all MPI calls be lifted up through the static call chain to the function scope of $main()$, resulting in severe code bloat and unacceptably long translation times on source code comprising just a few thousand lines.
3) Toucan supports incremental code development since any un-annotated code will be left unmodified by the translator, including MPI calls. Thus, the user can focus their effort on code critical to performance, and

other modules (such as initialization) need not need be annotated, conserving development time.

We are currently investigating ways of extending Toucan's annotation grammar to accommodate application-specific heuristics. MATE's runtime system supports more general region dependency logic than the one exposed by our Toucan annotation grammar, since the latter was customized to a common MPI computation/communication pattern. This generality should be useful in applications with different parallel control flow structures, such as in multi-physics applications, where separate simulation components interoperate. A more general interface could provide a way to define finer-grained regions that can be further accommodated into the pipeline during communication operations, exposing new opportunities for overlap. For example, a superblock could also contain separate *pack* and *unpack* regions that run independently from their containing *send* and *compute* regions, respectively in our code example in Fig. 1.

B. Related Work

Realizing overlap has generally followed 3 approaches: (1) libraries, (2) new language constructs together with a compiler and (3) compiler-based optimizations.

MPI/SMPSs [19] is a hybrid approach that applies the SPMSs model [24] to MPI C/C++ applications. Like Bamboo, SMPSs uses a directive-based interface to specify data-flow dependencies between functions. Whereas MPI/SMPSs identifies task parallelism at the function call level, Toucan is able to extract and define dependencies among code regions within a function. Toucan uses domain specific knowledge about MPI calls and relies on overdecomposition rather than explicit heterogeneous parallel control flow via MPI processes and OpenMP threads in MPI/SMPSs. MATE subranks run as user-level threads, and manage data motion explicitly via memory to memory copying. Overdecomposition avoids hybrid control flow and also improves latency hiding.

To support overdecomposition, Toucan's runtime system takes a similar approach to FG-MPI [17], executing MPI ranks as user-level threads. Although this enables FG-MPI to realize some overlap, MATE is able to further expose overlap opportunities by providing an even finer-grained out-of-order execution of code regions within subranks.

Adaptive MPI (AMPI) [12] is an implementation of MPI that supports overdecomposition. It is implemented on top of the Charm++ model [15]. AMPI instantiates MPI ranks as a set of user-level threads. Each thread runs until a blocking message request is reached. At this point a continuation is created, which will run once the data becomes available. Overdecomposition is realized by creating more threads (*virtual* ranks) than available cores. In addition to hiding latency, AMPI provides other features such as load-balancing and checkpointing that are supported by Charm++'s runtime system, which Toucan does not currently support.

The *Delta Send-Recv* model [4] uses a compiler to statically transform an MPI program based on an analysis of application data dependencies and communication patterns. The translator requires that the user first convert calls to MPI primitives into the delta send-recv format. In contrast, a Toucan annotated program remains a legal MPI program. The programmer introduces Toucan-specific annotations. These annotations can be ignored by a conventional compiler and thus do not change the semantics of the original source. The Delta Send-Recv work demonstrated improvements that increase performance by a factor of 2 [4]. Since communication overlap cannot improve performance by more than a factor of 2, it appears that the translator is optimizing other aspects of program behavior, perhaps memory locality.

Pellegrini *et al.* [23] describe a compiler that performs data-dependency analysis at the statement level. It does not rely on overdecomposition. Toucan performs a coarser grain analysis on user-annotated groups of statements and it relies on overdecomposition to improve pipelining of communication and computation. Results are provided for a single application, a 5-point 2D stencil method, using a small mesh running on up to 512 cores (32K unknowns per core). Toucan has been demonstrated on production software on thousands of cores and our 3D stencil application ran with 1.2M unknowns per core, where the communication overheads are expected to be lower than the 2D problem, as are the benefits due to overlap.

Similar techniques are explored in Das *et al.* [9], who use SSA based use-def analysis, and Danalis *et al.*, [8] who manually restructure code based on a domain specific translation algorithm that employs knowledge about the MPI API. Unlike Toucan, neither technique relies on overdecomposition and experiments were conducted on benchmarks only and at small scales of parallelism. In addition to moving MPI calls to improve performance, Danalis *et al.* employ strip mining and other classic locality optimizations. Thus, the benefit of the scheme includes not only improvements in communication overlap but also memory locality. Indeed, some results show an improvement exceeding the upper bound achievable through communication overlap.

VI. Conclusions and Future Work

This paper introduced a new approach for translating annotated MPI programs for communication/computation overlap. We demonstrated that Toucan can hide significant communication, requiring only modest programmer annotation. We verified that Toucan is able to meet and exceed the communication-hiding capabilities and speedups of manually optimized code on up to 24K cores of the Edison platform. We foresee that performance improvement from using Toucan in terms of speedup will become even larger in future systems, where communication is expected to dominate the running time of scientific applications. In terms

of code complexity, both Toucan and the MATE runtime system are lightweight and compact: just 4000 lines of C++.

To date, collectives have not played a role in performance in applications we have tested with Toucan. However, applications such as the FFT and sorting have move large amounts of data collectively, requiring further attention. We have not yet explored systems that provide hardware support for collectives.

We are also exploring techniques for integrating global memory into our model to alleviate the cost of communication among subranks in the same process. For example, to implement the data transport for the runtime services, we could use a language or library supporting global address space communication such as UPC++ [28] or GASNet [5].

Acknowledgments

We would like to thank our anonymous reviewers for their insightful comments and suggestions. This research was supported by the Advanced Scientific Computing Research office of the U.S. Department of Energy under contracts No. DE-FC02-12ER26118 and DE-FG02-88ER25053. Sergio Martin was supported in part by the Fulbright Foreign Student Program grant from the U.S. Department of State, and a scholarship from Universidad Nacional de La Matanza, Departamento de Ingeniería e Investigaciones Tecnológicas. Scott Baden dedicates his contributions to this paper to the memory of Lillemor Nilsson (1937-2016).

References

[1] D.A. Adams. *A Computation Model with Data Flow Sequencing*. Stanford University, 1968.

[2] M. Aftosmis, M. Berger, and G. Adomavicius. “A parallel multilevel method for adaptively refined Cartesian grids with embedded boundaries”. In: *AIAA’00*.

[3] Scott B. Baden and Daniel Shalit. “Performance Tradeoffs in Multi-tier Formulation of a Finite Difference Method”. In: *ICCS 2001*.

[4] B. Bao et al. “Delta Send-Recv for Dynamic Pipelining in MPI Programs”. In: *CCGrid 2012*.

[5] D. Bonachea. *GASNet Specification, v1.1*. Tech. rep. UCB/CSD-02-1207. EECS Department, University of California, Berkeley, 2002.

[6] L. E. Cannon. “A Cellular Computer to Implement the Kalman Filter Algorithm”. PhD thesis. 1969.

[7] P. Colella. *Defining Software Requirements for Scientific Computing*. 2004.

[8] A. Danalis et al. “MPI-aware Compiler Optimizations for Improving Communication-computation Overlap”. In: *ICS ’09*.

[9] D. Das et al. “Compiler-controlled extraction of computation-communication overlap in MPI applications”. In: *IPDPS 2008*.

[10] *Edison Design Group, Inc. - C++ Front End.* URL: https://www.edg.com/docs/edg_cpp.pdf.

[11] R. D. Hornung, J. A. Keasler, and M. B. Gokhale. *Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory*. Tech. rep. LLNL-TR-490254.

[12] C. Huang, O. Lawlor, and L. V. Kalé. “Adaptive MPI”. In: *LCPC’04*.

[13] S. Jacob and S. B. Baden. “Hiding Communication Latency with Non-SPMD, Graph-Based Execution”. In: *ICCS’09*.

[14] L. V. Kalé. “The virtualization model of parallel programming: Runtime optimizations and the state of art.” In: *LACSI’02*.

[15] L. V. Kalé and S. Krishnan. “CHARM++: A Portable Concurrent Object Oriented System Based on C++”. In: *OOPSLA ’93*.

[16] Laxmikant V. Kalé. “The Virtualization Model of Parallel Programming : Runtime Optimizations and the State of Art”. In: *LACSI’02*.

[17] H. Kamal and A. Wagner. “FG-MPI: Fine-grain MPI for multicore and clusters”. In: *IPDPSW’10*.

[18] I Karlin, J. Keasler, and R. Neely. *LULESH 2.0 Updates and Changes*. Tech. rep. LLNL-TR-641973.

[19] V. Marjanović et al. “Overlapping Communication and Computation by Using a Hybrid MPI/SMPSs Approach”. In: *ICS ’10*.

[20] T. Nguyen. “Bamboo: Automatic Translation of MPI Source into a Latency-Tolerant Form”. PhD thesis. 2014.

[21] T. Nguyen et al. “Bamboo - Preliminary scaling results on multiple hybrid nodes of Knights Corner and Sandy Bridge processors”. In: *WOLFHPC ’13*.

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

[23] S. Pellegrini, T. Hoefler, and T. Fahringer. “Exact Dependence Analysis for Increased Communication Overlap”. In: *EuroMPI’12*.

[24] J. M. Perez, R. M. Badia, and J. Labarta. “A dependency-aware task-based programming environment for multi-core architectures”. In: *CLUSTER’08*.

[25] D. Quinlan. “ROSE: Compiler Support for Object-Oriented Frameworks”. In: *CPC2000*.

[26] L. I. Sedov. *Similarity and Dimensional Methods in Mechanics*. 1959.

[27] G. Zheng et al. “Automatic Handling of Global Variables for Multi-threaded MPI Programs”. In: *ICPADS’11*.

[28] Y. Zheng et al. “UPC++: a PGAS extension for C++”. In: *IPDPS’14*.