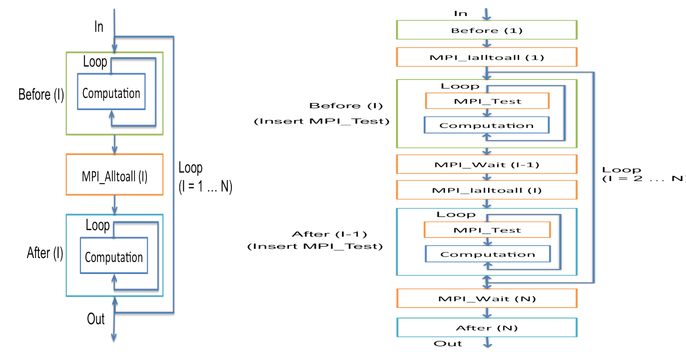
Compiler-Assisted Overlapping of Communication and Computation in MPI Applications

Jichi Guo,† Qing Yi,† Jiayuan Meng,‡ Junchao Zhang,‡ Pavan Balaji‡
University of Colorado Colorado Springs†
Colorado Springs, CO, USA
{jguo2, qyi}@uccs.edu
Argonne National Laboratory‡
Argonne, IL, USA
meng.jiayuan@gmail.com, {jczhang, balaji}@anl.gov

***Abstract*—The performance of distributed-memory applications, many of which are written in MPI, critically depends on how well the applications can ameliorate the long latency of data movement by overlapping them with ongoing computations, thereby minimizing wait time. This paper presents a study of the various optimization techniques to enable such overlapping in large MPI applications and presents a framework that uses an analytical performance model and an optimizing compiler to systematically enable a majority of such optimizations. In particular, we first generate an analytical performance model of the application execution flow to automatically identify potential communication hot spots that may induce long wait time. Next, for each communication hot spot, we search the execution flow graph to find surrounding loops that include sufficient local computation to overlap with the communication. Then, blocking MPI communications are decoupled into nonblocking operations when necessary, and their surrounding loop is transformed to hide the communication latencies behind local computations. We evaluated our framework using 7 MPI applications from the NAS benchmark suite. Our optimizations can attain 3% to 72% speedup over the original implementations.**



(a) Before optimization (b) After optimization

Fig. 1: Structure of NAS FT (1D layout) before (a) and after (b) overlapping computation and communication. *Before* and *After* in the figures are computation loops before and after the communication.

I. Introduction

As computing platforms migrate towards clusters of increasingly larger scale of microprocessors, applications need to manage the distributed memories of the processors via explicit message-passing runtimes, for example, MPI, to attain high performance. The relative latency and bandwidth of the communication network in relation to the compute capacity of processors are often hard to predict a priori and may change dramatically from one system to the next. Even on supercomputers comprised of homogeneous nodes, system noise is increasing on each node because of aspects such as power management, deeper memory hierarchies, and sharing of hardware such as caches and network. The "equal work means equal time" paradigm is no longer relevant on most systems, and load imbalance increasingly becomes the common scenario even on applications that are symmetrically structured. Consequently, bulk-synchronous communication, where all processes synchronize frequently, is no longer a valid option for high performance MPI applications. Application performance is often critically determined by its ability to flexibly overlap communications with local computations, thereby minimizing wait time.

This paper aims to automatically enable the use of nonblocking latency-hiding techniques to overlap local computation with remote communication in MPI applications, thereby enhancing their overall efficiency and performance portability. To illustrate the optimization, Figure 1a shows the structure of the NAS FT benchmark [3], which applies fast Fourier transform (FFT) to a 3D matrix through a loop that interleaves the computation of scaling the input matrix with a collective communication of MPI_Alltoall to exchange data among the processes. This is then followed by a final transposition of the resulting matrix. The clear separation of computation and communication phases makes the algorithm design easy to implement and maintain. Additionally, the communication buffers can be reused across different loop iterations, saving memory. However, the blocking MPI communication requires that all processes wait while the MPI_Alltoall operation is in progress. Consequently, unless the application is executed on a platform with the fastest network connections, its performance is likely to suffer because of the excessive wait time.

Figure 1b illustrates how the structure in 1a may be modified to better overlap computation with communication. In particular, the MPI_Alltoall operation is decoupled into two finer-grained operations: a nonblocking MPI_Ialltoall and a blocking MPI_Wait. Then, the loop is modified so that Before(i), which multiplies a local matrix with a time-evolution array and then saves a transpose of the matrix into a local buffer to be communicated to other processors, and MPI_Ialltoall(i), which exchanges the local transposes among different processes, are essentially moved so that they are evaluated before $MPI\_Wait(i-1)$, which waits for the completion of MPI communication of the previous iteration, and $After(i-1)$, which processes the just received remote data (of the $i-1$th iteration), and then print the result into an output file. By using two distinct buffers to store the data used in consecutive MPI communications, the output dependencoes between After(i-1) and Before(i)/MPI_Ialltoall(i) can be eliminated, thereby guaranteeing the correctness of optimization.

Finally, *MPI_Test* operations are inserted into the local computation to ensure the progress of the nonblocking communications.[1] By overlapping the MPI communications with the local computations, the transformed code allows the application to perform well even on systems with slow network connections, although nonblocking communications generally take longer time to finish than blocking ones, and more memory may be needed to hold the data during communications.

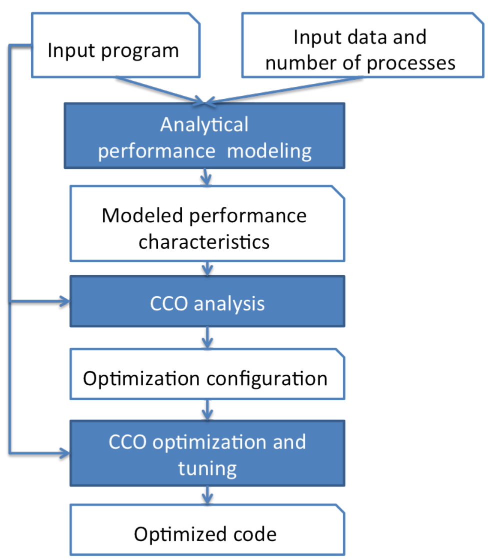


Fig. 2: Optimization Workflow

Figure 2 shows our workflow for systematically enabling communication-computation overlapping (CCO) in large MPI applications and thereby enhancing their performance portability. The workflow contains three key components: (1) the *performance modeling* component, which analyzes the runtime statistics of an MPI application to extract a *B*ayesian execution tree [15] representation of its execution flow, including the frequencies of various runtime code paths and their performance characteristics such as computation intensities, working set sizes, and communication characteristics of MPI operations; (2) the *CCO analysis* component, which identifies *h*ot computation and communication regions that are likely to benefit from the optimization and summarizes the result of profitability and safety analysis of the optimizations; and (3) the *CCO optimization and tuning* component, which applies the appropriate program transformations by replacing the blocking MPI operations with nonblocking ones, by reordering the computations and communications involved and by inserting MPI_Test operations with a frequency determined by empirical tuning of the optimized code. Note that in order to guarantee the profitability of the optimization, any communication slowdown from the use of the nonblocking operations must be fully overlapped with the local computation, and the insertion of MPI_Test operations should cause only marginal slowdown of the local computation so that its effect is insignificant when compared with the reduction of the original communication time. Our framework currently uses empirical tuning of the optimized code to select appropriate optimization configurations and to skip nonprofitable optimizations.

The idea of overlapping computation and communication in MPI applications have been well studied in the past [7], [12], including both using analytical performance models [20] and using compiler analysis [8] to assist the optimization. Similar to other existing work, we also manually applied the optimizing transformations for each application. Our work is unique in that it fully integrates both the performance modeling and compiler analysis to collectively determine both the profitability and safety of the overlapping optimization. Additionally, it focuses on a special inter-procedural pattern of loop-based communication-computation overlapping in scientific applications, which has not been addressed by existing literature. Compared to using profiling to directly measure and identify code blocks to optimize, our analytical modeling framework is able to collectively consider all the dynamic paths through each code block, to more accurately identify important inter-procedural communication patterns. Our main contributions are the following:

- We present a framework that integrates analytical performance modeling of large MPI applications, semantic inlining of developer-supplied domain knowledge, and pattern-based transformation in order to systematically enable better overlapping of MPI communications with independent local computations to enhance the performance portability of MPI applications. We currently manually applied the necessary program transformations (the last stage of the optimization) but expect to automate this step in our future work.
- We applied our approach to optimize 7 NAS Parallel Benchmarks (NPB) applications on both a high-speed and a slow network-connected cluster environment and achieved 3–88% speedup on both platforms.

[1] Although MPI communications do not need full usage of CPU, they need some CPU time, e.g., to manage communication progress, which is supplied only when operations such as MPI_Test and MPI_Wait are invoked.

The remainder of the paper is organized as follows. Section II presents our analytical performance modeling component for automatically identifying communication and computation hot spots in MPI applications Section III discusses how to determine the safety of computation-communication overlap through optimization analysis. Section IV summarizes strategies we used to perform the actual optimizations and the tuning of their configurations. Section V evaluates our framework using 9 NAS application benchmarks [3]. Section VI discusses related work, and Section VII presents our conclusions.

II. Analytical Modeling Of MPI Applications

To effectively reduce the overhead of network communications in MPI applications, one must understand when and where it becomes beneficial to enhance the overlap of communications with local computations in these applications, In particular, through an analytical approach, our framework aims to model the runtime execution flows of an input application in terms of their relative amount of time spent in local computations and network communications. This information then is used to automatically identify potential communication bottlenecks as candidates for optimization in the later steps.

To represent and estimate the time required to execute the local computation of each path, we use the `Bayesian Execution Tree` (`BET`) from the Skope analytical performance modeling framework [15]. Each BET essentially represents possible runtime code paths of an application together with their execution frequency and expected execution time. We use the Skope framework to automatically generate a BET representation of each application from the application source code combined with some sample input data and code-coverage profiling of the application execution. We then extend the Skope framework to additionally estimate the overhead of each MPI communication through the following steps.

1) Use LogGP-based communication model for the MPI runtime to estimate the communication time for each individual MPI call.
2) Statistically estimate the expected average communication time for each code path by combining the individual communication with the execution frequencies.

The balance between the time required for each MPI communication and the expected execution time of its surrounding local computation is used to project optimization opportunities. The following first illustrates the BET representation that we inherit from [15] and then explains our extensions for modeling MPI communications.

A. Bayesian Execution Tree

Figure 3 shows an example `BET` for one of the MPI processes of the 1D FFT benchmark in Figure 1a. Each node of the BET represents a code block (a sequence of statements in the user program) together with its runtime *execution frequency*, defined as the expected average number of times that statements in the node block will be executed at runtime. A `depth-first-traversal` (`DFS`) of each subtree of the `BET` corresponds to a possible runtime execution path of the

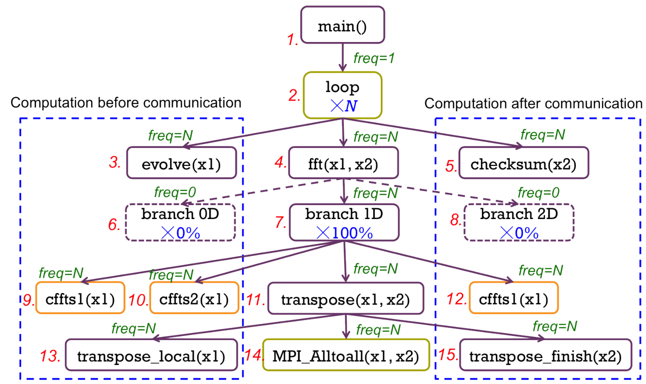


Fig. 3: Simplified Bayesian Execution Tree for NAS 1D FFT before overlapping computation and communications *(for simplicity, only important branches, loops, and function calls are shown, and not all nodes and frequencies in BET are drawn in this figure)*

statements. For example, in Figure 3, Node#2 is a loop of N iterations, so the frequency of its loop body is N. Node#7 is a branch inside *fft* function. If the application is to perform 1D FFT, this branch is taken 100% of time during execution, so its frequency is N, while the frequency of the alternative branches (Node#6 and Node$\hat{8}$) are set to 0.

In order to derive execution frequencies of each code block, the Skope framework requires a description of the application input data, either manually provided by the user or automatically collected via an instrumented run of the application. The input data description characterizes the possible values of all the data that the application may obtain from external sources, for example, command-line arguments, environment variables, or files. For array variables, only their dimensions and the size of dimension are required. For MPI applications, the total number of MPI processes (`MPI_Comm_size`) and the rank of the process to model (`MPI_Rank`) are additionally required. Based on the input data description, the Skope framework applies constant propagation to derive possible values of the expressions that control the directions of branch and loop controls. A fall-through probability to be 50% is assumed if the values cannot be accurately determined. For this paper, we used `gcov` to profile applications with sample input data,

B. Modeling MPI Communications

To predict the MPI communication overhead, we have extended the Skope framework with the LogGP model [2] to additionally model the latency (elapsed time) of each MPI operation using the following four parameters:

1) P: number of processes involved in the communication
2) n: size (in bytes)of the message being transferred
3) $alpha$: overhead of starting each message and time interval required between transmitting each pair of messages
4) $beta$: expected communication time per byte for large messages, determined by the underlying network bandwidth.

Among the four parameters, $alpha$ and $beta$ can be calculated from characteristics of the underlying network. In particular, we compute $beta$ as the reciprocal of the network bandwidth and $alpha$ by using microbenchmarks to measure the latency of `MPI_Send` and `MPI_Recv` operations on the target platform. The other two parameters, P and n, are either determined through instrumented runs of the user application or obtained directly from the user as expected runtime configurations of the application. In particular, P equals to `MPI_Comm_size`; and n (i.e., message size) is obtained from the values used to invoke the MPI operations.

Following the LogGP model, we model MPI point-to-point communications as:

$$cost_{p2p}(n; alpha, beta) = alpha + n \cdot beta. \tag{1}$$

To model the `MPI_Alltoal` operation, we use the following two formulas.

$$cost_{short} = logP \cdot alpha + \frac{n}{2} \cdot logP \cdot beta \tag{2}$$

$$cost_{long} = (P-1) \cdot alpha + n \cdot beta \tag{3}$$

The first formula models the communication time of short messages and the second that of long messages. We use values of *control variables* from the MPI runtime library, for example, `MPIR_CVAR_ALLTOALL_SHORT_MSG_SIZE` for MPI alltoall, to determine whether a message should be categorized as *short* or *long* and thereby select the proper formulas to use.

After estimating the execution time of each individual MPI operation, the overall communication time of a code path in BET can be calculated by adding the communication time of all code blocks along the path, using the following formula.

$$cost_n = \sum_{i}^{n} cost(i) * freq(i) \tag{4}$$

Specifically, the total communication cost of a path of n nodes in the BET can be computed as the sum of the communication time of each individual MPI operation multiplied by its execution frequency $freq(i)$. Here, $freq(i)$ is calculated by using BET as one of the properties of the BET node that contains the MPI operation, and $cost(i)$ is calculated as indicated above using LogGP formulas instantiated with the expected parameter values of the MPI operations. For example, the total communication time of `MPI_Alltoall` in Figure 3 can be computed by multiplying the average communication time of `MPI_Alltoall` by the number of iterations of the loop node#2 ($\times N$) and the fall-through probability of the 1D FFT branch node#7 ($\times 100\%$).

III. Optimization Analysis

The objective of our optimization analysis is to automatically identify which MPI communications to optimize and what local computations can be safely overlapped with the communication, through the following three steps.

1) Analytically identify MPI operations that are potential performance bottlenecks based on the modeling of communication cost and the execution flow modeling of the entire application described in Section II. In particular, based on the BET representation of the user application, this step identifies the top N most time-consuming MPI *calls*, which take more than $P\%$ of the overall communication time, where both N and P are user-configurable parameters and were set by default with $N = 10$ and $P = 80$. The selection is accomplished by simply sorting the pre-estimated communication time of all MPI calls in the `BET` and then selecting the top ones. For example, for the NAS FT application shown in Figure 3, a single MPI call, the `MPI_Alltoall` at the bottom of the BET, is selected since it takes more than 95% of the overall communication time.
2) For each identified MPI communication to optimize, select a loop of computation that can be potentially overlapped with the communication to improve performance, by locating the closest enclosing loops of the MPI communication in the `BET`—for example, node#2 in Figure 3 for the NAS FT application, to potentially overlap with the communication. If the enclosing loop does not exist, the communication is given up as an optimization target.
3) Apply dependence analysis to check the safety of overlapping the selected computation and communication.

A key challenge in optimizing large applications is that MPI communications are often scattered across procedural boundaries and the computation that can be overlapped with them is often some distance away and similarly across abstraction boundaries, By using the Skope framework and through the BET representation of the whole user application, we are able to inter-procedurally select MPI communication hot spots as well as their surrounding local computations as potential optimization targets. Then, an optimization pragma, `#pragma cco do`, illustrated at line 1 of Figure 4, is inserted automatically before each selected code region to instruct the compiler to perform additional analysis to determine the safety of optimization.

We use standard loop dependence analysis within the ROSE C/C++/Fortran compiler [33] to automatically determine the safety of the reordering optimization to each selected code region and make the compiler inline all function calls within the region when possible; that is, when source code of the callee is available. For other function calls where inlining is not advisable to the compiler, we insert the following pragmas to provide additional guidance to the loop dependence analysis.

1) `#pragma cco ignore`, illustrated at line 3 in Figure 4, which is manually inserted before each function call that can be safely ignored when performing dependence analysis. That is, these function calls will not implicate the safety of any reordering optimization, when the function call is not reachable at runtime but

involves I/O statements for debugging purposes. Examples of such function calls include the $timer\_start()$ and $timer\_stop()$ in Figure 4.

2) `#pragma cco override`, illustrated at the first line of Figure 8 and 5, which defines the memory side effects of the following function call. The override definitions, if specified, allow dependence analysis to proceed across procedural boundaries without actual inlining of the procedure implementations. They are also inserted manually but could be automatically generated through the integration of advanced interprocedural side effect analysis [22].

The above annotations are currently manually inserted to overcome situations where traditional function inlining is insufficient to overcome the difficulty imposed by procedural boundaries, either because the source code of the callee is unavailable or the low-level implementation details of the callee are too complex to be accurately deciphered by traditional compiler dependence analysis, which is a known issue in compiler analysis [16]. Figure 8 shows an example override definition for `MPI_Alltoall` in NAS FT, where we use the $read$ and $write$ pseudo statements to indicate read and write memory accesses. Based on the domain knowledge of the application that send and receive data have atomic types instead of user-defined types, its memory side effect can be expressed as data accesses to consequent memories in source and target data. Figure 5 shows another example of the $fft()$ function in NAS FT. The original function have several branches for different data layout, while the override definition has only 1D layout that is the target code path to optimize.

We manually override function inlining according to the following criteria:

- The definition of the function is not available or contains too many low-level implementation details that are likely to overcomplicate the inlined code. For example, we manually write a memory side effect definition for all MPI function calls.
- The runtime code path of the function call allows the side effects of the invocation to be simplified through specialization far more than if automatically determined after inlining. For example, in NAS FT, the procedure `fft` has 6 branches for solving different dimensions of the FFT problems (0D, 1D, or 2D), while only one branch will be taken for each test. By manually overriding the original definition, we can eliminate the unreachable branches.
- When the same array data are declared with different dimensions in the caller and callee, the manual override definition can normalize data accesses by converting linearized array accesses to easier-to-analyze coordinates.

IV. Program Transformation

After selecting the code regions to optimize, we currently manually transform the source code to systematically enable the overlapping of computation and communication through the following steps.

```
!$cco do
do iter = 1, niter
  !$cco ignore
  if (timers_enabled) call timer_start(T_evolve)
  call volve(u0,u1,twiddle,dims(1,1),dims(2,1),dims(3,1))
  !$cco ignore
  if (timers_enabled) call timer_stop(T_evolve)
  !$cco ignore
  if (timers_enabled) call timer_start(T_fft)
  call fft(-1,u1,u2)
  !$cco ignore
  if (timers_enabled) call timer_stop(T_fft)
  !$cco ignore
  if (timers_enabled) call timer_start(T_checksum)
  call checksum(iter,u2,dims(1,1),dims(2,1),dims(3,1))
  !$cco ignore
  if (timers_enabled) call timer_stop(T_checksum)
end do
```

Fig. 4: Source code with directives of the loop in NAS FT to optimize

```
$cco override
subroutine fft(dir, x1, x2)
  cffts1(-1,dims(1,3),dims(2,3),dims(3,3),x1,x1,scratch)
  transpose_x_yz(3, 2, x1, x2)
  cffts2(-1,dims(1,2),dims(2,2),dims(3,2),x2,x2,scratch)
  cffts1(-1,dims(1,1),dims(2,1),dims(3,1),x2,x2,scratch)
end subroutine
```

Fig. 5: 1D layout code path to override the original $fft()$ definition

A. Function outlining

Given a loop to optimize, we first outline the computation and communication inside the loop into separate functions, in order to make it easier to replicate and reorder them later into different loop iterations. In particular, we divide the statements at each iteration I of the target loop into the MPI communications at iteration I (Comm(I)), the computation (Before(I)) that should run before Comm(I), and the computation (After(I)) to evaluate after Comm(I) Each group of statements is then outlined into a separate procedure, with the loop index variables as its function parameters.[2] Take NAS FT in Figure 1a as an example. The loop to optimize is divided into $Comm(I)$, the MPI communication at iteration I; $Before(I)$, the computation before communication at iteration I; and $After(I)$, the computation after communication at iteration I.

B. Converting MPI communications

Each blocking MPI operation, for example, *alltoall* collectives and point-to-point send-receives, is converted to an equivalent nonblocking communication combined with a blocking wait. For example, in Figure 1a, the outlined communication function $Comm(I)$, which invokes $MPI\_Alltoall$ internally, is replaced by $Icomm(I)$ and $Wait(I)$, which are the corresponding nonblocking communication ($MPI\_Ialltoall$) and wait operations converted from $Comm(I)$, respectively.

[2]These components can alternatively be simply tagged as Comm(I), Before(I), and After(I) if the optimization were to be fully automated; outlining makes it easy to modify the code manually.

```
subroutine transpose_x_yz(l1, l2, xin, xout)
  call transpose2_local(dims(1,l1),
  > dims(2,l1)*dims(3,l1),xin,xout)
  call transpose2_global(xout,xin)
  call transpose2_finish(dims(1,l1),
  > dims(2,l1)*dims(3,l1),xin,xout)
end subroutine
```

Fig. 6: Source code of `transpose_x_yz`

```
subroutine transpose2_global(xin, xout)
  call mpi_alltoall(xin, ntdivnp/np, dc_type, xout,
  > ntdivnp/np, dc_type, commslice1, ierr)
end subroutine
```

Fig. 7: Original source code of `transpose2_global`

C. Reordering computation and communication

After the previous steps, the body of the loop to optimize now contains a sequence of specially named operations such as $Before(I)$, $After(I)$, $Icomm(I)$, and $Wait(I)$, where I is the loop index variable looping from 1 to N.

We then *interleave* the communication and computation operations of consecutive loop iterations, as illustrated in Figure 9, in two steps:

1) Move $Before(1)$ and $Icomm(1)$ to the outside before the first iteration of the loop starts, and move $Wait(N)$ and $After(N)$ outside after the last loop iteration as shown in Figure 9c.
2) Move $Before(I)$ and $Icomm(I)$ above $Wait(I-1)$ and $After(I-1)$ as shown in Figure 9d.

After the reordering, the nonblocking communication in the current iteration I (between $Icomm(I)$ and $Wait(I)$) can be processed in parallel with the computation in the previous ($After(I-1)$) and next ($Before(I+1)$) iterations.

D. Replicating the communication buffer

Each MPI operation needs a dedicated buffer to hold the data being communicated. Applications typically first allocate the necessary communication buffers at the initialization stage and then reuse the same buffers in the same MPI operations across different loop iterations. After applying our optimization, as illustrated in Figure 12, the communication ($Icomm(i)$ and $Wait(i)$) at each ith iteration, where $i \geq 2$, is overlapped with computation $Before(i+1)$ and $After(i-1)$. Assuming that two distinct buffers, ***InBuf*** and ***OutBuf***, are used for sending and receiving each message, respectively, each buffer needs to be replicated into a pair of equal size to ensure that distinct buffers are used across the overlapping iterations, as illustrated in Figure 10. In particular, we replicate each buffer by allocating additional memory outside the loop and then alternately use a distinct buffer in every pair of consecutive loop iterations.

E. Inserting MPI_Tests

When using nonblocking MPI operations, some CPU time needs to be allocated, by embedding MPI_Test calls in the local computation, to ensure continuous progress of the communications. If the local computation is not inside a loop, we insert one or more MPI_Test calls evenly distributed into the computation. On the other hand, if the local computation is inside a loop, we insert MPI_Test into the beginning of the loop body and use a conditional variable to adjust its frequency. The inserted code is illustrated in Figure 11. In both cases, the frequency of MPI_Test is empirically adjusted as the application is ported to each architecture.

```
$cco override
subroutine MPI_Alltoall(sendbuf, sendcount, sendtype,
  > recvbuf, recvcount, recvtype, comm, ierror)
  do i = 1, sendcount
    read sendbuf(i)
  end do
  do i = 1, recvcount
    write recvbuf(i)
  end do
end subroutine
```

Fig. 8: Memory side effect of $MPI\_Alltoall()$ with simple datatype to override its original definition

```
DO I = 1 .. N
  Before(I)
  Comm(I)
  After(I)
END DO
```

(a) Input loop

```
DO I = 1 .. N
  Before(I)
  Icomm(I)
  Wait(I)
  After(I)
END DO
```

(b) Decouple blocking comm

```
Before(1)
Icomm(1)
DO I = 2 .. N
  Wait(I - 1)
  After(I - 1)
  Before(I)
  Icomm(I)
END DO
Wait(N)
After(N)
```

(c) Move first and last iterations

```
Before(1)
Icomm(1)
DO I = 2 .. N
  Before(I)
  Wait(I - 1)
  Icomm(I)
  After(I - 1)
END DO
Wait(N)
After(N)
```

(d) Interleave consequent iterations

Fig. 9: Steps to reorder outlined communication and computation functions

V. Experiment Results

To evaluate the accuracy of our hot-spot communication predictions and the performance implications of our manually applied optimizations to better overlap computation with communication, we applied our approach to model and

```
Before(1, InBuf)
Icomm(1, InBuf)
DO I = 2 .. N
  Before(I, InBuf)
  Wait(I - 1)
  Icomm(I, InBuf
         , OutBuf)
  After(I - 1, OutBuf)
END DO
Wait(N, OutBuf)
After(N, OutBuf)
```

(a) Original communication that uses the same input and output Buf

```
Before(1, InBuf)
Icomm(1, InBuf)
DO I = 2 .. N
  Before(I, I % 2 ==  1 ? InBuf : InBuf2)
  Wait(I - 1)
  Icomm(I, I % 2 ==  0 ? InBuf : InBuf2
         , I % 2 ==  0 ? OutBuf : OutBuf2)
  After(I - 1, , I % 2 ==  1 ? InBuf : InBuf2)
END DO
Wait(N, N % 2 == 1 ? OutBuf : OutBuf2)
After(N, N % 2 == 1 ? OutBuf : OutBuf2)
```

(b) Replicate $Buff$ with $Buff2$ of the same size

Fig. 10: Replicate communication buffers for nonblocking communication

```
DO I = 1 ... L
  If I % Freq == 0
    MPI_Test
  Original_computation_statements
END DO
```

Fig. 11: Insert MPI_Test into the hot computation loop at specific frequency $Freq$

optimize 7 MPI applications from the NAS NPB [3] on two clusters shown in Figure I. The first Intel platform is a large high-performance computing cluster with very fast internode communication through InfiniBand. The second platform is in a small data center where the internode communication is through relatively slow Ethernet. Both clusters use MPICH 3.1.1 [14] as the underlying MPI library. Because our current analytical performance model cannot estimate intranode MPI communication time, we have allocated a single MPI process per node on both clusters.

For each MPI application, we first used our extended SKOPE performance modeling framework to find the most time-consuming MPI communication in the application. Then,

TABLE I: Experiment platforms

Server	Intel	HP ProLiant BL460c Gen6
	Intel Xeon	Intel Xeon
Instruction set	x86	x64
Frequency	2.6 GHz	3.2 GHz
Compiler	ICC/Ifort 13.1	GCC/Gfortran 4.4.7
Network	InfiniBand Qlogic QDR	1 Gbps Ethernet
Total nodes	301	24 on 3 racks
Max memory	64 GB	48 GB

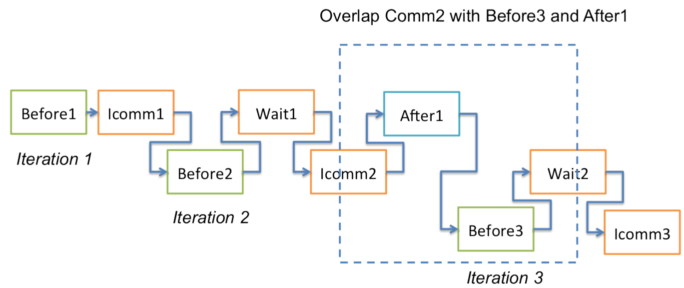


Fig. 12: Overlapped computation and communication operations

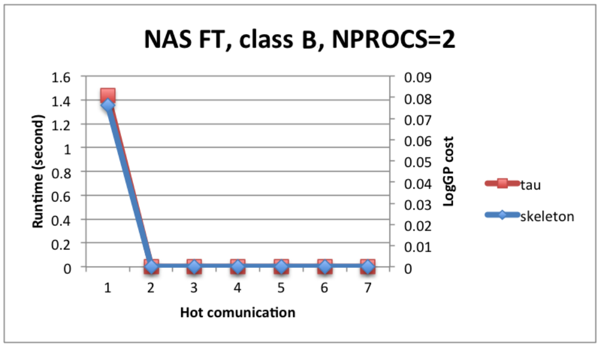
we manually applied the computation-communication optimization to the enclosing loop surrounding the identified communications. We measured the performance improvements from the optimizations using input data provided by the NPB benchmarks and using a range of 2 to 9 nodes for each application. Besides using the built-in timers within the NPB applications to collect their overall performance, we manually instrumented the source code of the applications to report the performance of individual communications.

A. Accuracy of Hot Communication Prediction

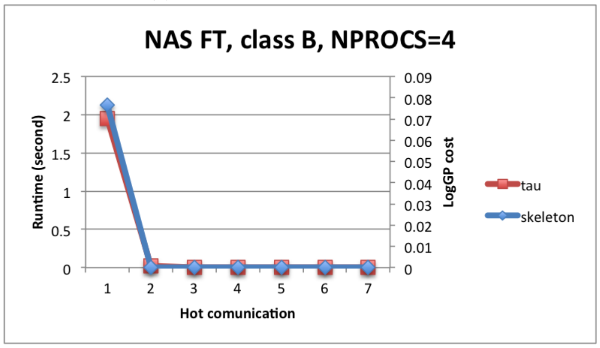
To evaluate the accuracy of our modeling of MPI communications, we compared the set of communication hot spots selected using our model with those found by profiling the NPB applications, using class B input on 4 nodes. The different hot-spot selection for the NPB applications are shown in Table II. When selecting the top time-consuming MPI communications, we required that their overall time be at least 80% of the application's overall communication time. In this case, our predictive modeling selected the same set of hot communications as found using application profiling. When asked to select a given number of the most time-consuming communications, however, the output by our predictive modeling differs by at most 2 selections compared with using profiling, for the NAS LU benchmark. Here the most time-consuming communications are pairs of sends/receives at four symmetric directions, which were estimated to take the same time by our predictive modeling. However, their actual runtime collected through profiling differ by 37%, because the execution of the processes is unbalanced, resulting in extra wait time to synchronize the corresponding MPI_Send/MPI_Recv operations. Figure 13 compares our projected communication time with the actual measured time for NAS FT, using two and four processors. Here in spite of the small error rates in projecting the absolute values of the communication time, our modeling framework was able to accurately capture the relative importances of the various communication operations.

B. Impact of Optimizations

Figures 14 and 15 show the speedups we attained by enabling better computation-communication overlapping for the NPB applications. In particular, for each benchmark, the performance of the original application and the optimized code



(a) Communication on 2 nodes



(b) Communication on 4 nodes

Fig. 13: Profiled runtime and modeled cost of NAS FT with middle-sized input (B) on x86 cluster

TABLE II: Differences between the projected hot-spot selection and the measured hot-spot selection based on profiling with 80% threshold for class B data on 4 nodes. Zero means the set of N hot spots equals the top N hot spots.

	Selected number of hot MPI communications							
	1	2	3	4	5	6	7	8
FT	0							
IS	0	0						
CG	0							
LU	0	1	2	2	1	1	0	0
MG	1	1	0	1	1	0		

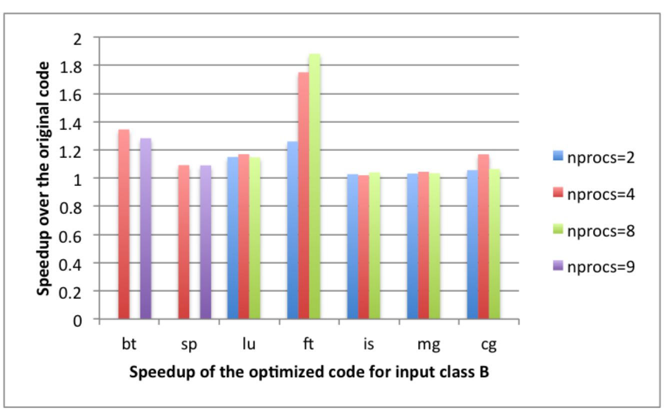


Fig. 14: Optimization speedups on the InfiniBand cluster.

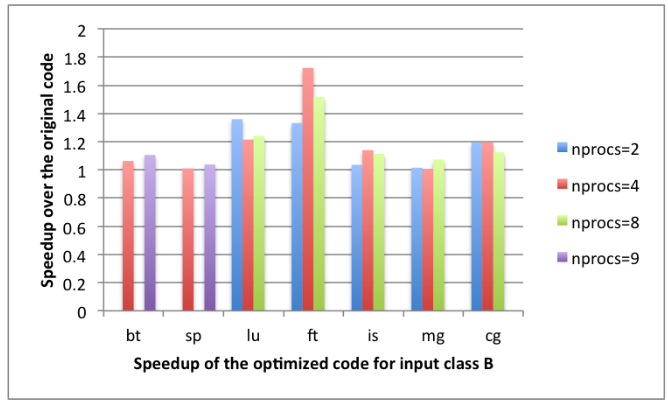


Fig. 15: Optimization speedups on the Ethernet cluster.

is measured by using input class B on 2, 4, 8, and 9 nodes, with one MPI process bound to each node, with the exception of NAS BT and SP, which require the number of processes to be a multiple of 3 and so are configured to run on 3 and 9 nodes only. The overall elapsed time of each application is measured by using NPB's built-in timer, which reports the elapsed time of multiple iterations with the exclusion of the initialization time.

Our optimization attained 3–88% speedup for all the NPB applications, with more significant speedups for FT and IS, which are the only two benchmarks that use *alltoall* collectives as the main communication operation, and less significant speedups for the other benchmarks, which mostly use point-to-point send/receives. The maximum 88% performance improvement is observed with the NAS FT benchmark, which uses a time-consuming $MPI\_Alltoall$ operation, enclosed inside the outermost loop of the application, to exchange a large amount of data. The lowest speedup (3%) is observed with NAS MG, which does not have sufficient local computation in the surrounding loop of the MPI communication to overlap with communication.

While our optimization was able to attain a consistent level of performance improvement on both platforms, the best speedups are observed at different runtime configurations. For example, the best speedup for NAS FT was attained when using 8 processors on the infiniband cluster but when using two processors on the Ethernet cluster, as the communication latency on the Ethernet is much longer than that of the Infiniband network, which in turn affects the amount of local computation time required to overlap with the communication. Overall, the possible speedup attained is bound by the latency of the communication being optimized and the amount of available local computation to overlap with the communication time. A larger amount of local independent computation is generally required to fully compensate the latencies of a slower network (e.g., Ethernet) than that of a faster network (e.g., infiniband).

VI. Related Work

In order to enhance the efficiency of MPI applications, existing work has focused mostly on communication mechanisms underlying the wide variety of MPI operations, for example, alternative protocols for point-to-point MPI communications [4], [9], collective operations [13], [26], [36], [37], remote direct memory access (RDMA) [17], [24], [40], load balancing of the operations [23], [28], and the elimination of redundant communications through software caching and the exploitation of data locality [6], [29], [38]. In contrast, our work focuses on application-level performance enhancement by enabling automated overlapping of MPI communications with independent local computations.

Iancu et al. [20] tried to automatically determine message sizes and schedules for MPI communications through an analytical model of system scale and load to avoid network congestion. Danalis et al. [8] investigated compiler optimizations to potentially automate the overlapping of MPI computations and communications, by formulating a set of data flow equations to describe the side effects of key MPI operations so that an MPI-aware compiler can automatically assess the safety of several optimizations, which were then manually applied in their paper. Various patterns of computation-communication overlapping and automated optimization schemes have also been discussed [7], [12]. In contrast, we present a systematic approach to enable a pattern of loop-based communication-computation overlapping in scientific applications, including automated identification of optimization opportunities and a semi-automated implementation to perform the optimizations using hot path analysis, dependence analysis, and empirical tuning to determine where and how to apply the optimizations.

To reason about the profitability of optimizing MPI applications, Sancho et al. [34] combined empirical tuning with networking models to quantify the potential benefit of overlapping communication and computation in large-scale scientific applications. Potluri et al. [31] empirically quantified the overlapping of MPI-2 operations in a seismic modeling application. Hu et al. [19], [35] identified the consumer-producer model from the control flow graph of the application to guide optimization decisions for overlapping Alltoall communication in a 3-D FFT. Didelot et al. [10], [11] developed a message progression model based on collaborative polling that allows an efficient autoadaptive overlapping of communication phases with computation. In this paper, we predicted the most time-consuming *hot* code path of the computation-communication patterns to optimize, using existing analytical model of communications [2].

Preissl et al. [32] summarized common communication patterns in MPI applications to enable automated optimization. Pellegrini et al. [30] proposed an exact dependence analysis approach for increasing the overlapping of computation and communication. Subotic et al. [39] speculatively extracted runtime data-flow to understand the dynamic dependence of the application. Aananthakrishnan et al. [1] used a hybrid static and runtime data-flow analysis of MPI programs. We also use dependence analysis to determine the correctness of optimization, enhanced with additional knowledge from developers about the MPI operations and runtime code paths within their applications.

In order to find the optimal placement of nonblocking MPI operations within the computation control flow, accurate modeling of the underlying computation and communication is required [5]. Hoefler et al. [18] presented an analytical approach to model MPI barriers. Ino et al. [21] presented a parallel computational model for synchronization analysis in MPI. Martinez et al. [25] developed an analytical model extending LogGP [2] for accurate estimation of individual MPI communication. Moritz and Frank [27] modeled network contention in MPI applications. In our optimization, we first reposition each pair of local computation and nonblocking communication as far apart as safety allows across different loop iterations and insert MPI_Test with empirically tuned frequencies into the local computation to ensure proper progress of the nonblocking communication.

VII. Conclusion

This paper presents a systematic approach to automate the overlapping of communications with local independent computations in large MPI applications, thereby enhancing their performance portability. Our optimization workflow starts with analytical performance modeling of the overall application execution flow to identify long-lasting MPI communications to overlap. Next, we conduct a semi-automatic safety and profitability analysis to find optimization opportunities. We complete the optimization by manually applying the necessary transformations in a systematic fashion that can be potentially automated. We applied our approach to optimize 7 NAS NPB applications on both a high-speed and a slow network-connected cluster environment. We achieved 3–88% speedup on both platforms.

References

[1] Sriram Aananthakrishnan, Greg Bronevetsky, and Ganesh Gopalakrishnan. Hybrid approach for data-flow analysis of mpi programs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 455–456, New York, NY, USA, 2013. ACM.

[2] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. Loggp: Incorporating long messages into the logp model—one step closer towards a realistic model for parallel computation. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 95–105, New York, NY, USA, 1995. ACM.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[4] Ron Brightwell and Keith Underwood. Evaluation of an eager protocol optimization for mpi. In *In Proceedings of EuroPVM/MPI*, pages 327–334, 2003.

[5] Ron Brightwell and Keith D. Underwood. An analysis of the impact of mpi overlap and independent progress. In *Proceedings of the 18th Annual International Conference on Supercomputing*, ICS '04, pages 298–305, New York, NY, USA, 2004. ACM.

[6] Darius Buntinas, Brice Goglin, David Goodell, Guillaume Mercier, and Stéphanie Moreaud. Cache-efficient, intranode, large-message mpi communication with mpich2-nemesis. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 462–469, Washington, DC, USA, 2009. IEEE Computer Society.

[7] Anthony Danalis, Ki-Yong Kim, Lori Pollock, and Martin Swany. Transformations to parallel codes for communication-computation overlap. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 58–, Washington, DC, USA, 2005. IEEE Computer Society.

[8] Anthony Danalis, Lori Pollock, Martin Swany, and John Cavazos. Mpi-aware compiler optimizations for improving communication-computation overlap. In *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pages 316–325, New York, NY, USA, 2009. ACM.

[9] Alexandre Denis. A high performance superpipeline protocol for infiniband. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 276–287, Berlin, Heidelberg, 2011. Springer-Verlag.

[10] Sylvain Didelot, Patrick Carribault, Marc Pérache, and William Jalby. Improving mpi communication overlap with collaborative polling. In *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface*, EuroMPI'12, pages 37–46, Berlin, Heidelberg, 2012. Springer-Verlag.

[11] Sylvain Didelot, Patrick Carribault, Marc Pérache, and William Jalby. Improving mpi communication overlap with collaborative polling. *Computing*, 96(4):263–278, April 2014.

[12] Lewis Fishgold, Anthony Danalis, Lori Pollock, and Martin Swany. An automated approach to improve communication-computation overlap in clusters. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 290–290, Washington, DC, USA, 2006. IEEE Computer Society.

[13] Richard L. Graham and Galen Shipman. Mpi support for multi-core architectures: Optimized shared memory collectives. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 130–140, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] William Gropp. Mpich2: A new start for mpi implementations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 7–, London, UK, UK, 2002. Springer-Verlag.

[15] Jichi Guo, Jiayuan Meng, Qing Yi, Vitali A. Morozov, and Kalyan Kumaran. Analytically modeling application execution for software-hardware co-design. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, May 19-23, 2014*, pages 468–477, 2014.

[16] Jichi Guo, Mike Stiles, Qing Yi, and Kleanthis Psarris. Enhancing the role of inlining in effective interprocedural parallelization. In *ICPP'11: International Conference On Parallel Processing (Accepted For Publication)*, Taipei, Taiwan, September 2011.

[17] Masayuki Hatanaka, Atsushi Hori, and Yutaka Ishikawa. Optimization of mpi persistent communication. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 79–84, New York, NY, USA, 2013. ACM.

[18] Torsten Hoefler, Lavinio Cerquetti, and Frank Mietke. A practical approach to the rating of barrier algorithms using the logp model and open mpi. In *Proceedings of the 2005 International Conference on Parallel Processing Workshops*, ICPPW '05, pages 562–569, Washington, DC, USA, 2005. IEEE Computer Society.

[19] Changjun Hu, Yewei Shao, Jue Wang, and Jianjiang Li. Automatic transformation for overlapping communication and computation. In *Proceedings of the IFIP International Conference on Network and Parallel Computing*, NPC '08, pages 210–220, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] Costin C. Iancu and Erich Strohmaier. Optimizing communication overlap for high-speed networks. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 35–45, New York, NY, USA, 2007. ACM.

[21] Fumihiko Ino, Noriyuki Fujimoto, and Kenichi Hagihara. Loggps: A parallel computational model for synchronization analysis. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 133–142, New York, NY, USA, 2001. ACM.

[22] K. D. Cooper K. Kennedy. Interprocedural side-effect analysis in linear time. *SIGPLAN Not.*, 23(7):57–66, June 1988.

[23] Vivek Kale, Amanda Randles, and William D. Gropp. Locality-optimized mixed static/dynamic scheduling for improving load balancing on smps. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 115:115–115:116, New York, NY, USA, 2014. ACM.

[24] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High performance rdma-based mpi implementation over infiniband. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, pages 295–304, New York, NY, USA, 2003. ACM.

[25] D. R. Martinez, J. C. Cabaleiro, T. F. Pena, F. F. Rivera, and V. Blanco. Accurate analytical performance model of communications in mpi applications. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[26] Anshul Mittal, Nikhil Jain, Thomas George, Yogish Sabharwal, and Sameer Kumar. Collective algorithms for sub-communicators. *SIGPLAN Not.*, 47(8):315–316, February 2012.

[27] Csaba Andras Moritz and Matthew I. Frank. Logpc: Modeling network contention in message-passing programs. *IEEE Trans. Parallel Distrib. Syst.*, 12(4):404–415, April 2001.

[28] Sun Nian and Liang Guangmin. Dynamic load balancing algorithm for mpi parallel computing. In *Proceedings of the 2009 International Conference on New Trends in Information and Service Science*, NISS '09, pages 95–99, Washington, DC, USA, 2009. IEEE Computer Society.

[29] David Ozog, Sameer Shende, Allen Malony, Jeff R. Hammond, James Dinan, and Pavan Balaji. Inspector/executor load balancing algorithms for block-sparse tensor contractions. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 483–484, New York, NY, USA, 2013. ACM.

[30] Simone Pellegrini, Torsten Hoefler, and Thomas Fahringer. Exact dependence analysis for increased communication overlap. In *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface*, EuroMPI'12, pages 89–99, Berlin, Heidelberg, 2012. Springer-Verlag.

[31] Sreeram Potluri, Ping Lai, Karen Tomko, Sayantan Sur, Yifeng Cui, Mahidhar Tatineni, Karl W. Schulz, William L. Barth, Amitava Majumdar, and Dhabhaleswar K. Panda. Quantifying performance benefits of overlap using mpi-2 in a seismic modeling application. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 17–25, New York, NY, USA, 2010. ACM.

[32] Robert Preissl, Martin Schulz, Dieter Kranzlmüller, Bronis R. de Supinski, and Daniel J. Quinlan. Transforming mpi source code based on communication patterns. *Future Gener. Comput. Syst.*, 26(1):147–154, January 2010.

[33] Daniel J. Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[34] José Carlos Sancho, Kevin J. Barker, Darren J. Kerbyson, and Kei Davis. Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[35] Sukhyun Song and Jeffrey K. Hollingsworth. Designing and auto-tuning parallel 3-d fft for computation-communication overlap. *SIGPLAN Not.*, 49(8):181–192, February 2014.

[36] Jesper Larsson Träff. Optimal mpi datatype normalization for vector and index-block types. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 33:33–33:38, New York, NY, USA, 2014. ACM.

[37] Jesper Larsson Träff and Antoine Rougier. Mpi collectives and datatypes for hierarchical all-to-all communication. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 27:27–27:32, New York, NY, USA, 2014. ACM.

[38] Yuichi Tsujita, Atsushi Hori, and Yutaka Ishikawa. Locality-aware process mapping for high performance collective mpi-io on fefs with tofu interconnect. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 157:157–157:162, New York, NY, USA, 2014. ACM.

[39] J. Labarta V. Subotic and M. Valero. Overlapping mpi computation and communication by enforcing speculative dataflow. 2008.

[40] Tim S. Woodall, Galen M. Shipman, George Bosilca, Richard L. Graham, and Arthur B. Maccabe. High performance rdma protocols in hpc. In *Proceedings of the 13th European PVM/MPI User's Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, EuroPVM/MPI'06, pages 76–85, Berlin, Heidelberg, 2006. Springer-Verlag.