Petal Tool for Analyzing and Transforming Legacy MPI Applications

Hadia Ahmed[1], Anthony Skjellum[2], Peter Pirkelbauer[1]

[1] University of Alabama at Birmingham
Birmingham, AL 35294
{hadia, pirkelbauer}@uab.edu

[2] Auburn University
Auburn, AL 36830
skjellum@auburn.edu

Abstract. Legacy MPI applications are an important and economically valuable category of parallel software that rely on the MPI-1, MPI-2 (and, more recently, MPI-3) standards to achieve performance and portability. Many of these applications have been developed or ported to MPI over the past two decades, with the implicit (dual) goal of achieving acceptably high performance and scalability, and a high level of portability between diverse parallel architectures. However they were often created implicitly using MPI in ways that exploited how a particular underlying MPI behaved at the time (such as those with polling progress and poor implementation of some operations). Thus, they did not necessarily take advantage of the full potential for describing latent concurrency or for loosening the coupling of the application thread from the message scheduling and transfer.
This paper presents a first transformation tool, Petal, that identifies calls to legacy MPI primitives. Petal is implemented on top of the ROSE source-to-source infrastructure and automates the analysis and transformation of existing codes to utilize non-blocking MPI and persistent MPI primitives. We use control flow and pointer alias analysis to overlap communication and computation. The transformed code is capable of supporting better application bypass, yielding better overlapping of communication, computation, and I/O. We present the design of the tool and its evaluation on available benchmarks.

1 Introduction

The Message Passing Interface (MPI) describes a library that enables the development of portable parallel software for large-scale systems. The first MPI standard [12] focused on providing a basic framework for point-to-point and collective communication. MPI-2 [8] introduced one-sided communication, added support for parallel file access, and dynamic process management, and extended the usefulness of two-group (inter-communicator) operations. MPI offers a small set of core functions that are sufficient for the development of many applications, and also offers functionality that helps experts optimize applications [10]. MPI bindings exist for C++, Fortran, and many other languages, making MPI one of the most prevalent programming models for high-performance computing. MPI is supported on many platforms, which makes applications developed with MPI portable to many large-scale systems. Building high-performance computing systems constitutes a large investment in human resources. As the communication infrastructure advances and the MPI standards and library implementations follow suite, legacy codes becomes a potential liability. Code that does not utilize more recent MPI primitives will not scale well on newer architectures. This effect will become more marked over time.

With Exascale systems on the horizon, the cost of communication is becoming a major concern. Compared to older architectures, communication incurs relatively more overhead. Legacy software written for older architectures often utilizes `MPI_Send` and `MPI_Recv` for the communication of point-to-point messages. These two primitives block until the data exchange completes (or at least till the send buffer can be reused by the calling thread). While this makes it easy for programmers to reason about communication, such methods fail to utilize computing resources efficiently. On next generation hardware, the implied cost of sending data using a polling and/or blocking mode of communication significantly rises and it is expected that software relying on blocking communication will have too much overhead. In order to take advantage of the architectural changes in Exascale, existing code needs to be transformed to use better primitives, some of which are only available in MPI-3 or higher. Non-blocking primitives allow overlap of communication with local computation[3]. A paired, non-blocking communication uses two MPI routines, one to start (`MPI_Isend`, `MPI_Irecv`) and one to complete (`MPI_wait`). After a communication has been initiated, code can compute, and only waits at the `MPI_wait` to synchronize with the communication operation. In addition to the benefits of non-blocking, applications that exhibit fixed point-to-point communication patterns can further utilize persistent operations introduced in MPI-1 and being extended in MPI-3.x. Persistent MPI primitives reduce communication overhead in applications that exhibit fixed patterns. Persistent MPI operations minimize the overhead incurred from redundant message setup.

Rewriting legacy MPI programs by hand is both tedious and error prone. To relieve programmers of the task of manually rewriting applications, the authors have developed tool support to replace uses of MPI primitives that are known to perform slowly on modern hardware (or may have better alternatives, especially on next-generation architectures) with better alternatives in the MPI standard. We have implemented a source code rejuvenation tool [16] called Petal using the ROSE source-to-source infrastructure [17][3]. We chose ROSE for its support of many languages relevant for high-performance computing. Petal an-

[3] provided the underlying MPI does not poll excessively to make progress or for message completion, the messages are long enough, and there is sufficient memory bandwidth for both communication and computation.

alyzes existing source code and finds calls to `MPI_Send` and `MPI_Recv`. It replaces these primitives with their non-blocking counterparts and uses data-dependency and control-flow information to find code locations where corresponding calls to `MPI_Wait` need to be inserted. If Petal can determine that the communication partners, message buffer, and message length do not change, persistent communication primitives will be used in lieu of non-persistent functions.

Overall, this paper offers the following contributions:

– program analysis and transformation to replace blocking MPI calls with non-blocking calls;
– program analysis and transformation to introduce persistent MPI calls; and,
– analysis of persistent MPI implementations.

The remainder of this paper is organized as follows. §2 presents more detailed information on MPI and ROSE. §3 describes our implementations and §4 discusses our evaluation and findings. §5 gives an overview of related work on MPI transformations, and §6 offers conclusions and an outlook on possible future work.

2 Background

This section provides background information on MPI and the ROSE compiler infrastructure.

2.1 MPI primitives

MPI offers several modes of operation for point-to-point communication. Many programs employ `MPI_Send` and `MPI_Recv`, two blocking MPI primitives. `MPI_Send` takes the following arguments: base pointer to message data, the number of elements to send, a type descriptor, the destination, and a communicator. The base pointer to data typically points to a send buffer, but it could also point to data described by a type descriptor. Blocking means that the MPI primitive waits until the message buffer containing the data being sent/received is safe to be used again by the calling process. Only then is control returned to the caller. On send, actual implementations of `MPI_Send` may either block until all data has been transmitted or copy the data to an intermediate internal buffer. The use of blocking primitives may be prone to deadlocks, if programmers do not carefully consider send and receive order [13]

`MPI_Isend` and `MPI_Irecv` are non-blocking versions for point-to-point message communication. Compared to `MPI_Send`'s arguments, `MPI_Isend` adds an additional argument for a request handle. The handle is used in calls to `MPI_Wait` to identify which send to wait for. Non-blocking calls return immediately after initiating the communication and the user thread can execute more operations, eventually followed by a completion operation (a wait or test) on the request. The communication is considered complete after a successful call to `MPI_Wait` (or `MPI_Test`, etc.). Non-blocking is used to help promote overlap communication and computation, resulting in communicating cost hiding and yielding overall better performance on systems that support it. To avoid tampering with the data, programmers must ensure that the message data is not modified before the communication is completed.

Another mode is offered by persistent communication primitives. If a program exhibits regular communication patterns (static arguments), where the same communication partners exchange fixed size messages, utilization of persistent MPI enables exploitation of faster communication paths. Provided MPI implementations efficiently implement these operations, persistence supports reduced overhead by eliminating cost associated with repeated operations and streamlined processing of derived datatypes. Persistence also can reduce jitter and allow for preplanned choice of algorithms, such as for MPI collectives. Since persistence in MPI offers many benefits (potential and long observed), it is likely that future MPI standards will enhance support for persistent primitives, for example by supporting variable length messages between the same communication partners.

Note that all three modes can be used interchangeably. It is possible that one side uses persistent MPI, while the other side does not. That is why the functions are sometimes referred to as providing half-channels.

Fig. 1 shows the use of blocking, non-blocking, and persistent operations for a simple 1D heat transfer code. The basic design of the heat-transfer code is depicted in Fig. 1d. The code uses two arrays, containing cells with temperature information. The initial temperatures are located in the even array. In odd numbered timesteps the odd array is computed from the even array and in even numbered timesteps vice versa. Red cells are computed by neighbors and dark blue cells are needed by neighbors for the next iteration. Fig. 1a shows a blocking implementation. The order of sends and receives is important to avoid deadlock. Even-numbered MPI processes send first, odd numbered processes receive first. `D` stands for `MPI_DOUBLE`, and `n` is the rank of this node. For simplicity, the codes assume that each process has two neighbors and ignores send and receive status. Fig. 1b demonstrates the overlap of communication and computation in non-blocking mode. The key idea is that the inner (light blue) cells can be computed before the data from neighbors are received. The code starts two receive operations to receive both neighbor's data from the last iteration. Then it starts two send operations to communicate its values from the previous iteration to its neighbors. While the communication is ongoing, the inner cells are computed. Before cells depending on neighbors' data can be computed, the code waits until the data have been received (Line 10). After computing the outer cells, the wait in Line 13 blocks until the data have been sent. This is necessary in order not to overwrite the data in the next iteration. Fig. 1c shows the persistent version of the code. Since the communication patterns, buffer, and buffer size do not change, we can set up the communication for sends and receives at the beginning of the program, and reuse this pattern in every iteration.

```
double b[4]; // send/receive buffer

for (int i = 0; i<MAX; ++i) {
  data_to_buf(prev, b+2);

  if (n%2 == 0) {
    MPI_Recv(b+0, 1, D, n−1, 0, com);
    MPI_Recv(b+1, 1, D, n+1, 0, com);
  }
  MPI_Send(b+2, 1, D, n−1, 0, com);
  MPI_Send(b+3, 1, D, n+1, 0, com);
  if (n%2 == 1) {
    MPI_Recv(b+0, 1, D, n−1, 0, com);
    MPI_Recv(b+1, 1, D, n+1, 0, com);
  }

  buf_to_data(b, prev);
  compute_all(prev, curr);
  swap(curr, prev);
}
```

(a) Blocking operations

```
MPI_request r[4]; // request handler
double b[4]; // send/receive buffer

for (int i = 0; i<MAX; ++i) {
  data_to_buf(prev, b+2);
  MPI_Irecv(b+0, 1, D, n−1, 0, com, r+0);
  MPI_Irecv(b+1, 1, D, n+1, 0, com, r+1);
  MPI_Isend(b+2, 1, D, n−1, 0, com, r+2);
  MPI_Isend(b+3, 1, D, n+1, 0, com, r+3);
  compute_inner(prev, curr);
  MPI_Wait(2, req+0, IGNORE);
  buf_to_data(b, prev);
  compute_outer(prev, curr);
  MPI_Wait(2, req+2, IGNORE);
  swap(curr, prev);
}
```

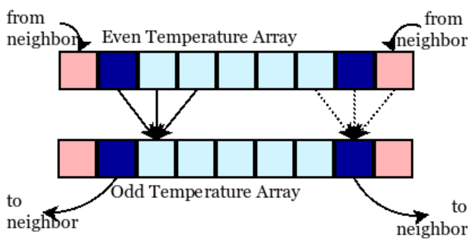
(b) Non-blocking operations

```
MPI_request r[4]; // request handler
double b[4]; // send/receive buffer

MPI_Recv_init(b+0, 1, D, n−1, 0, com, r+0);
MPI_Recv_init(b+1, 1, D, n+1, 0, com, r+1);
MPI_Send_init(b+2, 1, D, n−1, 0, com, r+2);
MPI_Send_init(b+3, 1, D, n+1, 0, com, r+3);
for (int i = 0; i<MAX; ++i) {
  data_to_buf(prev, b+2);
  for (int j = 0; j < 4; ++j)
    MPI_Start(r+j);

  compute_inner(prev, curr);
  MPI_Wait(2, r+0, IGNORE);
  buf_to_data(b, prev);
  compute_outer(prev, curr);
  MPI_Wait(2, r+2, IGNORE);
  swap(curr, prev);
}
```

(c) Persistent operations



(d) Design Overview

Fig. 1: 1D heat transfer

2.2 The ROSE compiler infrastructure

The ROSE source-to-source translation infrastructure is under active development currently at the Lawrence Livermore National Laboratory (LLNL). ROSE provides front ends for many languages, including C/C++, Fortran 77/95/2003, Java, and UPC. ROSE also supports several parallel extensions, such as OpenMP and CUDA. ROSE generates an Abstract Syntax Tree (AST) for the source code. The ASTs are uniformly built for all input languages. ROSE offers many specific analyses (e.g., pointer alias analysis) and makes these available through an API. Users can write their own analyses by utilizing frameworks that ROSE provides. These include attribute evaluation traversals, call graph analysis, control flow graphs, class hierarchies, SSA representation, and dataflow analysis. The Fuse framework[4], is an object-oriented dataflow analysis framework that affords users with the ability to create their own inter- and intra-procedural dataflow analyses by implementing standard dataflow components. ROSE has been used for building custom tools for static analysis, program optimization, arbitrary program transformation, domain-specific optimizations, performance analysis, and cyber-security. With the representation of the code as an AST and using the static analysis provided from the ROSE libraries, one can explore the code and determine how to improve it by looking for certain code style, inserting new code, changing and/or removing old code, hence generating modified source code while preserving the semantics of the original code.

3 Implementation

In this section, we describe Petal's implementation of a mechanism to transform applications from using blocking MPI point-to-point routines to using non-blocking versions. We also describe the analysis and transformations to introduce persistent routines.

3.1 Design

Petal transforms code to use non-blocking MPI operations to reveal a better potential overlap of computation and communication and adds persistent operations, whenever possible, to eliminate much of the overhead of repeatedly communicating with a partner node.

Fig. 2 shows an overview of our transformation framework. The tool takes MPI source files, for which ROSE compiles and generates the Abstract Syntax Tree (AST), then function calls are inlined if the function implementation should be available. Once inlined, ROSE's query and builder libraries are used to find and replace blocking with non-blocking calls and to identify where to insert corresponding calls to `MPI_Wait`. If some or all of these non-blocking calls are used repeatedly with the same arguments, they are replaced with persistent communication operations. At the end, Petal generates a new transformed source file as its output, using either non-blocking or persistent communications (which are always non-blocking).

The idea of following this approach is based on trying to maximize the overlap between communication and computation without compromising the semantics of the original application. Inlining eliminates the need to use inter-procedural analysis and simplifies moving `MPI_Wait` downward, crossing its original function boundaries if no unsafe access to the message buffer is found across the function calls. MPI uses pointers to the message buffers that they use in their communication. This fact allowed us to simplify the analysis used by the tool and focus only on using pointer alias analysis. ROSE's pointer alias analysis implements Steensgaard's algorithm, which has linear time complexity [19]. This allows our tool to scale well with large applications.

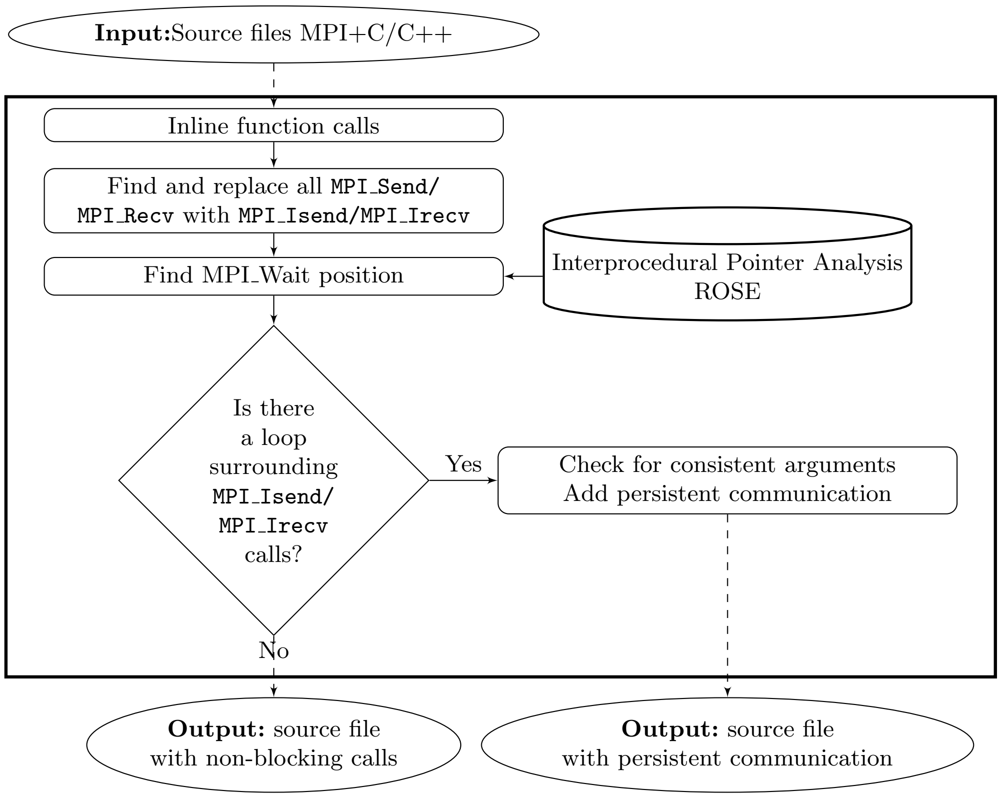

Input: Source files MPI+C/C++
Inline function calls
Find and replace all MPI_Send/ MPI_Recv with MPI_Isend/MPI_Irecv
Interprocedural Pointer Analysis ROSE
Find MPI_Wait position
Is there a loop surrounding MPI_Isend/ MPI_Irecv calls?
Yes
Check for consistent arguments Add persistent communication
No
Output: source file with non-blocking calls
Output: source file with persistent communication

Fig. 2: Transformation Framework

3.2 Blocking to Non-blocking Transformation

Petal allows changing the blocking function call `MPI_Send/MPI_Recv` to the corresponding `MPI_Isend/MPI_Irecv` while ensuring proper access to the message buffers, and once an operation that access the message buffer is encountered, `MPI_Wait` is inserted before it to ensure the safety of the data.

Calling `MPI_Send/MPI_Recv` is in effect the same as calling `MPI_Isend`/`MPI_Irecv` immediately followed by `MPI_Wait`. Our tool moves calls to `MPI_Wait` downward along forward control flow edges as long as the operations are safe with respect to the MPI operation and buffer access. Any write to a message buffer that is used in a send operation, and any access to a message buffer that is used in a receive operation is considered an unsafe access and `MPI_Wait` must be called before that to maintain the correctness of the code.
For each blocking call, to be replaced by the corresponding non-blocking, three variables are created, two of which are handlers for `MPI_Request` and `MPI_Status` plus a flag introduced to ensure the execution of `MPI_Wait` if and only if its corresponding non-blocking call is executed. Each blocking call is replaced with the corresponding `MPI_Isend/MPI_Irecv`. After finding and replacing blocking calls,

control flow analysis is used to find subsequent statements, extract the variables used in these statements and use pointer analysis to test for aliasing between the message buffer used and the variables in hand. For the send operation, we identify potential update operations, such as a variable occurring on the left hand side of an assignment. We use pointer alias analysis to check whether an update could modify some data. For the receive operation, all expressions that read values from a variable are tested. Variable extraction includes subscripts of an array, arguments in non-inlined function calls, variables used in conditions of control statements, initial and increment statements of for loop, and operands of binary and unary operations. Our tool uses ROSE's pointer alias analysis to test whether the extracted variables and the communication buffer could alias. If there could be an alias, the tool inserts the corresponding `MPI_Wait` before the statement using this variable.

Because of inlining, Petal is able to bypass the end of the function and keep searching for potential usage of the message buffer outside the function containing the original MPI calls. If no alias is found in all the statements following the block call, the tool identifies where this statement is located. If it is in `main()`, that means that no alias is found and the `MPI_Wait` is inserted before the `MPI_Finalize`. Because of the complexity of loop-carried data dependencies, currently the tool does not support moving `MPI_Wait` outside the loop body. Hence, if it is in a loop statement (for, while, do-while) `MPI_Wait` is inserted as the last statement in the loop. Otherwise the statement following the block that has the blocking call is examined for alias analysis. To ensure that the `MPI_Wait` in its new position gets executed only if its corresponding non-blocking call is executed, a flag is set to true with each non-blocking call and then based on its value, the corresponding `MPI_Wait` is executed.

Fig. 3 shows an example of a snippet of code before and after transformation. Fig. 3a shows the original blocking code and Fig. 3b shows how the code looks after the transformation. Lines 3-5 shows the declaration of the `MPI_Request`. `MPI_Status` and the flag variables.Line 10 sets the flag to 1 where Line 21 tests for the flag's value before executing the `MPI_Wait` on Line 22. Since this is a send call, the printf function call is a safe read access and the wait call is inserted after it.

3.3 Non-persistent to Persistent Transformation

If a program exhibits regular communication patterns, where the same communication partners exchange fixed size messages, utilization of persistent MPI enables exploitation of faster communication paths[4]. In Shao et. al. [18] work to identify communication patterns for MPI programs, they discovered that many programs that are considered dynamic can use persistent communication. This means that changing these programs to use persistence will result in better performance. The difficulty of persistent communications is that possible uses in real world codes are hard to determine statically. To overcome this limitation, we use

[4] at least on high quality implementations of MPI.

```
int *buffer;
int x;
... //code for main,initialization,...
for(int i=0;i<1000;i++)
{
if (myid == source) {
    *buffer = 123;
    MPI_Send(buffer,count,MPI_INT,
        dest,tag,MPI_COMM_WORLD);
    x = 0;
    }
  else {
    *buffer = 456;
    x = 1;
  }
  printf("%d\n",*buffer);
}
```

(a) Before

```
int *buffer;
int x;
MPI_Request reqs[1];
MPI_Status stats[1];
int flags[1];
... //code for main,initialization,...
for(int i=0;i<1000;i++)
{
if (myid == source) {
    flags[0]=1;
    *buffer = 123;
    MPI_Isend(buffer,count,MPI_INT,
        dest,tag,MPI_COMM_WORLD,&reqs[0]);
    x = 0;
    }
  else {
    *buffer = 456;
    x = 1;
  }
  printf("%d\n",*buffer);
  if (flags[0] == 1)
    MPI_Wait(&reqs[0],&stats[0]);
}
```

(b) After

Fig. 3: Non-Blocking Transformation Example

dynamic analysis. Petal transforms code to persistent mode and inserts guards that test that the arguments did not change. Persistent communication is a four-step process. First, a persistent request is created. Then, data transmission is initiated. After that, wait routines must be called to ensure proper completion. Lastly, the persistent request handlers must be explicitly deallocated.

Changing to persistent mode is best suited for non-blocking calls in a loop. Petal does such transformations from non-blocking non-persistent to persistent automatically. A structure is created to hold initial values for non-blocking call arguments as its members. Using ROSE queries, the tool identifies `MPI_Isend/MPI_Irecv` and checks to see which one is enclosed in a loop. If no call is in a loop, no transformations are performed. If one or more are found inside a loop, the tool initiates a persistent request with the same arguments as the corresponding non-blocking call and places this initiation process before the loop (`MPI_Send/Recv_Init`). In addition, it stores the values of the `MPI_Isend/MPI_Irecv` arguments in a struct variable for comparing the values across iterations. Then inside the loop, it inserts an if statement to check if the current values are the same as the persistent request argument values, if the outcome is yes, it uses this persistent request using `MPI_Start(&request)`, otherwise it uses the normal `MPI_Isend/MPI_Irecv` call. After the loop, all the created persistent requests are freed.

Following the output from Fig. 3b, Fig. 4 shows the result of applying the persistence transformation. On the left side, line 6 shows the persistent request handler and line 7-16 shows the struct definition and its instance declaration. Line 20 initiates the persistent communication passing it all the non-blocking

```
1   int *buffer;
    int x;
3   MPI_Request reqs[1];
    MPI_Status stats[1];
5   int flags[1];
    MPI_Request preqs[1];
7   struct buf_data
    {
9     void *buf;
      int count;
11    MPI_Datatype datatype;
      int dest;
13    int tag;
      MPI_Comm comm;
15  }
    struct buf_data temp_data[1];
17  ... //code for main,initialization,...
19  MPI_Send_init(buffer,count,MPI_INT,
21  dest,tag,MPI_COMM_WORLD,&preqs[0]);
23        temp_data[0] . buf = buffer;
          temp_data[0] . count = count;
25        temp_data[0] . datatype = MPI_INT;
          temp_data[0] . dest = dest;
27        temp_data[0] . tag = tag;
          temp_data[0] . comm =
29                  MPI_COMM_WORLD;
```

(a) Persistent

```
1   for(int i=0;i<1000;i++)
    {
3   if (myid == source) {
     flags[0]=1;
5    *buffer = 123;
     if (temp_data[0] . buf == buffer
7          && temp_data[0] . count == count
           && temp_data[0] . datatype == MPI_INT
9          && temp_data[0] . dest == dest
           && temp_data[0] . tag == tag
11         && temp_data[0] . comm == MPI_COMM_WORLD
     {
13    MPI_Start(&preqs[0]);
      }
15    else {
       MPI_Isend(buffer,count,MPI_INT,
17     dest,tag,MPI_COMM_WORLD,&reqs[0]);
       }
19      x = 0;
      }
21    else {
        *buffer = 456;
23      x = 1;
      }
25    printf("%d\n",*buffer);
      if (flags[0] == 1)
27      MPI_Wait(&reqs[0],&stats[0]);
    }
29  MPI_Request_free(&preqs[0]);
```

(b) contd

Fig. 4: Persistent Transformation Example

arguments and lines 23-29 represents the copying of the arguments values to the struct instance. On the right side, line 6-11 represents the test against the current values with the values stored in the persistent request. If they are the same `MPI_Start` on line 13 is executed, otherwise the original `MPI_Isend` is executed on line 16-17. Line 29 shows the deallocation of the persistent request.

3.4 Discussion

Even though the tool can detect any unsafe access to the message buffers correctly, the applied analysis has limitations in two cases. First, it treats any access to a part of the array as an access to the whole array. For example if MPI sends the first 10 elements of a 100-element array, an assignment to the 20th element will be considered unsafe even though it is in a different place and can be safely used. The second case is that Steensgaard algorithm treats a struct member access as an access to the whole struct [19]. These two cases might lead to placing the `MPI_Wait` in overly conservative positions in some applications. We plan to improve our tool to handles these cases better, since identifying these cases could result into achieving better communication-computation overlap.

Currently, Petal cannot combine multiple consecutive calls to `MPI_Wait`, if found together, into a single `MPI_Waitall` call. This is because different calls to

`MPI_Isend`/`MPI_Irecv` may originate in alternative blocks. For example, two calls are part of the then and else branch of an if statement. We hope to find a better solution instead of using flags and if-statement, to ensure the semantics of the code and being able to take advantage of using `MPI_Waitall`.

4 Evaluation

In this section, we present the preliminary evaluation of using Petal and the effect of its transformations on overall application performance. The experiments were performed on the TACC Stampede system. Stampede is a 10 Petaflop (PF) Dell Linux Cluster with 6400+ Dell PowerEdge server nodes each with 32GB memory, 2 Intel Xeon E5 (8-core Sandy Bridge) processors and an additional Intel Xeon Phi Coprocessor (61-core Knights Corner) (MIC Architecture) [20]. We used the mvapich2 MPI library. Petal was tested with the 1D heat decomposition described earlier, 2D heat [7] and DT from the NAS NPB 3.3 benchmark [1].

We tested the performance of the application while varying the number of MPI processes. For 1D heat, we varied the number of MPI processes in each case ranging from 6 to 200 tasks. For 2D heat and DT with classes W and A, the number of MPI processors varied between 16 and 256. Fig. 5 shows the execution time speedup ($S = T_{\text{original}}/T_{\text{transformed}}$) after applying non-blocking transformation, and adding persistent communication. Fig. 5a shows the effect when running applications with only 16 MPI processes, while Fig. 5b shows the effect on applications with 200 and more processes. As shown in the figures, we experienced good improvement with larger number of processes while flat to minor slowdown was observed with fewer numbers of processes. However, in both cases we experienced minor slowdown when adding persistence[5].

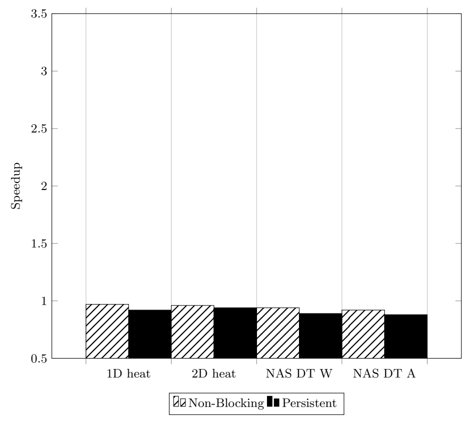
4.1 Discussion of Results

Petal was able successfully to transform applications from blocking to non-blocking while pushing `MPI_Wait` as far as possible, while also preserving the correctness of the code output. The results shows that with smaller programs and few tasks, the non-blocking improvement is negligible and sometimes hurts the application performance. However, with increasing problem size and number of MPI tasks, non-blocking enhanced the performance by up to 30%.

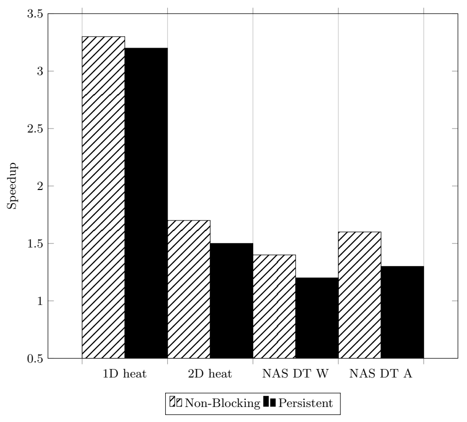
Unfortunately, even though Petal was able to transform code to persistent mode, the results of persistent performance showed a flat improvement and sometimes a slowdown.

To gain more insight into the usage of persistent communications, we applied the persistent transformation on the LULESH code from LLNL [11] on Stampede and on a Debian 7.6 amd64 computer with 1 Xeon E5410 @ 2.33GHz using the Open MPI 1.6.5 library. LULESH already exploits non-blocking operations. Since it has some communications that are fixed for most of the program's

[5] This indicates that mvapich may not optimize the code path for persistent send and/or receive.



(a) (NP=16)



(b) (NP=256)

Fig. 5: Execution Time Speedup

execution time, persistent communication should be beneficial. However, upon transforming to persistent no gain was seen and with increasing number of tasks we saw a minor slowdown. Since Open MPI is open source, we investigated how it implements its non-blocking and persistent communications. We found that they optimize the code by creating persistent requests and using them whenever possible. Hence, changing the applications' code to persistent will not give a speedup as Open MPI already uses similar optimization techniques internally. The slowdown might be because of the overhead of checking the arguments on each iteration.

According to the MPI Forum [2], persistent requests are considered to be half-channels, which makes the connection faster by reducing the overhead of communication processing within each of the sender and receiver. Our results suggest that the performance improvement is dependent not only on the standard definition of how code should work but it also depends on the actual MPI implementation and architecture. While the tested systems did not show any performance improvements, the transformation may be beneficial on other systems.

5 Related Work

The idea of overlapping communication and computation code is of interest to many researchers because of the promising results in better performance it can give when applied efficiently. In this section, we describe previous research work done to produce overlapped communication and computation in MPI.

Several methods were studied and implemented to handle the communication computation overlap approach. Das et al. [6] represents the closest work to our tool in which they developed an algorithm for pushing wait downward in a segment of code. However, they use Static Single Assignment (SSA) use-def analysis

to determine the statements that access the message buffer. Even though they describe a method for moving a `MPI_Wait` out of its current scope interval possibility of going to the parent, they did not implement their method and currently their compiler tool only detects MPI calls and finds `MPI_Wait`'s final position; however, insertion is done by hand. Haque et. al. [9] developed a similar tool for transforming blocking to non-blocking; however, it does not use any compiler analysis techniques and relies heavily on the programmer annotation to identify where to move the corresponding non-blocking call and its corresponding wait. Another work is presented by Nguyen et. al. in [14] in which they developed Bamboo, a transformation system that transforms MPI C code into a data-driven application that overlaps computation and communication. It was implemented with the ROSE compiler framework and runtime support using the Tarragon runtime library. Their approach is to determine task precedence. It relies on programmer annotations to mark parallel loops and data packing/unpacking plus calls to communication routines. Other approaches were developed using different techniques to achieve the same goal of maximizing communication and computation overlap. Danalis et al. developed the ASPhALT tool [5] within Open64. Their idea is based on automatically detecting where data is available and applying the pre-pushing transformation to send data as soon as possible. They focused on specific a type of applications that does its communication in two parts where at first, it computes the data in a loop with minimum dependencies across iterations, and then uses communication call(s) after the loop to exchange the data generated by the loop. Pellegrini et al. [15] offer a different approach in which they use the polyhedral model to determine exact dependencies and automatically detect potential overlap on a finer grain. To simplify the analysis, they normalize the code by changing non-blocking to blocking. Their work is limited by polyhedral model requirements of using only affine expressions.
Even though MPI included persistent communication since MPI-1 and these calls emphasize the benefits of using persistent, to our knowledge, no available work offers a tool that automatically transforms non-persistent to persistent communication, when such patterns can be identified.

6 Conclusions and Future Work

In this paper, we described our development of Petal, a tool that supports transforming a blocking MPI code to non-blocking version and introduces persistent communication if possible. We have described the approach used in order to push `MPI_Wait` as far as possible from the corresponding communication call in order to improve the potential for overlap of communication and computation code and also to use persistent communication whenever two points communicate the same type and amount of data over multiple iterations. Petal is based on the ROSE framework and uses ROSE's alias analysis to apply transformation required and to preserve correctness of the code. Preliminary results showed that we can improve performance by using non-blocking. In some cases we found that

persistent communication does not improve performance even with code that is proved to have fixed communication for most of the execution time. It does not only depend on having fixed arguments but the MPI library used has an effect too. Further detailed analyses of persistent performance on different architectures with different libraries will be explored.
In addition to analyzing data dependency within loop iterations and moving `MPI_Wait` outside the loop body, if no dependency found, techniques to eliminate loop-carried dependencies on send and receive buffers and perhaps unrolling the loops will also be explored. This will provide another opportunity to move `MPI_Wait`(s) outside loops boundaries. Another future step is to work on cases where we have 3-D data models and to explore how they can be safely overlapped in communication.
We are also extending the Petal tool to do other automatic translation and/or refactoring that will allow a smooth transition for legacy MPI systems to Exascale systems, such as the use of one-sided communications and changing further to use non-blocking and persistent collective operations (being proposed at present in MPI-3.x).

Acknowledgements This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1053575.

References

1. The NAS parallel benchmarks. https://www.nas.nasa.gov/publications/npb.html
2. Persistent MPI communication. http://www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node51.html, accessed on June, 28th 2015
3. The ROSE source-to-source compiler. http://rosecompiler.org
4. Aananthakrishnan, S., Bronevetsky, G., Gopalakrishnan, G.: Hybrid approach for data-flow analysis of mpi programs. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing. pp. 455–456. ICS '13, ACM, New York, NY, USA (2013)
5. Danalis, A., Pollock, L., Swany, M.: Automatic MPI application transformation with asphalt. In: Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. pp. 1–8 (March 2007)
6. Das, D., Gupta, M., Ravindran, R., Shivani, W., Sivakeshava, P., Uppal, R.: Compiler-controlled extraction of computation-communication overlap in MPI applications. In: Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on. pp. 1–8 (April 2008)
7. Frexus: mpi-2d-plate (2013), http://project.github.com, accessed on September 1, 2015
8. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2: Advanced Features of the Message-Passing Interface. MIT Press, Cambridge, MA, USA (1999)
9. Haque, M., Yi, Q., Dinan, J., Balaji, P.: Enhancing performance portability of MPI applications through annotation-based transformations. In: Parallel Processing (ICPP), 2013 42nd International Conference on. pp. 631–640 (Oct 2013)

10. Hoefler, T.: New and old features in MPI-3.0: The past, the standard, and the future (Apr 2012)
11. Karlin, I., Keasler, J., Neely, R.: Lulesh 2.0 updates and changes. Tech. Rep. LLNL-TR-641973 (August 2013)
12. Message Passing Interface Forum: MPI: A message-passing interface standard. Tech. rep., Knoxville, TN, USA (1994)
13. Message Passing Interface Forum: MPI: A message-passing interface standard version 3.1 (June 2015)
14. Nguyen, T., Cicotti, P., Bylaska, E., Quinlan, D., Baden, S.B.: Bamboo: Translating MPI applications to a latency-tolerant, data-driven form. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 39:1–39:11. SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA (2012)
15. Pellegrini, S., Hoefler, T., Fahringer, T.: Exact dependence analysis for increased communication overlap. In: Träff, J.L., Benkner, S., Dongarra, J.J. (eds.) Recent Advances in the Message Passing Interface, Lecture Notes in Computer Science, vol. 7490, pp. 89–99. Springer Berlin Heidelberg (2012)
16. Pirkelbauer, P., Dechev, D., Stroustrup, B.: Source code rejuvenation is not refactoring. In: 36th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM). LNCS 5901, Springer (2010)
17. Schordan, M., Quinlan, D.: A source-to-source architecture for user-defined optimizations. In: JMLC'03: Joint Modular Languages Conference. vol. 2789 of LNCS, pp. 214–223. Springer-Verlag (August 2003)
18. Shao, S., Jones, A., Melhem, R.: A compiler-based communication analysis approach for multiprocessor systems. In: Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International. pp. 10 pp.– (April 2006)
19. Steensgaard, B.: Points-to analysis in almost linear time. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 32–41. POPL '96, ACM, New York, NY, USA (1996)
20. Towns, J., Cockerill, T., Dahan, M., Foster, I., Gaither, K., Grimshaw, A., Hazlewood, V., Lathrop, S., Lifka, D., Peterson, G., Roskies, R., Scott, J., Wilkins-Diehr, N.: XSEDE: Accelerating scientific discovery. Computing in Science Engineering 16(5), 62–74 (Sept 2014)