ETI Technical Report 02

Legacy MPI Codes and its interoperability with fine grain task-parallel runtime systems for Exascale

Sergio Pino

Guang R. Gao

Abstract
Exascale software will be unable to rely on minimally invasive system interfaces to provide an execution environment. Instead, a task-parallel software runtime layer is necessary to mediate between an application and the underlying hardware and software. Industry and academia have years of effort developing MPI codes. For this reason, a progressive transition to the new exascale execution models will require the interoperability with legacy MPI codes. Ideally this interoperability should not degrade the current performance of legacy codes, but it may hinder optimal performance and programmer intervention may be required to remove bottlenecks. In this work, we focus on the codelet-based execution model called SWARM, and explore two methods to provide this interoperability named MPI+SWARM and Codelet MPI. First, MPI+SWARM takes an MPI program and add SWARM calls. Second, Codelet MPI creates an MPI compatibility layer in SWARM which is used by applications. We show the feasibility of these approaches by presenting some simple applications. These examples are attached as a .zip file.

Introduction
The DynAX project needs to interoperate with legacy MPI codes. Because the codes are being modified and recompiled to fit into a new exascale paradigm, we assume that the codes can be recompiled through the XStack software. We also note that interoperability with MPI should not degrade the current performance of legacy codes, but it may hinder optimal performance and programmer intervention may be required to remove bottlenecks.
Legacy code can be parallelized between MPI calls rather straightforwardly. The main MPI thread will be suspended while the parallel code is executed, then the main thread is resumed in a manner similar to how OpenMP and MPI interoperate today. However, this method is limited because only the main MPI thread may make MPI calls.

In this work, we focus on the codelet-based execution model called SWARM, and explore two methods to provide this interoperability named MPI+SWARM and Codelet MPI. First, MPI+SWARM takes an MPI program and add SWARM calls. Second, Codelet MPI creates an MPI compatibility layer in SWARM which is used by applications. We show the feasibility of these approaches by presenting some simple applications.

MPI+SWARM
The first approach for the MPI interoperability is called MPI+SWARM. Here, a developer takes a base MPI program and add SWARM calls in a similar way as the hybrid model MPI+OpenMP. In this approach, SWARM doesn't perform MPI calls to communication routines (point to point communication routines, such as MPI_Send) or Collective Communication Routines (such as synchronization, data movement, or collective computation). The general code structure for this approach is presented below. This can be considered as the first step in order to test the interoperability between the two runtime systems, and doesn't target real applications to use it.

```
#include <eti/swarm_convenience.h>
#include <mpi.h>
// Declare N codelets
CODELET_DECL(c0);
...
CODELET_DECL(cN-1);
int main(int argc, char *argv[]) {
        …
        // Initialize MPI.
        MPI_Init(...); // parallel code begins
        // some MPI calls
        …
        // using SWARM to exploit parallelism in each MPI process
        swarm_posix_enterRuntime(NULL, &CODELET(...), …, ...);
        …
        // more MPI calls
        ...
        // Terminate MPI environment
        MPI_Finalize();
        ...
}
// Codelet implementations
...
```

Fig 1: General structure of a MPI+SWARM application

It is important to notice that we assume a basic interaction as described below:

1. Perform MPI calls to communication routines (point to point communication routines, such as MPI_Send) or Collective Communication routines (such as synchronization, data movement, or collective computation).
2. Enter the swarm runtime system passing the necessary data to the first codelet.
3. Performing the intra-node parallel work with SWARM.
4. Exiting the SWARM runtime.
5. Perform MPI calls to communication routines or Collective Communication routines.
6. Go to 1 is needed, if not exit MPI environment.

Codelet MPI

The second approach for the MPI interoperability is called Codelet MPI. Codelet MPI creates an MPI compatibility layer in SWARM which is used by applications. We addressed this in two ways. First, at the user level code, we created codelets that perform blocking MPI send or recv, each codelet schedules its continuation once the blocking call returns control to the codelet. Second, It is basically a library that provides two general purpose codelets, one to perform

non-blocking MPI_Send and the other to perform non-blocking MPI_Recv, each codelet schedules its continuation when the test for completion of the non-blocking MPI call is true. For this first version, we assume that the application has data dependencies between codelets, so a codelet that performs an MPI_recv operation will need to make sure the data has been received before scheduling its continuation codelet, an example of this behavior is presented in fig 4 and fig 5.

Creating a Codelet MPI that uses MPI blocking calls

By creating codelets that wrapped MPI blocking calls we created some simple applications that demonstrate the possibility of performing MPI calls inside codelets, and it motivated us to pursue the approach described in the next section. This straightforward approach uses a similar basic interaction or code structure as the one defined in the section MPI+SWARM. However, in this approach the step 3 assumes that SWARM codelets not just perform the intra-node parallel but also can perform MPI calls.

An example of a codelet performing a blocking MPI_Send is shown below, it is part of a modified version that uses SWARM and MPI of the mpi_ping.c example presented in [1]. The drawback of this implementation comes from the restricted functionality that a codelet must follow. In SWARM, codelets must not block, because it ties up the runtime thread indefinitely and could stalls out program execution [2]. Thus, unless you can afford that cost, it is better to find a way to define a codelet that interacts with MPI in a non-blocking approach, as presented in the next section.

```
CODELET_IMPL_BEGIN_NOCANCEL(rank0_Send)
   info* data = (info*) (swarm_natP_t) THIS;

   int dest = 1, source = 1, rc;
   rc = MPI_Send(&data->outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);

   // continuation
   swarm_schedule(&CODELET(rank0_Recv), THIS, NULL, NULL, NULL);

CODELET_IMPL_END;
```

Fig 2: Excerpt of code for a codelet using MPI blocking calls.

Creating a Codelet MPI that uses MPI non-blocking calls

We implemented this functionality in a library called dynax_mpix.h. This library provides two general purpose codelets, one for perform asynchronous MPI_Isend and the other to perform asynchronous MPI_Irecv. We assume that the application has data dependencies between codelets, so a codelet that performs an MPI non-blocking call will need to make sure the data has been received before scheduling its continuation codelet. These codelets uses the underlying MPI non-blocking calls and check (without tie up a runtime thread indefinitely)

whether or not the operation was successful. If it was successful, then the codelet calls the continuation codelet defined by the user. If the non-blocking operation hasn't complete then the codelet yields its control in order to give room for another codelet to execute in the current runtime thread.

The API exposes three components and it is presented in figure 3. First, there is a typedef struct called mpix_str, which is used to pass the mpi required information as the THIS parameter in the dynax_mpix codelets.

Second, the codelet to perform non-blocking MPI send is called mpix_send. It takes as its THIS argument an instance of mpix_str. Third, the codelet to perform non-blocking MPI recv is called mpix_recv. As mpix_send, it takes as its THIS argument an instance of mpix_str.
As presented in the code excerpt, the codelet perform an MPI_Isend/MPI_Irecv call and test continuously if the non-blocking operation has finished. If it has not finished then execute another codelet by calling swarm_yield(). With this simple approach SWARM is able to perform MPI calls without tie up a SWARM runtime thread indefinitely.

```
//  Information: buffer, count, type, dest, tag, comm used to call the underlying MPI call
typedef struct {
   int rank;
   void* buf;
   int count;
   MPI_Datatype type;
   int dest_src;
   int tag;
   MPI_Comm comm;
} mpix_str;

// Codelet to manage non-blocking mpi sends
CODELET_DECL(mpix_send);

// Codelet to manage non-blocking mpi recvs
CODELET_DECL(mpix_recv);

// Uses MPI_Test to tests for the completion of a send or receive
bool mpix_mpiTest(MPI_Request* req);
...
CODELET_IMPL_BEGIN_NOCANCEL(mpix_send)

   mpix_str data = *(mpix_str*) (swarm_natP_t) THIS;
   MPI_Request req;

   // non-blocking sending the data
```

```
  MPI_Isend(data.buf, data.count, data.type, data.dest_src, data.tag, data.comm, &req);

  // while "no finishing with non-blocking send" then execute another codelet
  while(!mpix_mpiTest(&req))
          swarm_yield();

  // schedules the continuation
  swarm_schedule(NEXT, NEXT_THIS, INPUT, NULL, NULL);

CODELET_IMPL_END;

CODELET_IMPL_BEGIN_NOCANCEL(mpix_recv)

  mpix_str data = *(mpix_str*) (swarm_natP_t) THIS;
  MPI_Request req;
  // non-blocking receiving the data
  MPI_Irecv(data.buf, data.count, data.type, data.dest_src, data.tag, data.comm, &req);
  // while "no finishing with non-blocking recv" then execute another codelet
  while(!mpix_mpiTest(&req))
          swarm_yield();
  // schedules the continuation
  swarm_schedule(NEXT, NEXT_THIS, INPUT, NULL, NULL);

CODELET_IMPL_END;
...
```

Fig 3: Excerpt from the dynax_mpix.h library. This library encapsulates and offers codelets for MPI and SWARM interoperability.

Results

In figure 4 and figure 5, we show the tracing for an synthetic application that the dynax_mpix library. This application schedules send and recv operations using the codelets defined before and also schedules a dummy codelet that consumes cpu without real work done. Figure 4, shows that the the 8 workers mainly execute the dummy codelet (blue label).

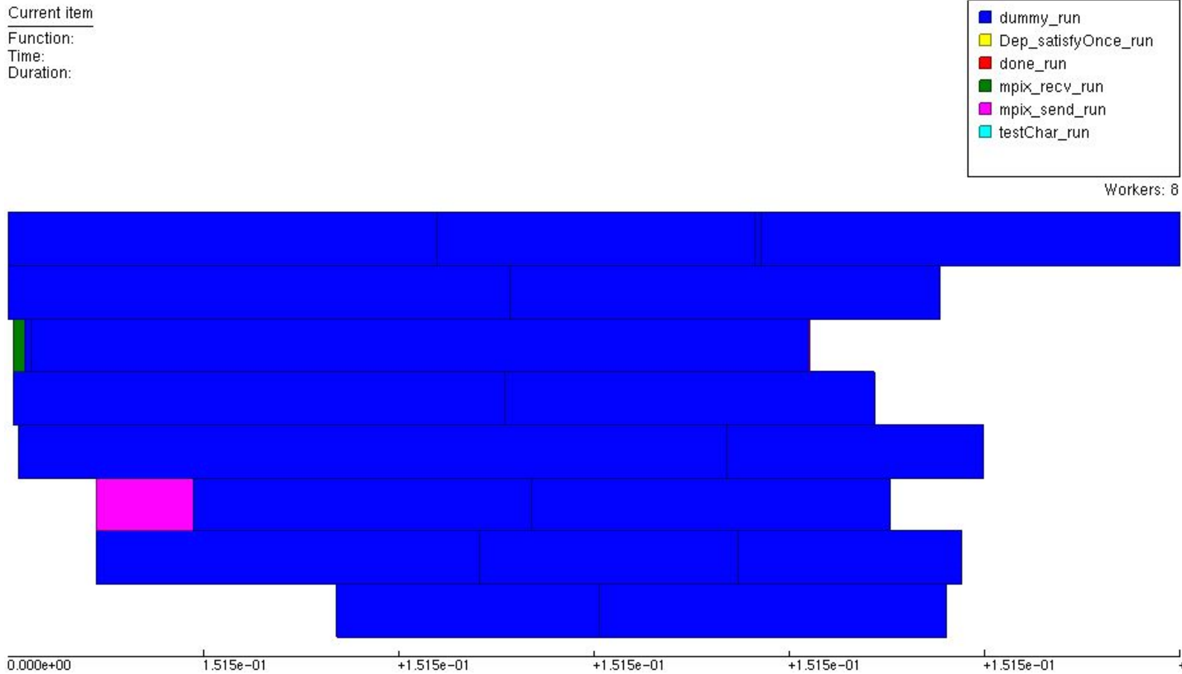

Fig 4: Tracing of the program with all the codelets selected for visualization.

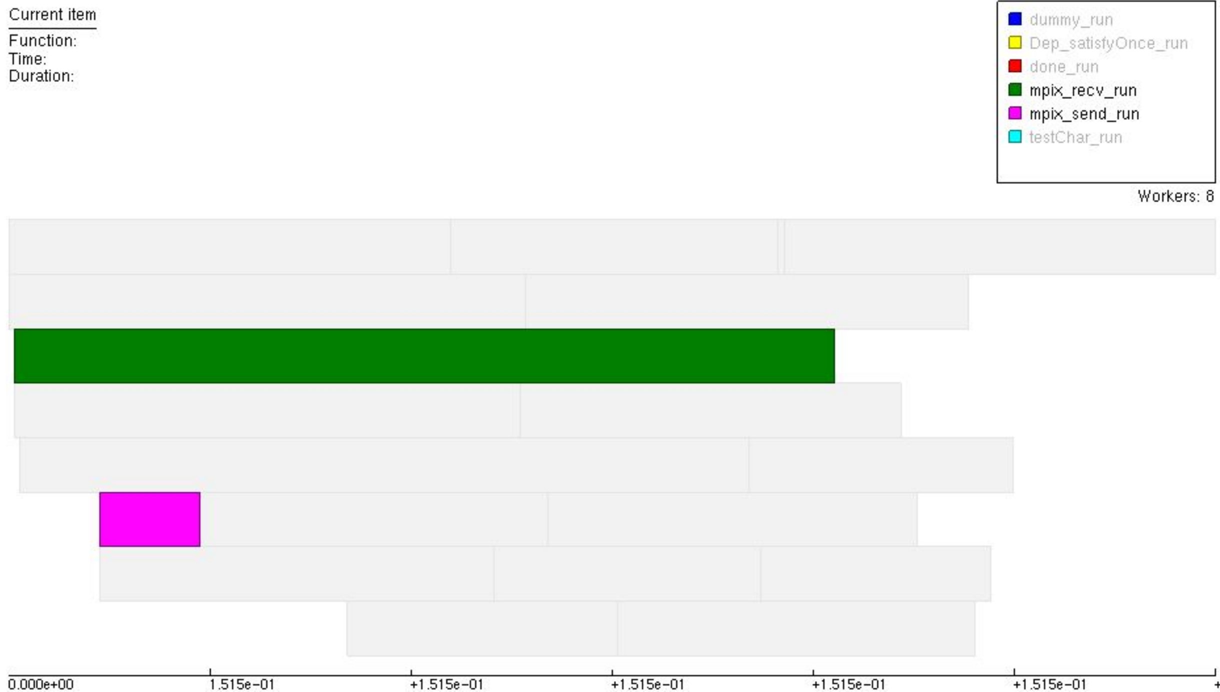

Fig 5: Tracing of the program with just mpix_recv and mpix_send codelets selected for visualization.

In figure 5, we can see the same tracing of the program with just mpix_recv and mpix_send codelets selected for visualization. In here, we can see that mpix_send ran in background the check for completion and allows the runtime thread to schedule other codelets.

Conclusions and Future work

In conclusion, we have demonstrated that SWARM and MPI runtimes can interoperate in a simple application. The examples presented in this work and attached as a .zip file, are simple enough to evaluate the feasibility of the approach and serve as basis to the creation of a more complex benchmarks in the following quarter. Our sample applications shown that MPI calls can be used in a decentralized, continuation-based manner, to provide a fine-grained, low-overhead framework for MPI interoperability with SWARM.

For the dynax_mpix library more work needs to be done to handle errors in the underlying MPI send or recv. For instance, if the message is never sent or it is lost, the current implementation will keep checking indefinitely for the non-blocking call to finish.

References

[1] Message Passing Interface (MPI). High Performance Computing Training. Lawrence Livermore National Laboratory. Retrieved May 4, 2015, from https://computing.llnl.gov/tutorials/mpi

[2] Working with codelets. Programmer's Guide to the SWARM API - SWARM documentation. ET International Inc. Retrieved May 18, 2015, from http://www.etinternational.com/downloads/swarm_docs/swarm/programmers-guide/index.htm