FG-MPI: Fine-Grain MPI
User Guide

Humaira Kamal, Alan Wagner
The University of British Columbia
Email: {kamal,wagner}@cs.ubc.ca

May 5, 2014
Version: 1.0

Contents

1 Introduction

Fine-Grain MPI (FG-MPI) [5, 6] extends the execution model of Message Passing Interface (MPI) to allow for interleaved execution of multiple concurrent MPI processes inside an OS-process. FG-MPI is integrated into the MPICH middleware and has a light-weight design based on coroutines that can scale to millions of MPI processes on a node or across nodes on a cluster. FG-MPI provides the ability to take advantage of finer-grain parallelism available on today's multicore systems, while maintaining MPI's rich support for communication inside clusters. Its flexible process mapping allows granularity of MPI programs to be adjusted through the command-line to better fit the cache leading to improved performance. FG-MPI supports function-level concurrency which enables design of novel algorithms and techniques to achieve scalable performance and match the number of processes to the problem rather than the hardware.

2 Installation

2.1 Installing from the binary package

Download the Ubuntu binary package and install it by:

```
sudo dpkg --install fgmpi_1.0.0-1_amd64.deb
```

The above requires root access and will install in `/usr/bin`. Please be aware that it may overwrite an existing version of `mpich` in `/usr/bin`. If you wish to install in a custom directory then see "Installing from source code" (Section 2.2).

To list the package and check its status:

```
dpkg -l | grep fgmpi
```

To un-install the binary package, do the following:

```
sudo dpkg --remove fgmpi
```

and to purge the package completely:

```
sudo dpkg --purge fgmpi
```

2.2 Installing from source code

The installation steps for FG-MPI are the same as those listed in MPICH installation guide. They are summarized below.

- Download the source code tar file from here.
- Extract the tar file.

  ```
  tar xvzf fgmpi_1.0.0.tar.gz
  ```

  Assume that the files are extracted in /home/user/fgmpi-src
- Create your install directory. Assuming that you wish to install in /home/user/fgmpi-install.

  ```
  mkdir fgmpi-install
  ```

- Create your build directory. Assuming that it is /home/user/fgmpi-build

  ```
  mkdir fgmpi-build
  cd fgmpi-build
  ```

- Specify your configure options. A basic configuration is:

  ```
  /home/user/fgmpi-src/configure --prefix=/home/user/fgmpi-install
  ```

  Other configure options that work with the `mpich2-1.0.8p1` release may also be used with FG-MPI.

- Build and install FG-MPI.

  ```
  make
  make install
  ```

- Update your path settings to add /home/user/fgmpi-install and do a quick installation check by:

  ```
  which mpicc
  which mpiexec
  ```

  By default, FG-MPI uses the hydra process manager and supports many, but not all, of the `mpiexec` options applicable to mpich2-1.0.8p1. For help type:

  ```
  mpiexec --help
  ```

3 Mailing List

A discussion mailing list is available for FG-MPI. In order to subscribe to it please send an email to
`majordomo@cs.ubc.ca` containing the single line:

```
subscribe fgmpi-discuss
```

in the body of the email. There should be no signatures or HTML in the email. You will receive an email asking for confirmation that the subscription request was made by you. Please follow the instructions in that email and after that emails can be sent to `fgmpi-discuss@cs.ubc.ca`. Welcome to the FG-MPI discussion group!

4 Writing FG-MPI Programs

The MPI routines used in writing an FG-MPI program are the same used in any standard MPI program, without any special prefixes or suffixes. The only additional information required is at MPI environment initiation time to specify the mapping of the co-located fine-grain MPI processes to the functions they will be executing. Mapping of the MPI processes to functions is done through a call to a function called `FGmpiexec`, as discussed in Section 4.1 below.

4.1 A simple HelloWorld program

We'll start with a simple SPMD program where all the MPI processes are mapped to the same `helloworld` function.

Listing 1 contains a boilerplate that is inserted at the top of the listing and a `helloworld` function. This `helloworld` function looks exactly like a `main` function in a standard MPI program. The boilerplate contains two user-defined functions; `binding_func` and `map_lookup`. The `binding_func` function takes the `MPI_COMM_WORLD` rank as input and maps it to the function pointer that the corresponding MPI process will be executing. The `map_lookup` function takes a string as its third parameter and uses it to select a binding function. The purpose of the `map_lookup` and `binding_func` combination is to allow the ability to determine mapping of processes at runtime in addition to compiled mappers. This allows experimentation with different bindings of the MPI processes to functions without re-writing and re-compiling the program. Since the mapping is localized to each OS-process, it is also possible to specify a different binding function for each OS-process.

`FGmpiexec` initializes the FG-MPI runtime environment and assigns functions to the corresponding MPI processes.

Listing 1: A simple SPMD FG-MPI program.

```
#include <stdio.h>
#include "mpi.h"

/******* FG-MPI Boilerplate begin *********/
#include "fgmpi.h"
int helloworld( int argc, char** argv ); /*forward declaration*/
FG_ProcessPtr_t binding_func(int argc, char** argv, int rank){
    return (&helloworld);
}
FG_MapPtr_t map_lookup(int argc, char** argv, char* str){
    return (&binding_func);
}
int main( int argc, char *argv[] )
{
    FGmpiexec(&argc, &argv, &map_lookup);
    return (0);
}
/******* FG-MPI Boilerplate end *********/

int helloworld( int argc, char** argv )
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("Hello! I am rank %d of %d\n", rank, size);
    MPI_Finalize();
    return(0);
}
```

4.2 Process deployment

`mpiexec` is a common command-line utility for launching MPI processes and the mapping of processes to cores and machines can be flexibly configured through a `hostfile`.

```
  mpiexec -f hostfile -n Y program
```

Support for MPMD (Multiple Program Multiple Data) is through the colon notation along with `mpiexec` to assign different executables to different processes.

```
  mpiexec -f hostfile -n Y prog1 :  -n Z prog2
```

FG-MPI adds another dimension by mapping multiple MPI processes to OS-processes. The user can use `mpiexec` with the `nfg` flag to specify the number of MPI processes per OS-process in addition to the `n` flag specifying the number of OS-processes.
For example, the command:

```
  mpiexec -f hostfile -nfg X -n Y program
```

starts up $X \times Y$ MPI processes with X MPI processes inside each of the Y OS-processes. In the current implementation, the co-located MPI processes are assigned ranks in consecutive blocks. In the above example, the first OS-process will contain MPI processes of ranks $[0 \ldots X-1]$ and the next one will have processes with ranks $[X \ldots 2X-1]$ and so on. It is possible to specify a different number of MPI processes in each OS-process using the colon notation.

```
   mpiexec -nfg T -n U prog1 :  -nfg V -n W prog2
   :  -nfg X -n Y prog3
```

One important feature of this approach is the flexibility of mapping co-located MPI processes among OS-processes, from the one extreme of executing them all inside a single OS-process to the other extreme of having only one MPI process per OS-process. The number of MPI processes that can be launched per OS-process is limited only by the available memory on the system. The `mpiexec` command is backward compatible, so omitting the `nfg` parameter equates one MPI process with one OS-process.

The second level of mapping introduces new opportunities for executing MPMD programs, where each of the co-located MPI processes invoke functions instead of main programs. We treat SPMD as a special case of the MPMD program, where each process invokes the same function. The boilerplate code in Listing 1 gives an example of a simple SPMD FG-MPI program. However, the real flexibility of mapping comes from the ability to assign different functions for each MPI process. Listing 2 gives a simple MPMD example where the odd numbered ranks are mapped to `ProcessA` function and the rest to `ProcessB` function.

The `str` parameter of the `map_lookup` function is not being used in the examples in Listings 1 and 2 and a single `binding_func` is returned. The `str` parameter can be specified on the `mpiexec` command line to allow selection of different binding functions at runtime.

Listing 2: An example of a simple MPMD mapping

```
/******* FG-MPI Boilerplate begin *********/
#include "fgmpi.h"
int ProcessA( int argc, char** argv ); /*forward declaration*/
int ProcessB( int argc, char** argv ); /*forward declaration*/
FG_ProcessPtr_t binding_func(int argc, char** argv, int rank){
    if (rank % 2) return (&ProcessA);
    else return (&ProcessB);
}
FG_MapPtr_t map_lookup(int argc, char** argv, char* str){
    return (&binding_func);
}

int main( int argc, char *argv[] )
{
    FGmpiexec(&argc, &argv, &map_lookup);
    return (0);
}
/******* FG-MPI Boilerplate end *********/
```

All the functions invoked by the MPI processes e.g., `helloworld` in Listing 1 and `ProcessA` and `ProcessB` in Listing 2, are written as regular MPI applications beginning with `MPI_Init` and ending with `MPI_Finalize` routines. The `argc` and `argv` arguments provided to these functions are the same that are passed to the `main` function in a MPI program. In Section 5, we present the complete code of a small application to demonstrate writing a program using FG-MPI.

5 Example of a Program Using FG-MPI

In this section we present the code of a small application to demonstrate writing a program using FG-MPI. This application creates the sieve of Eratosthenes by composing several fine-grain MPI processes to form a pipeline. A pipeline is a commonly used pattern in process-oriented environments for creating process networks within programs and is also used in dataflow applications. This application demonstrates the use of two different mapping functions for composing the pipeline of processes. These mappings can be selected on the command-line to allow experimentation with load-balancing of processes without recompiling the code.

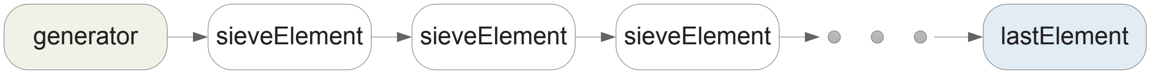

Figure 1: A pipeline of processes, where numbers generated by the `generator` are streamed through the chain to be processed by each element.

As Figure 1 shows we have three types of MPI processes:

1. A `generator` process that generates odd numbers that are passed down the pipeline. The generator keeps the prime number 2 for itself.

2. `sieveElement` process that keeps the first prime number it sees and filters the remaining numbers by either discarding them or passing them to the next process in the chain.

3. A `lastElement` process that terminates the prime number generation at the end of the sieve.

The number of prime numbers generated is equal to the length of the pipeline (i.e., the total number of MPI processes in this application). For example, the following command will generate 40 prime numbers.

```
mpiexec -nfg 10 -n 4 ./primeSieve
```

Listing 3 on page 10 shows the code for the three functions `generator()`, `sieveElement()` and `lastElement()` bound to these processes. As discussed in Section 4.2 (page 5), `map_lookup` function takes a string as its third parameter and uses it to select a binding function. In this example we provide two different binding functions:

1. `sequential_mapper` that binds the `generator` to process rank 0, `lastElement` to the highest rank in `MPI_COMM_WORLD` and the remaining processes are all of type `sieveElement`. The pipeline is composed by specifying the previous and next neighbours of a process in the chain (see `who_are_my_neighbors` function on page 10). In this case we have a sequential assignment where process of $rank - 1$ is the previous neighbour and process $rank + 1$ is the next neighbour. Process rank i will generate the $i + 1$th prime number. Due to the packed assignment of MPI process ranks inside OS-processes, this mapping is not the most efficient since the processes appearing later in the pipeline do not become active until a large number of primes are found. The `random_mapper` binding function addresses this problem.

2. `random_mapper` uses a technique similar to the shuffling of a deck of cards to create a random chain sequence. It uses two user-defined parameters (`seed` and `cuts`) to randomize the next and previous neighbours of a process. This allows a more even distribution among processes and the parameters can be used for experimentation with load-balancing.

We also define two special constants `MAP_INIT_ACTION` and `MAP_FINALIZE_ACTION` that can be used inside the binding functions. `MAP_INIT_ACTION` can be used by the user for any initialization of structures or actions prior to the actual binding of functions to process ranks. An example of this is in the `random_mapper` binding function, where `MAP_INIT_ACTION` is used to allocate a temporary array for storing the random permutation of ranks that is later read by all the newly spawned processes when they start executing and call the `who_are_my_neighbors()` function. `MAP_FINALIZE_ACTION` can be used by the user in the binding function for any action subsequent to the binding operation. Note that at this stage only the binding of functions to MPI process ranks has been completed, but the processes have not executed yet.

The environment variable `FGMAP` is used to select a binding function on the `mpiexec` command line. For example, the random function can be specified as follows.

```
mpiexec -nfg 10 -n 4 -genv FGMAP random ./primeSieve
<seed> <cuts>
```

Whereas, the following specifies the sequential binding function.

```
mpiexec -nfg 10 -n 4 -genv FGMAP seq ./primeSieve
```

It is possible to specify different scheduling policies in a similar manner. For example, FG-MPI implements receive-side blocking where a receiver process is blocked from execution until a matching message is received [2]. The blocking scheduler can be specified as follows (the round-robin (`rr`) is the default scheduler if none is specified).

```
mpiexec -nfg 10 -n 4 -genv SCHEDULER block -genv FGMAP
random ./primeSieve <seed> <cuts>
```

5.1 A quick checklist in writing FG-MPI programs

- Remove any global or static variables in the program that are not read-only.
- Each fine-grain process should begin with `MPI_Init()` and end with `MPI_Finalize()`.
- Do not call `exit()` at the end of the functions mapped to each fine-grain MPI process as that will result in premature termination of the program. Use `return()` instead.

6 Limitations

FG-MPI is a research project and is not a complete implementation of the MPI Standard. The current release is based off MPICH2 release mpich2-1.0.8p1 with some modules updated from mpich2-1.2.1. FG-MPI supports the core MPI routines used for point-to-point communications, collective operations and intra-communicator operations. FG-MPI does not currently support inter-communicators, dynamic process management functionality and remote memory access operations.

We are currently working on a new version of FG-MPI based on the latest MPICH 3.1 version. Since the release of mpich2-1.0.8p1 there have been several improvements to the MPICH middleware in terms of scalability, communication algorithms and the MPI runtime environment and process launching. We are integrating the existing FG-MPI functionality into the latest MPICH 3.1 version to leverage these improvements and as well to be able to keep FG-MPI synchronized with MPICH as it continues to evolve. We do not intent to enhance the current release of FG-MPI but rather replace it with a newer version once we have integrated FG-MPI into MPICH 3.1. FG-MPI has been used by a number of students at UBC for their research projects [7] and has proved to be stable and robust. It has been predominately tested on Linux and Mac OS platforms. At present, we offer best-effort email support for FG-MPI.

There are certain limitations that are inherent to implementation of MPI processes as user-level non-preemptive threads that share the same address space. One limitation of multiple co-located MPI processes is that global and static variables in the program can cause undesired side effects. Such variables, are no longer private to the MPI process and now become shared variables with the potential for read write conflicts. Since we are using non-preemptive threads access to these shared variables will be atomic but the order in which they are accessed depends on the scheduler. Users should be careful in using shared variables since the program is no longer purely message-passing and it may limit the way in which processes are mapped to OS-processes.

In general one should remove the shared variables, unless they are read-only, from the code to ensure the existing MPI program operates correctly in the FG-MPI runtime environment. The side effects of using shared global variables and static variables is an issue that has been well studied in other systems that allow multiple user-level threads per core. There are tools available for both FORTRAN [8] and C [1] that re-factor the source code to privatize global variables.

One effect of non-preemptive scheduling is that a computationally intensive process may block the progress of other co-located processes. We added a `MPIX_Yield` routine that allows one process to voluntarily yield to the scheduler. This can be used to balance the amount of computation and communication in the application. Execution of blocking file I/O by one process is another operation that can impede the progress of other co-located processes. One possibility is to structure the code so that I/O system calls are placed in an OS-process of their own. This can help avoid blocking other processes during I/O execution. Another scheme that is used for cooperative multitasking is to wrap the I/O library function so that the process initiating the I/O operation yields control to another process.

7 Issues

7.1 Known Issues

- `MPI_Group_xxx` routines are not supported in FG-MPI. We do, however, implement the `MPI_Comm_group` routine. For translation of ranks of MPI processes in one communicator to another, we provide an additional routine `MPIX_Comm_translate_ranks` (see Appendix 8.2) that may be used instead of `MPIX_Group_translate_ranks`. We discuss the scalability issues with groups and our reasons for not supporting group operations in [3].

- Communicator operations such as creation, duplication etc. on `MPI_COMM_SELF` are not currently supported.

- We have seen certain runs of the Cannon matrix multiplication program hang at times. We believe this may be due to a communication race condition between OS-processes on a single node in the current base version.

- The current release (fgmpi_1.0.0) of FG-MPI is based off `mpich2-1.0.8p1`, which uses all to all communication during MPI environment initialization. This type of communication is not scalable and has subsequently been fixed in later releases of MPICH. We ran into this problem during our experiments on the WestGrid cluster [4] and had to patch it will a new communication module. This patch is not part of the current release and therefore it will not scale to the extent reported in [4].

- We have a vanilla implementation of `MPI_Finalize` in that it executes a barrier to synchronize among the processes and then returns. It needs to be extended to properly de-allocate all the shared middleware structures. Since it is usually the last routine to be called before the program terminates, it does not affect the program execution.

- Issues related to I/O are discussed in Section 6. We have not tested FG-MPI with MPI-IO / ROMIO.

- FG-MPI does not currently support inter-communicators, dynamic process management functionality and remote memory access operations.

7.2 Unknown Issues

- Interactions of pre-emptive threads e.g. pthreads with FG-MPI are not known. Our testing has been limited to the case where only a single preemptive thread in an OS-process makes calls to the MPI middleware.

If you notice any other issues with FG-MPI, please let us know and we'll add them here.

8 Appendix

8.1 Prime sieve code example

Listing 3: An MPMD prime sieve program.

```
/************************************************************************
*    Sieve of Eratosthenes Prime Finder
*    Alan Wagner 2011
*    (1) sequential mapper and (2) random mapper
*    These mappers can be selected on command-line, without
*    re-compiling the code.
*    Three functions:
*       generator():  generates all odd numbers to pass into the sieve
*       sieveElement(): keeps first number (a prime), filters the rest
*       lastElement():  terminates the sieve
*    USAGE:
*      mpiexec -nfg 4 -n 4 ./primeSieve  -- generates nfg*n primes (16)
*
*     FG-MPI uses a packed assignment, for example the previous use creates
*     4 processes assigned as [0..3][4..7][8..11][12..15].  The use above has
*     0 as the generator so that process i generates the ith+1 prime.
*     This mapping is not the best as nfg is large since later processes
*     do not become active until after a large number of primes are found.
*
*      mpiexec -nfg X -n Y -genv FGMAP random ./primeSieve [seed] [cuts]
*
*      is a version that cuts the ring (as in a deck of cards)
*      the chain to produce random length sequences of chain to more evenly
*      distribute the processes. Enough cuts results in completely random
*      assignment.  These parameters allow one to experiment with the load-balancing.
*
*     Other examples:
*     mpiexec -nfg 100 -n 4 -genv SCHEDULER block -genv FGMAP random ./primeSieve [seed] [cuts]
*
*      uses the receive-side block scheduler (round-robin scheduler (rr) is
*      the default scheduler if none is specified).
*
*    TERMINATION: When the last process receives the last prime it sends a STOP
*     message to the generator that then passes it down the chain terminating the
*     sieve processes.   All stop and the last process prints the time.
*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include <string.h>
#include <assert.h>
#include <mpi.h>

#define MAXINT 1000000
#define FALSE 0
#define TRUE !FALSE
#define DATA_TAG 111
#define STOP_TAG 999

/* forward declarations */
int lastElement(int argc, char **argv);
int sieveElement(int argc, char **argv);
int generator(int argc, char **argv);
int who_are_my_neighbors(int rank, int size, int *prevproc_ptr, int *nextproc_ptr);

/****** FG-MPI Boilerplate begin *********/
#include "fgmpi.h"
/* forward declarations */
FG_ProcessPtr_t sequential_mapper(int argc, char** argv, int rank);
FG_ProcessPtr_t random_mapper(int argc, char** argv, int rank);

FG_MapPtr_t map_lookup(int argc, char** argv, char* str)
{
    /* Two mapping functions */
    if ( str && !strcmp(str, "random")) {
        return (&random_mapper);
    } else if ( str && !strcmp(str, "seq")) {
        return (&sequential_mapper);
    }

    /* return default mapper if FGMAP environment variable is not specified */
    return (&sequential_mapper);
}

int main( int argc, char *argv[] )
{
    FGmpiexec(&argc, &argv, &map_lookup);

    return (0);
}
/****** FG-MPI Boilerplate end *********/
```

```
int sieveElement(int argc, char **argv)
{
    int nextproc, prevproc;
    int rank, size;
    MPI_Status status;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    who_are_my_neighbors(rank, size, &prevproc, &nextproc);

    uint32_t num,myprime=0;

    int notdone = TRUE;
    while ( notdone )
      {
            MPI_Recv(&num,1,MPI_INT,prevproc,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
            if ( status.MPI_TAG == DATA_TAG )
    {
                    if ( myprime == 0 )
      {
                            myprime=num;
                            printf("%u, ",myprime);
        }
                    else if ( num % myprime )
      {
                            /* Not divisable by this prime */
                            MPI_Send(&num,1,MPI_INT,nextproc,DATA_TAG,MPI_COMM_WORLD);
        }
                    else { ; }
    } else if ( status.MPI_TAG == STOP_TAG )
    {
                    notdone = FALSE;
                    /* Send the terminate_TAG*/
                    num=1;
                    MPI_Send(&num,1,MPI_INT,nextproc,STOP_TAG,MPI_COMM_WORLD);
    } else
    {
                    fprintf(stderr, "ERROR ERROR bad TAG \n");
                    MPI_Abort(MPI_COMM_WORLD,rank);
    }
  }
    MPI_Finalize();
    return 0;
}


int generator(int argc, char **argv)
{
    int nextproc, prevproc;
    int rank, size;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Request request;
    MPI_Status  status;
    who_are_my_neighbors(rank, size, &prevproc, &nextproc);

    uint32_t myprime=2;
    uint32_t num=myprime+1;
    printf("%u, ",myprime);
    /* Set up a MPI_Irecv to stop the sieve */
    MPI_Irecv(&num,1,MPI_INT,prevproc,STOP_TAG,MPI_COMM_WORLD,&request);
    while ( num <= MAXINT )
  {
            int result=FALSE;
            MPI_Send(&num,1,MPI_INT,nextproc,DATA_TAG,MPI_COMM_WORLD);
            num+=2;
            MPI_Test(&request,&result,&status);
            if ( result == TRUE) break;
      }
    num=0;
    MPI_Send(&num,1,MPI_INT,nextproc,STOP_TAG,MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}


int lastElement(int argc, char **argv)
{
    int nextproc, prevproc;
    int rank, size;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Status status;
    uint32_t num,myprime=0;
    who_are_my_neighbors(rank, size, &prevproc, &nextproc);

    int notdone = TRUE;
    while ( notdone )
```

```
      {
            MPI_Recv(&num,1,MPI_INT,prevproc,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
            if ( status.MPI_TAG == DATA_TAG )
    {
                    if ( myprime == 0 )
      {
                            myprime=num;
                            printf("%u \n",myprime);
                            /* Send STOP_TAG to generator */
                            num=1;
                            MPI_Send(&num,1,MPI_INT,nextproc,STOP_TAG,MPI_COMM_WORLD);
        }
    } else
    {
                    /* Received a STOP_TAG */
                    notdone = FALSE;
    }
  }
    MPI_Finalize();
    return 0;
}

FG_ProcessPtr_t sequential_mapper(int argc, char** argv, int rank)
{
    int worldsize;
    MPI_Comm_size(MPI_COMM_WORLD, &worldsize);

    if ( (rank == MAP_INIT_ACTION) || (rank == MAP_FINALIZE_ACTION) )
        return (NULL);
    if ( 0 == rank ) return (&generator);
    if ( worldsize-1 == rank ) return(&lastElement);
    return (&sieveElement);
}

/* A temporary shared array that holds the random
   permutation of the processes created by random mapper.
   This array is only read by the co-located processes
   once to discover their previous and next neighbors
   and is de-allocated after that */
int *proc = NULL;

FG_ProcessPtr_t random_mapper(int argc, char** argv, int rank)
{
    int first=FALSE;
    int last=FALSE;
    int nfg, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPIX_Get_collocated_size(&nfg);

    int cuts= (size/nfg)-1;
    int seed=0;

    if ( rank == MAP_INIT_ACTION ) {
        if ( argc == 2 ) {
            seed = atoi(argv[1]);
        } else if ( argc ==  3 ) {
            seed = atoi(argv[1]);
            cuts = atoi(argv[2]);
        } else {
            printf("USAGE:  primeSieve [seed] [cuts]\n");
            exit(-1);
        }
        /* do the cuts and swaps */
        srand(seed);
        int i;
        proc = malloc(sizeof(int)*size);
        int *procswap = calloc(sizeof(int),size);
        for ( i=0; i<size; i++) proc[i] = i;
        for ( i=0; seed && i<cuts; i++){
            int ik1 = rand() % size;
            int ik2 = rand() % size;
            int k1 = (ik1 < ik2 ? ik1 : ik2);
            int k2 = (ik1 >= ik2 ? ik1 : ik2);
            /* swap */
            if ( (k2-k1) != 0 )
            {
                if ((i % 2) == 0 )
                {
                    memcpy(procswap,&(proc[k1]), sizeof(int)*(k2-k1));
                    memcpy(&(procswap[k2-k1]),proc, sizeof(int)*k1);
                    memcpy(&(procswap[k2]),&(proc[k2]), sizeof(int)*(size-k2));
                }
                else
                {
                    memcpy(procswap,proc, sizeof(int)*k1);
                    memcpy(&(procswap[size-(k2-k1)]),&(proc[k1]), sizeof(int)*(k2-k1));
                    memcpy(&(procswap[k1]),&(proc[k2]), sizeof(int)*(size-k2));
                }
                int *tmp=proc; proc = procswap; procswap=tmp;
            }
        }
        free(procswap);
        return (NULL);
    }
    if ( rank == MAP_FINALIZE_ACTION )
        return (NULL);

    assert (proc != NULL);

    if  ( proc[size-1] == rank ) { last=TRUE; }
    if  ( proc[0] == rank ) { first=TRUE; }

    if ( first )
        return (&generator);
    else if ( last )
        return (&lastElement);
    else
        return (&sieveElement);

}

int who_are_my_neighbors(int rank, int size, int *prevproc_ptr, int *nextproc_ptr)
{
    int prevproc = -1, nextproc = -1;
    char *mapstr = getenv("FGMAP");

    if ( mapstr && !strcmp(mapstr, "random")){
        static int times_called = 0;
        int i;
        times_called++;
        assert(proc);
        for ( i=1; i<size-1; i++) if ( proc[i] == rank) { prevproc = proc[i-1]; nextproc = proc[i+1];
            }
        if  ( proc[size-1] == rank ) { prevproc = proc[size-2]; nextproc = proc[0];}
        if  ( proc[0] == rank ) { prevproc = proc[size-1]; nextproc = proc[1];}
        if (times_called == size){
            free(proc);
        }
    }
    else {
        prevproc = (0==rank) ? size-1 : rank-1;
        nextproc = (size-1==rank) ? 0 : rank+1;
    }

    *prevproc_ptr = prevproc;
    *nextproc_ptr = nextproc;
    return (0);
}
```

8.2 Routines specific to FG-MPI

Following is a list of the `MPIX` routines, introduced by FG-MPI, to provide additional functionality related to co-located MPI processes. The boiler-plate code was described in Section 4.

MPIX_Yield

The calling processes performs a voluntary yield to the scheduler.
Prototype:

```
void MPIX_Yield(void)
```

MPIX_Usleep

The calling processes yields to the scheduler which blocks it for at least *utime* microseconds before placing it back on the run queue.
Prototype:

```
int MPIX_Usleep(unsigned long long utime)
```

MPIX_Get_collocated_size

Determines the number of co-located MPI processes in an OS-process, as specified by `nfg` flag with `mpiexec`.
Prototype:

```
int MPIX_Get_collocated_size(int *size)
```

`size` is the number of co-located MPI processes (integer)

MPIX_Get_collocated_startrank

Determines the smallest `MPI_COMM_WORLD` rank from among the co-located processes in an OS-process.
Prototype:

```
int MPIX_Get_collocated_startrank(int *startrank)
```

`startrank` is the smallest rank (integer).

MPIX_Comm_translate_ranks

Translates the ranks of MPI processes in one communicator to another communicator.
Prototype:

```
int MPIX_Comm_translate_ranks(MPI_Comm comm1,
                              int n, int *ranks1,
                              MPI_Comm comm2, int *ranks2)
```

The input parameters are `comm1` (communicator handle), `n` is the number of ranks in `ranks1` and `ranks2` arrays (integer), `comm2` (communicator handle). The output parameter is `ranks2` which is an array of corresponding ranks in `comm2`.

Following are zero-copy routines which allow passing of pointers to the data among co-located MPI processes without making intermediate copies [2].

```
int MPIX_Zrecv(void ** buf_handle, int count,
               MPI_Datatype datatype, int source,
               int tag, MPI_Comm comm,
               MPI_Status *status)

int MPIX_Zsend(void **buf_handle, int count,
               MPI_Datatype datatype, int dest,
               int tag, MPI_Comm comm)

int MPIX_Izrecv(void ** buf_handle, int count,
                MPI_Datatype datatype, int source,
                int tag, MPI_Comm comm,
                MPI_Request *request)

int MPIX_Izsend(void **buf_handle, int count,
                MPI_Datatype datatype, int dest,
                int tag, MPI_Comm comm,
                MPI_Request *request)
```

9 License

10 Acknowledgements

We acknowledge the support of Intel Corporation Inc., Mitacs Canada, and NSERC for the ongoing FG-MPI project.

References

[1] Elsa: An elkbound based C++ parser. Available from http://www.cs.berkeley.edu/~smcpeak/elkhound, accessed on March 17, 2014.

[2] Humaira Kamal. *FG-MPI: Fine-Grain MPI*. PhD thesis, The University of British Columbia, July 2013.

[3] Humaira Kamal, Seyed M. Mirtaheri, and Alan Wagner. Scalability of communicators and groups in MPI. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 264–275, New York, NY, USA, 2010. ACM.

[4] Humaira Kamal and Alan Wagner. A Signpost on the Road to Exascale. Available from http://archive.hpcwire.com/hpcwire/2013-01-15/british_columbia_researchers_notch_milestone_in_pursuit_of_exascale_computing.html.

[5] Humaira Kamal and Alan Wagner. Added concurrency to improve MPI performance on multicore. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 229 –238, Sept. 2012.

[6] Humaira Kamal and Alan Wagner. An integrated fine-grain runtime system for MPI. *Computing*, 96(4):293–309, 2014.

[7] Fine-Grain MPI. A complete list of FG-MPI publications are available from http://www.cs.ubc.ca/~humaira/fg_publications.html, accessed on March 17, 2014.

[8] Photran. An Integrated Development Environment and Refactoring Tool for Fortran. Available from http://www.eclipse.org/photran, accessed on March 17, 2014.