Computing manuscript No.
(will be inserted by the editor)

An Integrated Fine-Grain Runtime System for MPI

Humaira Kamal · **Alan Wagner**

Received: date / Accepted: date

Abstract Fine-Grain MPI (FG-MPI) extends the execution model of MPI to allow for interleaved execution of multiple concurrent MPI processes inside an OS-process. It provides a runtime that is integrated into the MPICH2 middleware and uses light-weight coroutines to implement an MPI-aware scheduler. In this paper we describe the FG-MPI runtime system and discuss the main design issues in its implementation. FG-MPI enables expression of function-level parallelism, which along with a runtime scheduler, can be used to simplify MPI programming and achieve performance without adding complexity to the program. As an example, we use FG-MPI to re-structure a typical use of non-blocking communication and show that the integrated scheduler relieves the programmer from scheduling computation and communication inside the application and brings the performance part outside of the program specification into the runtime.

Keywords MPICH2 · Function-level Parallelism · Fine-Grain · MPI-aware Scheduler · MPI Runtime.

1 Introduction

MPI has the reputation of being difficult to program (Gropp, 2001). Although some of the difficulties may be inherent to message passing, many of the popular parallel languages used on multicore processors also use message-passing. However, one notable difference between MPI and these parallel languages is the granularity of MPI processes. Processes in MPI are coarse grained and programmed to make it easy to match the number of processes to the available hardware, whereas many parallel languages support finer grain to match processes to the structure of the program.

H. Kamal · A. Wagner
Department of Computer Science, University of British Columbia, Vancouver, Canada.
Tel.: 604-822-3061
Fax: 604-822-5485
E-mail: kamal@cs.ubc.ca, wagner@cs.ubc.ca

By fine grain we mean function-level parallelism where processes may have tens of instructions rather than the thousands of instructions in coarse grain program-level parallelism. One can have function-size programs in MPI but it is not done because over-subscribing processes to processor cores is inefficient due to the context switch time between OS-level processes and because the OS scheduler is unaware of the cooperative nature of the processes. There are also OS limitations, even with lighter-weight OS processes, when there are too many processes on a compute node[1].

We introduced Fine-Grain MPI (FG-MPI) to investigate the extent to which function-level parallelism can be supported in MPI. FG-MPI extends the execution model of MPI to allow for interleaved execution of multiple concurrent MPI processes inside an OS-process. FG-MPI is integrated into MPICH2 middleware and supports hundreds and thousands of finer-grain MPI processes within an OS-process. There are more MPI processes than cores and we still can match the number of OS-processes to number of cores to maximize the parallelism, but now we can map multiple MPI processes to each OS-process. There is still "over-subscription", but it is now the FG-MPI runtime and scheduler that is managing the MPI processes inside each OS-process. This makes it possible to support the added concurrency that results when functions are processes.

When writing FG-MPI programs we noticed that we did not need to rely as much on non-blocking communication. Non-blocking communication makes it possible to have multiple outstanding messages that increases asynchrony and allows one to overlap communication with computation. This can reduce the idle time that results when processes are blocked waiting for a message to arrive. To avoid idle time the programmer tries to post messages as soon as possible, overlap that with some computation while periodically checking for new messages to process as well as posting new ones. Optimizing the messaging in this manner to reduce idle time and increase "slackness" breaks the cohesion of the program structure, adds complexity, and is less portable with respect to performance. Our key observation is that having multiple processes per OS-process with an MPI-aware scheduler provides an alternative way to achieve the performance without the complications to the program. The runtime scheduler acts as an abstraction device that the programmer can use to replace the hand-coded message scheduling parts of their program. As a result, the program is easier to understand and the performance-oriented aspect is outside of the specification of the application in the runtime where performance can be tuned with few, if any, changes to the code.

In the paper we describe two main design issues in FG-MPI that made it possible to support this MPI runtime model: (a) the use of coroutines and non-preemptive threads, (b) the integration of FG-MPI into existing middleware (MPICH2) rather than a layer running on top of MPI (Section 2). In Section 3, we describe the design of the scheduler and how it interacts with the MPI progress engine. Finally in Section 4, we give an example of using FG-MPI to re-structure a typical use of non-blocking communication and measure the effect of introducing additional concurrency and the overhead of the scheduler.

[1] We will be using the terms "node" and "machine" interchangeably in this paper to refer to a single computational node with multiple processor cores, operating under a single operating system.

Our hope is that the FG-MPI design and its proof of concept in a working system may provide a way for other MPI implementations to augment MPI to support this fine-grain model. Secondly we hope, by way of illustration in this paper, that extending MPI's runtime model to fine-grain can make MPI programming easier and a better overall solution that can seamlessly scale from a multicore node to multiple machines in a cluster.

2 FG-MPI Runtime

One major decision in the design of FG-MPI and the support of multiple MPI processes within an OS-process was the use of coroutines as a basis for non-preemptive scheduling of the processes.[2] Our system uses a modular approach and is capable of making use of different coroutine libraries through a configuration option. We currently support Toernig's coroutine library, and PCL (Portable Coroutine Library). Capriccio (Von Behren et al, 2003) and other systems have shown that coroutine-based threads have fast context-switching time, low communication and synchronization overhead and scale to support large numbers of threads. The benefits of coroutines at the language level are well-known and they are supported in many languages (Python, Lua) including parallel languages used on multicore (Erlang, Go Language). Cooperative multitasking can be difficult in general but for MPI the messaging-passing and calls to the middleware provide a natural yield point.

With regards to implementation, having non-preemptive processes was crucial. Since only one co-located process is active, it was possible to share the middleware without using locks and ensure that the middleware is in a consistent state between scheduling points. Previous attempts at *pre-emptive* thread-based MPI implementations (Demaine, 1997; Tang and Yang, 2001) have remained largely incomplete due to the complexity of managing synchronization primitives and the challenges in scaling. The challenges and overheads of thread-safety of MPI middleware are well known (Balaji et al, 2008; Thakur and Gropp, 2007) and it is an important problem but the use of coroutines circumvents the need for locks to support multitasking and the guaranteed atomicity made it easier to reason about the state of the middleware.

The second major design decision was integration of FG-MPI directly into MPI rather than an attempt to design a new implementation of MPI or to use coroutines and layer it on top of MPI. Adaptive MPI is an implementation of MPI that supports fine-grain processes, however, AMPI (Huang et al, 2003) implements the MPI library on top of Charm++ rather than directly into an existing MPI implementation. This requires their own implementation of MPI and the Charm++ runtime also needs a communication layer. This can result in an MPI sandwich, with MPI running on top of Charm++ which in turn runs over MPI. In FG-MPI, all MPI communication directly invokes the corresponding lower level MPI implementation of the call in the middleware, whereas in the layered approach only a subset of the MPI communication in the lowest layer is used. More importantly, a scheduler layered on top of MPI operates independently from the lower level MPI progress engine. The result is multiple independent control loops and schedulers, where it is difficult to coordinate their activities

[2] MPI processes sharing the same address space are referred to as *co-located* processes.

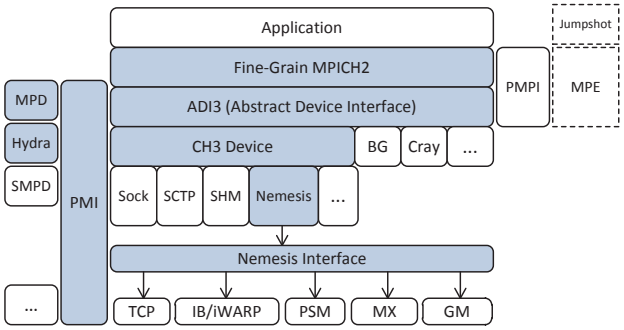

Fig. 1 FG-MPI Architecture. Shading shows the layers of MPICH2 that were augmented in the FG-MPI implementation. Figure adapted from Argonne National Laboratory, USA (2007).

with regards to the scheduling of asynchronous and synchronous messages. On the other hand, we integrated FG-MPI into the MPI library by extending the MPICH2 middleware. Figure 1 shows the integration of FG-MPI in the layered modular architecture of MPICH2. The first layer, below the application, defines the MPI API and implements user abstractions such as MPI data types and communicators, etc. The second `ADI3` (Abstract Device Interface) layer contains the progress engine and provides abstract middleware services to support the functionality of the first layer. Representation in this layer is in terms of MPI requests/messages and the functions for manipulating those requests. The third layer provides the device interface such as communication protocols and implements the ADI for the channels. FG-MPI uses the Nemesis `CH3` channel, as the communication subsystem (Buntinas et al, 2006). The Nemesis communication subsystem is designed for scalability and low shared memory communication overhead, making it suitable for our fine-grain system. Communication among processes in different OS-processes, on the same physical node, takes place through Nemesis's low latency, lock-free shared memory queues. The communication through shared memory employs optimizations to reduce L2 cache misses and techniques such as "fastboxes" to speed up message transfers. Communication across different physical nodes is enabled through its multi-network support and integration allows FG-MPI to leverage MPICH2's rich support for network fabrics in cluster environments. As well, we exploit the locality of MPI processes in the system and implement optimized communication between concurrent processes in the same OS-process.

One of the main considerations in FG-MPI was to support large amounts of concurrency through scalable sharing of MPI structures among the coroutines. To this end, a large number of MPI storage structures such as posted receive queues, unexpected messages queues, communicator and request pools are shared by the coroutines. Figure 2 shows a high-level picture of the FG-MPI runtime system. It shows a number of key components of the middleware and in the following sections we describe how FG-MPI augments the middleware to support large-scale concurrency with multiple MPI processes within an OS-process.

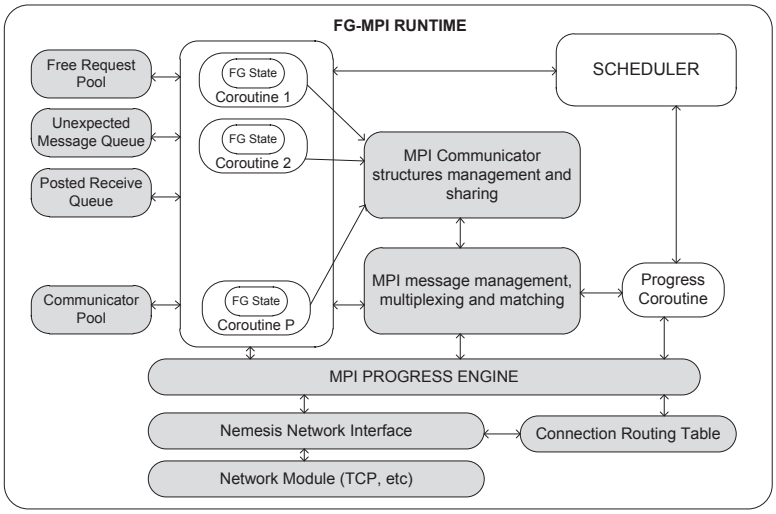

Fig. 2 FG-MPI Runtime System. Shaded regions show the middleware structures shared among the co-located MPI processes.

2.1 Separation of Namespaces

FG-MPI decouples the MPI processes from the OS-processes, which requires separating the namespace of the OS process's network point of attachment from the namespace of the MPI process ranks. In this section we describe how connection information is stored in MPICH2 and our approach to decoupling the MPI process names from the connection routing information.

In MPI, communicators are used to define separate communication contexts where communication within one group of processes cannot interfere with another group. In order for a process to communicate with another it must provide a communication context (i.e., communicator) and the local rank of the process in that communicator. Each process also needs to maintain connection information about the other processes with which it communicates. The MPICH2 implementation stores the connection state information in a virtual connection (`VC`) object, and creates one `VC` for every MPI process. MPICH2 maintains a virtual connection reference table (`VCRT`) for each communicator and translates the communication context and the local rank of the process to the appropriate `VC` object. The `VCRT` is stored as a dense array of pointers to the `VC` objects and is indexed by the local process rank in the communicator. The process rank, in this case, is tightly associated with the connection routing information.

In FG-MPI, we decouple the MPI processes from the hardware and have taken a different approach than MPICH2 to storing process connections. The MPI process groups are not tied to the `VCRT`, but instead we maintain two tables in each OS-process (a) a single `VCRT` for all the OS-processes in the execution environment that maps OS-processes to the virtual connections. The size of `VCRT` is proportional to the number of

Fig. 3 Structure of the message envelope.

OS-processes, (b) a process name table that uses `MPI_COMM_WORLD`[3] ranks and maps the MPI processes to OS-processes. All the MPI processes co-located within an OS-process are assigned the same point of attachment. There is a single shared instance of each of these tables per OS-process. There is, however, still the issue of efficiently storing the process group in each communicator (Balaji et al, 2009; Träff, 2010). As the process group is decoupled from the `VCRT`, this allows us to employ a number of techniques and representations to reduce the amount of space required to store the process groups. We describe the implementation of these techniques in FG-MPI in Section 2.3.

The separation of the two namespaces requires that we have a two-level hierarchy of ranks: (a) an OS-process namespace for virtual connection management and routing and (b) a namespace for MPI processes in `MPI_COMM_WORLD`. We emphasize the separation of these two namespaces because it is an example of the importance of naming in a distributed system (Saltzer, 1993). Although we have not yet considered process mobility, it simplifies that as well.

2.2 Message multiplexing

An interesting issue related to separation of namespaces is the message match header (envelope) in MPICH2. This header (see Figure 3) is appended to each message that is communicated between processes and contains the MPI tag, rank of the sender (source) process and the context ID of the communicator. In MPICH2, the message envelope does not contain the rank of the receiver (destination) process because that information is implicit from the OS-process identifier and hence the corresponding virtual connection used for message transmission. In FG-MPI, since there may be multiple MPI processes inside an OS-process, the destination rank of the process is necessary to de-multiplex the message from the OS-process network point of attachment to the MPI process. As a result we had to extend the message envelope as well as increase the packet header size to include the destination rank. This destination rank is the rank of the receiver in `MPI_COMM_WORLD` and uniquely identifies it.

A second issue is that the MPICH2 version we used as our code base used `int16_t` for storing the rank. This was not sufficient for supporting an environment with millions of MPI processes. We use 32-bit integers for both the source and destination ranks. There are trade-offs in extending the message envelope. MPICH2's motivation for using 16-bit ranks was to fit the entire message header in a 64-bit field to allow 64-bit instruction comparisons on platforms that support it and also slow communication links can benefit from a smaller header size (Balaji and Goodell, 2008). Although we have not done a low-level comparison we have not noticed any performance differences at the application layer as a result of our extension.

[3] We do not support MPI dynamic process management functionality.

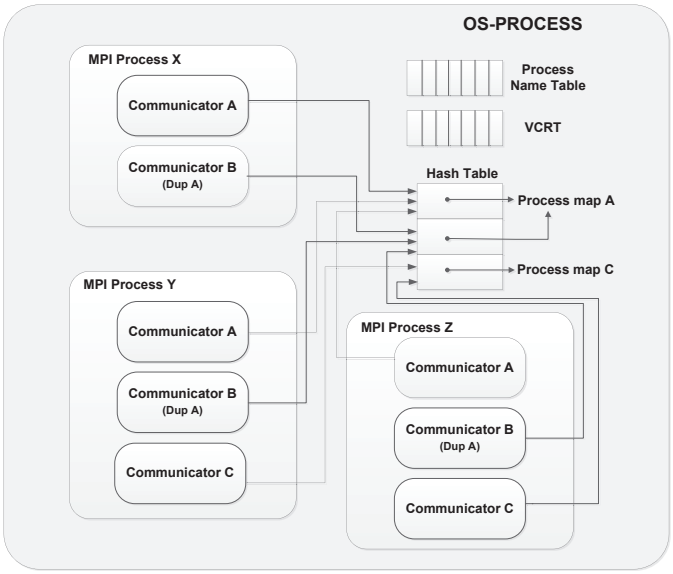

Fig. 4 Sharing of process group map inside communicators among co-located MPI processes.

2.3 Scalability of Communicators

FG-MPI's ability to expose large-scale concurrency allows for more opportunities to structure and group processes, which makes it more important to support efficient creation and storage of communicators. In this section we give a high-level overview of our group sharing and communicator creation techniques. More detail on the design of the different components can be found in Kamal et al (2010).

In MPI, all processes belong to groups and a communicator encapsulates the communication context and the process group in one object. As mentioned in Section 2.1, the MPI processes in FG-MPI are not coupled to the `VCRT` and we store a process group map in each communicator that maps the local rank of that process in that communicator to its rank in `MPI_COMM_WORLD`. In order for FG-MPI to expose large-scale concurrency the creation of communicators and storage of the process maps must be scalable and space efficient. A simple calculation shows that keeping separate process maps as arrays for P processes takes $\mathcal{O}(P^2)$ space, which is not feasible in a system with millions of MPI processes. We employ three techniques for communicator creation and storage that is both time and space efficient (Kamal et al, 2010): (1) We enable sharing of process maps among co-located processes that are part of the same communicator. (2) We use different memory reduction techniques for storage of process maps and provide a framework that allows selection of different storage structures as a configurable option. (3) We define new efficient algorithms for communicator creation `MPI_Comm_split` and `MPI_Comm_create` and an algorithm for creation of a globally unique context ID. The process map for `MPI_COMM_WORLD` is simply an identity vector and stored as a function.

Figure 4 shows an example of three co-located processes which share the maps of different communicators. During the communicator creation operation, one of the

members of the communicator creates the process map and stores a pointer to it in a global hash table. The context ID of each communicator is globally unique (Kamal et al, 2010) in our implementation and is used as the key for the hash table. Other co-located processes of the same communicator lookup the pointer to this process map in the hash table and cache it locally in their communicator structures. We thus store a single process map per OS-process for each communicator. Sharing is also enabled if a routine like `MPI_Comm_dup` is used to duplicate any communicator. In Figure 4, communicator `B` is created by duplicating communicator `A`. The entry in the hash table corresponds to the new context ID for communicator `B`, however, the process map is shared. Notice that the hash table lookup is only done once to access the pointer to the process map, during communicator creation, and then the pointer is cached locally. This allows us to use reference counting to keep track of how many MPI processes are sharing a map. If communicator `A` is freed, the entry corresponding to it in the hash table is removed, however, process map `A` remains until all references to it are removed.

MPI provides routines like `MPI_Comm_group` to access the process group associated with a communicator. We use a uniform definition for process maps inside MPI group and communicator structures. For certain routines like `MPI_Comm_group`, this allows us to use reference counting, in a way similar to that described for the `MPI_Comm_dup` routine, to share the process maps across communicators and associated MPI groups. In general, creation of MPI groups from existing groups through routines like `MPI_Group_incl` are not scalable, as these are local operations and store their individual maps. However, if a group is used to create a new communicator as in `MPI_Comm_create`, then we de-allocate all of the individual maps of the group members that are co-located and share a single map with the new communicator. However, as mentioned in Gropp et al (1999), routines to create new groups from existing ones are rarely needed and the use of `MPI_Comm_split` is recommended for creation of communicators. We discuss scalability of group management operations, details of memory storage for process maps and communicator creation algorithms in our past work (Kamal et al, 2010).

2.4 MPI Environment Initialization and Synchronization

MPICH2 uses external agents called process managers to launch and manage parallel jobs. These agents communicate with MPI processes through an interface called PMI (Process Manager Interface) via the `mpiexec` command. We extended the PMI to support an `nfg` (**n**umber of **f**ine-**g**rain) flag to the `mpiexec` command. Using `nfg` the user can choose how many MPI processes to run per OS-process in combination with the `n` flag specifying the number of OS-processes. For example, the `mpiexec -nfg 10 -n 4 myprogram` command specifies that there are ten co-located MPI processes in each of the four OS-processes. The FG-MPI runtime system inside each OS-process is initialized through a call to a function called `FGmpiexec()`, called from `main()`. At the beginning of the program execution there exists a single main coroutine (`MAIN_CO`), which communicates with the process manager and gathers the environment settings. The main coroutine plays an additional initialization and syn-

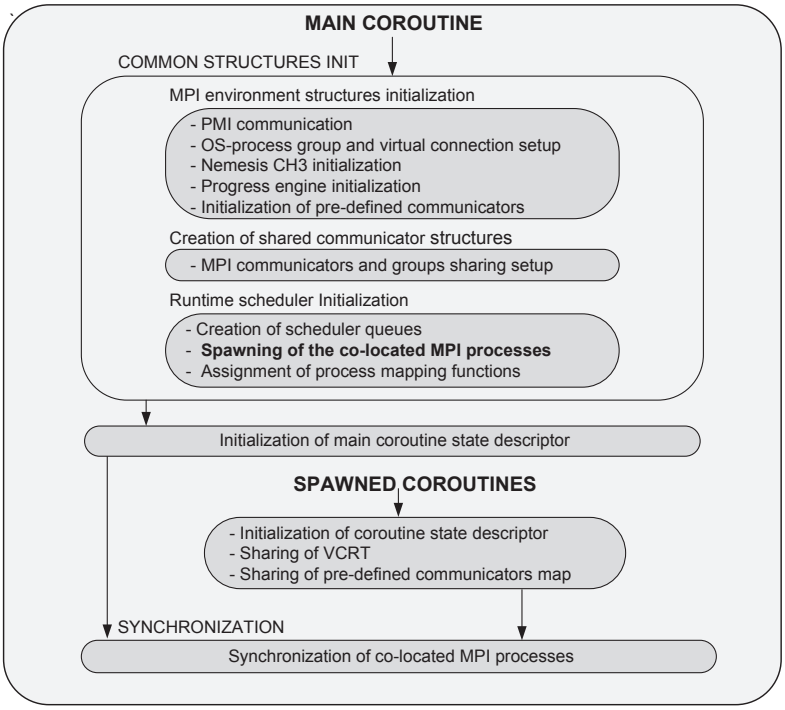

Fig. 5 Initialization of the MPI environment during `MPI_Init`.

chronization role in the `MPI_Init()` and `MPI_Finalize()` calls, but otherwise behaves identically to the other co-located coroutines during program execution. When the `FGmpiexec()` function is called by the main coroutine, it performs two functions: firstly, it initializes the MPI execution environment and secondly it spawns the other coroutines.

Figure 5 shows the major structures that are created and initialized by the main coroutine and subsequently shared among all the co-located MPI processes in an OS-process. These include the Nemesis communication subsystem queues and shared memory segments, progress engine initialization and creation of `VCRT` and communicator hash table. The MPI storage structures such as posted receive queues, unexpected messages queues and request pools are also global in the middleware and shared among the coroutines. Sharing of the request queues and progress engine not only enables scalability, it also allows one MPI process to cooperatively progress messages for another co-located MPI process. We describe message progression in more detail in Section 3. The last step of the MPI initialization is the creation of scheduler queues and the spawning of the co-located MPI processes in each OS-process.

We have two levels of mapping in FG-MPI. The first level of mapping is performed by the `mpiexec` command that maps OS-processes to different cores and machines. The `FGmpiexec` function provides a second dimension to the mapping and binds co-located MPI processes, within each OS-process, to different functions.

The binding of processes to functions can be defined in a general way through a user-defined function that takes a process's MPI_COMM_WORLD rank as its input parameter and returns a pointer to the function that MPI process will be executing. Each of these functions, which the processes are bound to, are written as regular MPI programs beginning with MPI_Init and ending with MPI_Finalize calls. The number of co-located processes is specified by the nfg parameter to mpiexec and the co-located MPI process ranks are assigned in consecutive block ranges of size nfg. The main coroutine yields at this point and each of the spawned processes are run by the scheduler and they initialize their coroutine state (described in Section 2.5) and share the common structures created by the main coroutine. All co-located MPI processes synchronize at the end of their MPI_Init calls and are queued for scheduling.

2.5 Coroutine State Descriptor

Each coroutine maintains a descriptor to store the state of execution of the MPI process associated with it. This MPI state information per coroutine is 1,328 bytes on a 64-bit machine and consists of the process's unique rank in MPI_COMM_WORLD, a pointer to the pre-defined built-in communicators, pointer to its context ID bitmap indicating the available and used communicator context IDs for this process and the state of initialization, i.e., whether it has called MPI_Init (and similarly for MPI_Finalize). Each coroutine has its own stack with a default size of 128 Kbytes. The lower bound on the stack size for the Toernig's and the PCL coroutine libraries is 1K bytes and 4K bytes, respectively. We currently have a fixed stack size for each coroutine, however, it is possible to extend FGmpiexec to provide the stack size as an argument.

Based on our experience we believe this type of integration is possible with other implementations of MPI. Finally, note that FG-MPI extends MPICH2 and the FG-MPI runtime is only set-up when there is more than one MPI process in an OS-process and it is possible to freely mix OS-processes with one process with those having multiple processes.

3 Integrated MPI Scheduler

We maintain a run queue and a blocked queue for co-located MPI processes inside each OS-process. Scheduling events inside the middleware invoke the scheduler, which according to the scheduling policy, blocks the current process or adds it back onto the run queue, and chooses the next process to resume. We provide a scheduler framework that allows us to add new policies as the need may arise. The selection of the scheduler is provided as a command line option to mpiexec. The most interesting aspect of the scheduler is its integration into the MPI middleware and interaction with events occurring inside the progress engine.

As Figure 2 shows, many of the key data structures in the middleware, such as the message queues, the request pools and the communicator pool, are shared among all of the co-located MPI processes. In FG-MPI, communication can be both internal

(among co-located processes) and external (between non-colocated processes). When a process makes an MPI call it progresses its request as far as possible. For example, consider the case of standard communication through `MPI_Send` and `MPI_Recv` between co-located processes. If the receiver process runs first, it queues its request in the posted receive queue and yields to the scheduler, which blocks it and resumes another co-located process to run. When the corresponding sender process runs, it matches the pre-posted receive request and complete its call. The sender sends a notification to the scheduler to unblock the receiver process and continues executing. However, in case the matching receive has not yet been posted, the send request is placed in the unexpected message queue and the sender yields to the scheduler so that the receiver can run. When the corresponding receiver executes, it finds and completes the matching request and continues execution.

For the case of communication between non-colocated processes, when a process sends a message it initiates a communication transfer over the external link to the receiver. Depending on the size of the send request, it may be able to complete the transfer and continue executing or it may require an acknowledgement from the receiver to complete the call as, for example, in a long message rendezvous transfer. In the latter case, the sender's message is queued among the pending sends in the progress engine, the sender yields to the scheduler and another co-located process runs. Symmetrically, a message arriving over the network at the message matching layer may complete a pending request or this may be an unexpected message, which will be queued until a matching receive request arrives.

In case of non-blocking MPI calls, the process does not block but continues executing until, for example, a corresponding wait is called. Depending on the state of the process's request at that point it may be able to complete the call or yield to another co-located process.

Most importantly, however, since the state of the progress engine is shared, MPI processes can cooperatively progress pending messages for other co-located processes and notify the scheduler. The scheduler, based on the notification may add processes to the run queue. An example of cooperation between co-located processes is that of a pre-posted receive request for which a ready-to-send (RTS) arrives to initiate the long message handshake. It is possible that the MPI process, which posted that receive request, is not currently executing, but a clear-to-send (CTS) can be sent by the currently executing process on its behalf.

One effect of non-preemptive scheduling is that a process that is a busy computing blocks the progress of all other co-located processes. One assumes that as long as the process is busy it is making progress, however, we did add `MPIX_Yield()` to handle cases when a fairer scheduling is needed. `MPIX_Yield()` is a FG-MPI specific routine which allows the calling process to voluntarily yield control to the scheduler.

Internal communication is optimized to take advantage of a single address space and it is an opportunity for the scheduler, depending on the type of the communication, to block one process until the communication can be completed after which both processes can proceed. For co-located processes, the scheduler follows a natural order where a send message schedules the corresponding receive process that can continue to progress the message chain. The communication among co-located processes involves a single `memcpy`, avoiding any intermediate system copies (Ka-

mal and Wagner, 2012). Similarly for external events, once a message is received and completed the corresponding MPI process is scheduled to continue advancing the computation. As well, for collectives such as barrier, the last co-located process completing the barrier can gang-schedule all of the processes in the barrier since they can all proceed (Kamal and Wagner, 2012).

In many cases we have found that even a very basic round-robin (RR) scheduler which keeps all the processes on the run queue is adequate. As the scheduling overhead is relatively small, as long as the co-located processes are easy to keep busy, the RR scheduler works well. One more additional advantage of the RR scheduler is that it is deterministic and gives more predictable executions. This is one of the nice properties of introducing a user-level scheduler instead of scheduling by the OS, where the programmer has less control over when the processes are de-scheduled. The deterministic property of RR has also been useful as a tool for debugging programs.

It is not sufficient to have only RR since there are simple cases where RR does extremely poorly. For example, consider the simple ring program, the forward communication of messages works well when it is the same order as the scheduling order, however, communication in the reverse direction is slow due to re-scheduling delay of all of the processes on the run queue. But more generally it was important to introduce a scheduling framework rather than one or more fixed policies. The policy ultimately depends on the application where ideally processes on the critical execution path are scheduled first. Finally, note that the scheduling policy is local to an OS-process and the runtime inside each OS-process can select its own scheduling policy.

One interesting problem that arises with the scheduler, that allows blocking of MPI processes, is indefinite waiting of the processes in the scheduler's block queue. Indefinite blocking can occur, for instance, when all of the co-located processes are blocked on a receive call, waiting for an external event, and there is no runnable process that can check for the arrival of messages and unblock those processes. One alternative is simply not to block all processes or to simply keep one or more processes on the queue. Deciding on whether or not to block a process depending on the state of other co-located processes is complicated. There are a large number of MPI calls and different scenarios that would need to be considered including analysis of corner cases involving collectives and the many different communication modes, where a change in the implementation could inadvertently cause problems.

However, there is a simple and scalable solution to this problem. We solved the potential indefinite blocking problem by introducing a progress coroutine in our runtime that comes into existence the first time an MPI process blocks on a receive call. Once created, the progress coroutine remains on the run queue. When called, this coroutine executes the progress-loop in the middleware and progresses pending incoming and outgoing messages. Whenever there is a receive that could be matched by a message from a remote process it ensures that we poll the external link for more data and on arrival of such a message wakes up the blocked process. As well, as discussed above, a clear-to-send (CTS) may be sent by the progress coroutine for a pre-posted receive. A progress coroutine avoids the checking that would have been necessary when blocking processes and also provides an easy way to measure the idle time and "slackness" during runtime.

4 Programmability and Non-blocking Communication

Non-blocking communication makes it possible to overlap communication with computation by increasing asynchrony in the system, but it breaks the cohesion of the program structure and is less portable with respect to performance. It typically involves structuring the code into stages and scheduling these stages inside the application code. Exposing and scheduling even modest number of stages in the application can result in complex code that is hard to read and maintain (Marjanović et al, 2010). One of the advantages of our approach is that it reduces the need for non-blocking communication.

Consider the program in Listing 1, showing a simple use of non-blocking communication, which tries to post as many messages as possible to keep the process busy.

```
int main( int argc, char *argv[] )
{  ...
  MPI_Irecv(...,recvRequests[0]);
  MPI_Irecv(...,recvRequests[1]);
  do {
    compute_local(...);
    MPI_Waitany(2, recvRequests, &index, recvStatus);
    switch(recvStatus->MPI_TAG) {
    case tag1:
      compute_A();
      MPI_Send(...);
      MPI_Irecv(...,recvRequests[index]);
      break;
    case tag2:
      compute_B();
      MPI_Send(...);
      MPI_Irecv(...,recvRequests[index]);
      break;
    }
  }while(...);
}
```

Listing 1 Scheduling communication and computation by non-blocking operations

There are three main parts to the program: (a) allocating and managing message request buffers, (b) checking for message completions and then processing the messages, (c) a compute part that may or may not depend on the messages sent and received. Some of the complexities in Listing 1 are:

(i) The compute and communication parts of the code are interleaved and the programmer needs to balance the computation with the polling of the link via the middleware.

(ii) The user needs to manage the request buffers for the multiple outstanding messages. The programmer also needs to be aware of all the different types of outstanding messages and how messages are matched. This often results in the use of `MPI_ANY_SOURCE` and `MPI_ANY_TAG`.

With FG-MPI, as shown in Listing 2, we can achieve a similar overlap by reorganizing the code fragment into three smaller processes: `compute_local()`,

process_A() and process_B(). As opposed to Listing 1, there are no non-blocking requests and associated structures in Listing 2 and no need to remember that the posted requests have to be checked for completion. Listing 1 has requests that are global over the entire program and no clear demarcation between different types of requests. FG-MPI places all of corresponding computation and communication code pertaining to one activity into one process. This adds to the cohesiveness of the program and makes it easier to read and change the code.

```
int main( int argc, char *argv[] ){
    FGmpiexec(&argc, &argv, &binding_func);
    return (0);
}
int process_A( int argc, char** argv ){
    MPI_Init(...);      ...
    do{
      MPI_Recv(...,tag1,...);
      compute_A();
      MPI_Send(...);
    }while(...);
    MPI_Finalize();
}
int process_B( int argc, char** argv ){
    MPI_Init(...);     ...
    do{
      MPI_Recv(...,tag2,...);
      compute_B();
      MPI_Send(...);
    }while(..);
    MPI_Finalize();
}
int compute_local( int argc, char** argv ){
    MPI_Init(...);      ...
    do{ ...
      if (...) MPIX_Yield();
    }while(...);
    MPI_Finalize();
}
```

Listing 2 Defining MPI processes as concurrent functions all mapped to the same OS-process. Each MPI process also calls MPI_Init and MPI_Finalize.

The purpose of the control loop in Listing 1 is to schedule different parts of the code based on the message events from MPI_Waitany(). In the FG-MPI version of the code there is no MPI_Waitany(). The control loop is now handled by the FG-MPI scheduler, which acts as an abstraction device, so that the programmer does not have to hand-code it into the program. In Listing 2, should process_A() now require we receive two messages rather than one, we only need to add another MPI_Recv(), however, for Listing 1 there are questions as to whether we need to introduce another case and tag and how it might be matched. In both listings it is important that the compute_local() code invoke the progress engine sufficiently often to not unduly delay the remaining computation and communication. In Listing 2, MPIX_Yield() can be appropriately placed when needed to provide an explicit de-scheduling point that automatically resumes at the proper place. Changing the rate at which the net-

work is polled in Listing 1 requires reorganizing the computation, which is yet another complication.

Expressing additional concurrency in the program gives us the opportunity to exploit it, however, it does require structuring the code and mapping MPI processes to functions as we now have the MPMD (Multiple Program Multiple Data) process model. In Listing 2, we encapsulate different receive actions in separate MPI processes and the sender process needs to use the appropriate receiver process's rank to trigger the right computation. As discussed in Section 2.4, `FGmpiexec` spawns the co-located MPI processes and the mapping of process ranks to functions is specified through the user-defined `binding_func`, which takes as input the `MPI_COMM_WORLD` rank of a process and returns a pointer to the function that the process is bound to. The extra-level of mapping gives us more flexibility in mapping to OS-processes and cores. As well, we can match the OS-processes to the cores to minimize the effect of OS-noise and not rely on the OS scheduler, which introduces yet another control loop that is unaware of the cooperative nature of the MPI processes (Ferreira et al, 2008).

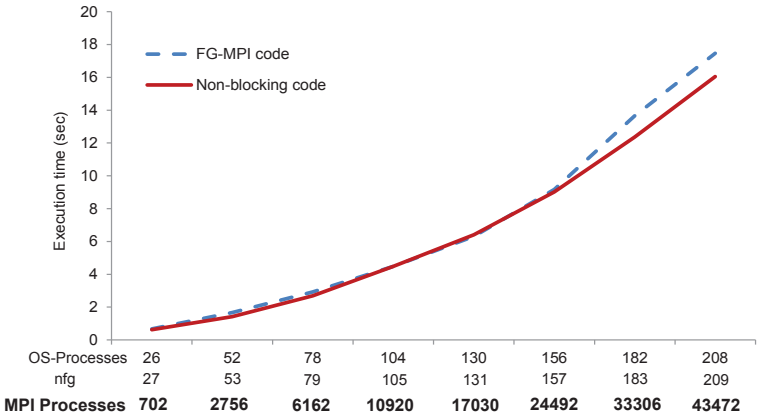

Fig. 6 Performance comparison of non-blocking code using `MPI_Waitany` with function-level concurrency in FG-MPI. Number of OS-Processes is same in both cases. In FG-MPI, the MPI processes are evenly distributed across the OS-Processes.

We created a benchmark program, similar to the codes in Listings 1 and 2, to evaluate the effect of exposing function-level parallelism and the overhead of the scheduler in FG-MPI. The MPI code of the benchmark launches N MPI processes as OS-processes, each of which uses non-blocking `MPI_Isend` and `MPI_Irecv` calls to pre-post send and receive requests for all the other processes. The MPI code carries out some local computation and then calls `MPI_Waitany` on these requests to progress them. The local computation and progression of the requests is done in a loop until all the requests are completed. The FG-MPI code also launches N OS-processes, however, each of these OS-processes contain $N+1$ co-located MPI processes. The MPI processes within each OS-process are organized as follows. (A) There is one MPI sender process that calls `MPI_Send` $N-1$ times to send to one receiver in each of the N OS-processes. (B) There are $N-1$ receiver processes that call `MPI_Recv`. Each

of these receivers is matched with the sender in one of the OS-processes. (C) There is one compute process that does local computation similar to `compute()` function in Listing 2. To allow computation and communication overlap, all messages sent in this benchmark are long messages that use the rendezvous protocol in MPICH2. The MPI code introduces asynchrony by pre-posting non-blocking operations and then manages scheduling of the computation and the communication through explicit calls to the middleware via the `MPI_Waitany` calls. The FG-MPI version, on the other hand, re-structures the code so that there is a separate MPI process for each of the receive calls. The scheduling of computation and communication is outside of the application specification and is managed by the runtime scheduler. In order to isolate the effects of introducing additional concurrency by mapping MPI processes to functions, we did not introduce any dependency between the computation and the communication in this benchmark.

Figure 6, shows the results of this benchmark. For the MPI non-blocking code, the number of MPI processes are equal to the number of OS processes, while in the FG-MPI code the number of MPI processes are a multiple of the `nfg` parameter and the number of OS-processes. The time reported is for ten iterations of the benchmark. Our results show that even with the introduction of more than 24,000 fine-grain MPI processes compared to 156 coarse-grain processes, the performance remains the same. As we increase beyond this to more than 43,000 processes, there is a small overhead of 8.7%. We are not sure of the reason for the deviation after 24,000 processes, however, the MPI code has an advantage in this benchmark, since it pre-posts all the send requests. Multiple of these pre-posted send requests can be progressed in the `MPI_Waitany` call. The FG-MPI code has a single sender process which makes `MPI_Send` calls one after the other. This benchmark, however, stress tests asynchrony at a large scale and shows that the overhead incurred by exposing function-level parallelism remains low.

5 Conclusions

Our runtime scheduler, through direct integration in MPICH2, is reactive to MPI events occurring inside the progress engine and its light-weight design enables definition of MPI processes as functions that can be flexibly mapped to OS-processes, cores and nodes. FG-MPI provides a task-oriented programming approach and support for MPMD that makes it easier, by exposing more concurrency, to overlap communication with computation. This relieves the programmer from scheduling computation and communication inside the application and focus on "what" needs to be scheduled rather than "how" to manage it.

References

Argonne National Laboratory, USA (2007) MPICH2: Performance and portability. MPICH2 flyer at Super Computing SC07.

Balaji P, Goodell D (2008) Using 32-bit as rank. Available from https://trac.mcs.anl.gov/projects/mpich2/ticket/42/, accessed on April 5, 2013

Balaji P, Buntinas D, Goodell D, Gropp W, Thakur R (2008) Toward efficient support for multithreaded MPI communication. In: Proc. of the 15th Euro PVM/MPI Users' Group Meeting, Springer-Verlag, Berlin, Heidelberg, pp 120–129

Balaji P, Buntinas D, Goodell D, Gropp W, Kumar S, Lusk EL, Thakur R, Träff JL (2009) MPI on a million processors. In: PVM/MPI, pp 20–30

Buntinas D, Gropp W, Mercier G (2006) Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem. In: Proc. of the 6th IEEE Intl. Symp. on Cluster Computing and the Grid, pp 521–530

Demaine E (1997) A threads-only MPI implementation for the development of parallel programs. In: Proceedings of the 11th International Symposium on High Performance Computing Systems, pp 153–163

Ferreira KB, Bridges P, Brightwell R (2008) Characterizing application sensitivity to OS interference using kernel-level noise injection. In: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, IEEE Press, pp 19:1–19:12

Gropp W (2001) Learning from the success of MPI. In: Proceedings of the 8th Intl. Conf. on High Performance Computing, Springer-Verlag, HiPC '01, pp 81–94

Gropp W, Lusk E, Skjellum A (1999) Using MPI - 2nd Edition: Portable Parallel Programming with the Message Passing Interface. Scientific and Engineering Computation Series, MIT Press

Huang C, Lawlor OS, Kale LV (2003) Adaptive MPI. In: Languages and Compilers for Parallel Computing, 16th International Workshop, Revised Papers, Springer, Lecture Notes in Computer Science, vol 2958, pp 306–322

Kamal H, Wagner A (2012) Added concurrency to improve MPI performance on multicore. In: 41st Intl. Conf. on Parallel Processing (ICPP), pp 229 –238

Kamal H, Mirtaheri SM, Wagner A (2010) Scalability of communicators and groups in MPI. In: Proc. of the 19th ACM Intl. Symposium on High Performance Distributed Computing, ACM, New York, USA, HPDC '10, pp 264–275

Marjanović V, Labarta J, Ayguadé E, Valero M (2010) Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In: Proc. of the 24th ACM Intl. Conf. on Supercomputing, ACM, New York, NY, USA, pp 5–16

Saltzer J (1993) On the naming and binding of network destinations. Network Working Group. http://tools.ietf.org/html/rfc1498, accessed on April 5, 2013

Tang H, Yang T (2001) Optimizing threaded MPI execution on SMP clusters. In: ICS '01: Proc. of 15th Intl. Conf. on Supercomputing, ACM, New York, pp 381–392

Thakur R, Gropp W (2007) Test suite for evaluating performance of MPI implementations that support `MPI_THREAD_MULTIPLE`. In: PVM/MPI, pp 46–55

Träff JL (2010) Compact and efficient implementation of the MPI group operations. In: Proc. of the 17th EuroMPI conference, Springer-Verlag, Berlin, Heidelberg, pp 170–178

Von Behren R, Condit J, Zhou F, Necula G, Brewer E (2003) Capriccio: scalable threads for Internet services. In: SOSP '19, ACM, New York, pp 268–281