An Integrated Runtime Scheduler for MPI

Humaira Kamal and Alan Wagner

Dept. of Computer Science, University of British Columbia, Vancouver, Canada
{kamal,wagner}@cs.ubc.ca

Abstract. Fine-Grain MPI (FG-MPI) supports function-level parallelism while staying within the MPI process model. It provides a runtime that is directly integrated into the MPICH2 middleware and uses light-weight coroutines to implement an MPI-aware scheduler. Our key observation is that having multiple MPI processes per OS-process, with a runtime scheduler can be used to simplify MPI programming and achieve performance without adding complexity to the program. The performance part of the program is now outside of the specification of the program in the runtime where performance can be tuned with few, if any, changes to the code.

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

1 Introduction

MPI has the reputation of being difficult to program [6]. 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 grain 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. 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 nodes 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 node.

We introduced Fine-Grain MPI (FG-MPI) to investigate the extent to which function-level parallelism can be supported while staying within the MPI process model [8,9]. To this end, FG-MPI augments the MPI middleware inside each OS-process with a runtime that supports hundreds and thousands of finer-grain MPI processes inside 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

J.L. Träff, S. Benkner, and J. Dongarra (Eds.): EuroMPI 2012, LNCS 7490, pp. 173–182, 2012.

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 stuck 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 program 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 also compare the performance of the scheduler one to the hand-coded one.

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 multicore inside the box to multiple machines and cores outside of the box.

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.[1] 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

[1] MPI processes sharing the same address space are referred to as *collocated* processes.

(Portable Coroutine Library). Capriccio [13] and other systems had 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 multithreading 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 collocated 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. There have been previous attempts at pre-emptive thread-based MPI implementations [5,11], but they have remained largely incomplete due to the complexity of managing synchronization primitives and challenges in scaling. The challenges and overheads of thread-safety of MPI middleware are well known [2,12] 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 [7] implements the MPI library on top of Charm++ rather than directly into an existing MPI. 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 would be MPI-aware but 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 with regards to the scheduling of asynchronous and synchronous messages.

We integrated FG-MPI into MPI by extending the MPICH2 middleware. Figure 1 shows the integration of FG-MPI in the layered modular architecture of MPICH2. The MPICH2 `ADI3` layer represents the data structures and functions that are provided by an implementation. Representation in this layer is in terms of MPI requests/messages and the functions for manipulating those requests. 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 (see Figure 2). Other structures that are integral to MPI are communicators and groups and their scalability and sharing is essential to FG-MPI. In past work [8] we discuss in detail how we share these structures and

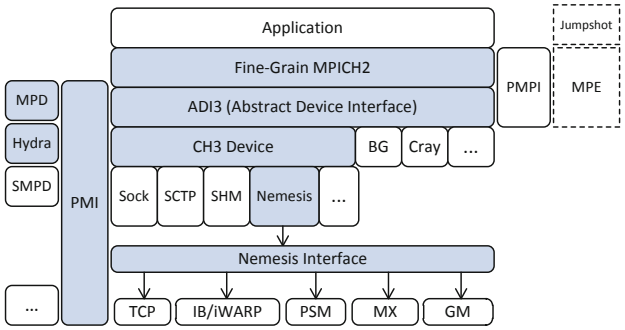

Fig. 1. FG-MPI Architecture. Shading shows the layers of MPICH2 that were augmented in the FG-MPI implementation. Figure adapted from [1].

scale to hundreds and thousands of MPI processes. FG-MPI uses the Nemesis `CH3` channel, which is a highly optimized communication subsystem that provides multi-network support [3] including fast shared memory communication between processes on the same node.

MPICH2 is optimized for OS-process level communication, and one interesting problem that arose as a result was that the message match header did not need to contain the rank of the destination process, as it is implicit from the OS-process identifier. In FG-MPI, since there may be multiple MPI processes inside the 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 match header as well as increase the packet header size to include the destination rank. Communication in Nemesis is tuned for better cache performance and 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.

Inside the middleware we maintain two separate tables: (a) connection routing table (point of attachments) and (b) process name table (`MPI_COMM_WORLD` ranks).[2] This separates the namespace of the point of attachment from that of the ranks. We emphasize the separation of these two namespaces because it is an example of the importance of naming in a distributed system [10]. Separating the namespace for the point of attachment and ranks is required to de-couple MPI processes from the hardware. Although we have not yet considered process mobility, it greatly simplifies that as well. The issue of naming also arises with respect to mapping where it is easier to map processes when the rank is used as a semantically pure identifier rather than an integer range from 0 to $N-1$.

Based on this 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.

[2] We do not support MPI-2 dynamics.

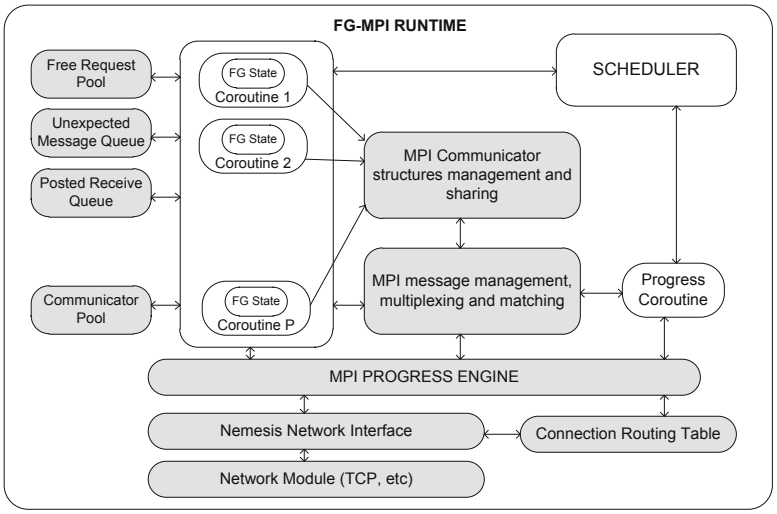

Fig. 2. FG-MPI Runtime System. Shaded regions show the middleware structures shared among the collocated MPI processes.

3 Integrated MPI Scheduler

We maintain a run queue and a blocked queue for collocated 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, request pools and communicator pool, are shared among all of the collocated MPI processes. In FG-MPI, communication can be both internal (among collocated processes) and external (between non-collocated processes) and supports the different types of MPI calls such as blocking, non-blocking etc. When a process makes an MPI call it will progress its request as far as possible. For example, if it is a send request, it may match a pre-posted receive from another collocated process and complete its call, or it may initiate a communication transfer over the external link to a receiver in another address space. A message arriving 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. Most importantly, however, since the state of the progress

engine is shared, MPI processes can cooperatively progress pending messages for other collocated processes and notify the scheduler. The scheduler, based on the notification may add processes to the run queue.

Another example of cooperation 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.

Internal communication is optimized to take advantage of a single address space, as well it is an opportunity for the scheduler, depending on the type of communication, to block one process until the communication can be completed after which both processes can proceed. For collocated 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 collocated processes involves a single `memcpy`, avoiding any intermediate system copies. 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, that last collocated process completing the barrier can gang-schedule all of the processes in the barrier since they can all proceed.

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 [9]. Because the scheduling overhead is relatively small, as long as the collocated 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 scheduler over preemptive threads (either pthreads or OS-threads) where the programmer has less control on when processes are de-scheduled. The deterministic property of RR has also proven 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. This was our motivation for introducing 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 does not have to be global.

One interesting problem that arises with the scheduler, that allows blocking of MPI processes, is deadlock. Deadlock can occur, for instance, when all of the collocated processes are blocked waiting for a external event. 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 collocated processes is complicated. There are a large number of MPI calls and different scenarios that would need to be considered and analyzed. However, there is a simple very scalable solution to this problem.

We introduced a progress coroutine in our runtime that comes into existence the first time an MPI process blocks on a receive. 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

As mentioned in Section 1, non-blocking communication adds to the programming complexity of MPI programs. Consider the program in Listing 1.1 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(2));
  do {
    compute_local(...);
    MPI_Waitany(2, recvRequests, ..., recvStatus);
    switch(recvStatus->tag) {
    case tag1:
      compute_A();
      MPI_Send(...);
      MPI_Irecv(...,recvRequests(1));
    case tag2:
      compute_B();
      MPI_Send(...);
      MPI_Irecv(...,recvRequests(1));
    }
  }while(...);
    ...
}
```

Listing 1.1. Scheduling communication and computation by non-blocking operations

As previously described 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 send and received. What are complexities in the above listing:

(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`.

As shown in Listing 1.2 with FG-MPI we can re-organize the program into three smaller processes: `compute_local()`, `process_A()` and `process_B()`. The `FGmpiexec` call binds the MPI process ranks to concurrent functions through a user-specified `binding_func` and initiates the runtime.

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

Listing 1.2. Defining MPI processes as concurrent functions all mapped to the same OS-process. Each MPI process also calls `MPI_Init` and `MPI_Finalize`.

As opposed to Listing 1.1, there are no non-blocking requests and associated structures in Listing 1.2 and no need to remember that the posted requests have to be checked for completion. Listing 1.1 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 makes it easier to read and easier to change the code.

The purpose of the control loop in Listing 1.1 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 rather than having to be hand-coded into the program. In Listing 1.2, should `process_A()` now require we receive two messages rather than one, we only need to add another `MPI_Recv()`, however, for Listing 1.1 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 1.2, `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 network is polled in Listing 1.1 requires reorganizing the computation, which is not as easy.

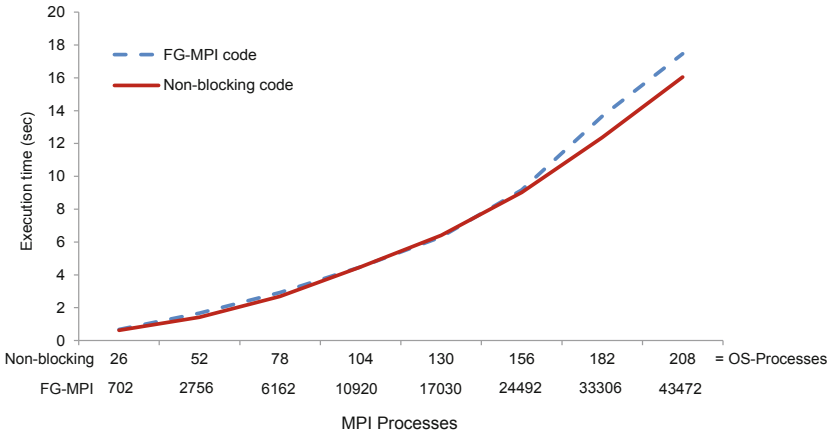

Fig. 3. Performance comparison of non-blocking code using `MPI_Waitany` with functional-level concurrency in FG-MPI. Number of OS-Processes is same in both cases. In FG-MPI, the MPI processes are evenly distributed across the OS-Processes.

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. We have two levels of mapping in FG-MPI. The first level specifies how OS-processes are mapped to the cores and nodes and the number of MPI processes mapped to each OS-process. This is done through an `nfg` (**n**umber of **f**ine-**g**rain) flag to the standard `mpiexec` command. The second level defines MPI processes as functions and this is done through a call to `FGmpiexec` in `main()`. Although we have an extra-level of mapping, this is outside of the application code and gives us more flexibility in mapping to OS-processes and nodes. 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 MPI processes. Finally, FG-MPI extends MPI so the programmer can manage as little or as much of the non-blocking communication as they wish.

We created a benchmark program, similar to the codes in Listings 1.1 and 1.2, to evaluate the overhead of introducing more MPI processes in FG-MPI. This benchmark introduced asynchrony on a much larger scale than shown in the two listings and the total amount of computation and communication increased with the number of OS-processes. In Figure 3, we compare the FG-MPI code with multiple MPI processes per OS-process with the non-blocking MPI code. 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%. A real-world example from the CoSMoS [4] project that uses FG-MPI and models emergent behaviour through thousands of MPI processes was presented in past work [9].

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. 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.

Acknowledgements. This work was supported in part by the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

References

1. Argonne National Laboratory. MPICH2: Performance and portability. In: SC 2007 Flyer (2007)
2. Balaji, P., Buntinas, D., Goodell, D., Gropp, W.D., Thakur, R.: Toward Efficient Support for Multithreaded MPI Communication. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 120–129. Springer, Heidelberg (2008)
3. Buntinas, D., Gropp, W., Mercier, G.: 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. IEEE Computer Society, Washington, DC (2006)
4. CoSMoS. Complex systems modelling and simulation infrastructutre, `http://www.cosmos-research.org/`
5. Demaine, E.: 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 (1997)
6. Gropp, W.D.: Learning from the Success of MPI. In: Monien, B., Prasanna, V.K., Vajapeyam, S. (eds.) HiPC 2001. LNCS, vol. 2228, pp. 81–94. Springer, Heidelberg (2001)
7. Huang, C., Lawlor, O.S., Kal, L.V.: Adaptive MPI. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 306–322. Springer, Heidelberg (2004)
8. Kamal, H., Mirtaheri, S.M., Wagner, A.: Scalability of communicators and groups in MPI. In: Proc. of the 19th ACM Intl. Symposium on High Performance Distributed Computing, HPDC 2010, pp. 264–275. ACM, New York (2010)
9. Kamal, H., Wagner, A.: FG-MPI: Fine-Grain MPI for multicore and clusters. In: 11th IEEE Intl. Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC) held in conjunction with IPDPS-24, pp. 1–8 (April 2010)
10. Saltzer, J.: On the naming and binding of network destinations. Network Working Group (1993), `http://tools.ietf.org/html/rfc1498`
11. Tang, H., Yang, T.: Optimizing threaded MPI execution on SMP clusters. In: ICS 2001: Proc. of 15th Intl. Conf. on Supercomputing, pp. 381–392. ACM, New York (2001)
12. Thakur, R., Gropp, W.: Test Suite for Evaluating Performance of MPI Implementations That Support `MPI_THREAD_MULTIPLE`. In: Cappello, F., Herault, T., Dongarra, J. (eds.) PVM/MPI 2007. LNCS, vol. 4757, pp. 46–55. Springer, Heidelberg (2007)
13. von Behren, R., Condit, J., Zhou, F., Necula, G., Brewer, E.: Capriccio: scalable threads for internet services. In: SOSP 19, pp. 268–281. ACM, New York (2003)