OMPI: Optimizing MPI programs using Partial Evaluation

Hirotaka Ogawa
Department of Information Engineering, The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan
ogawa@ipl.t.u-tokyo.ac.jp
http://www.ipl.t.u-tokyo.ac.jp/~ogawa/

Satoshi Matsuoka
Department of Information Engineering, The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan
matsu@ipl.t.u-tokyo.ac.jp

Abstract:

MPI is gaining acceptance as a standard for message-passing in high-performance computing, due to its powerful and flexible support of various communication styles. However, the complexity of its API poses significant software overhead, and as a result, applicability of MPI has been restricted to rather regular, coarse-grained computations. Our *OMPI* (Optimizing MPI) system removes much of the excess overhead by employing partial evaluation techniques, which exploit static information of MPI calls. Because partial evaluation alone is insufficient, we also utilize *template functions* for further optimization. To validate the effectiveness for our OMPI system, we performed baseline as well as more extensive benchmarks on a set of application cores with different communication characteristics, on the 64-node Fujitsu AP1000 MPP. Benchmarks show that OMPI improves execution efficiency by as much as factor of two for communication-intensive application core with minimal code increase. It also performs significantly better than previous dynamic optimization technique.

Keywords:

MPI, message passing, partial evaluation, SUIF, communication optimization, parallel computing.

1. Introduction

With the proliferation of MPPs and NOWs, standardized message passing interfaces such as PVM and MPI [6] are becoming increasing popular. They are used not only for writing new parallel applications, but also as a effective tool for parallelizing existing applications, as well as serving as a runtime message passing library for implementations of parallel languages, such as HPF.

Since MPI was announced two years ago, it is gaining increasing popularity, thanks to its powerful & flexible support of communication, such as different communication contexts via communicators, various synchronous/asynchronous communication modes, derived datatypes, group communications, etc. There have been a number of recent implementations of MPI as well. But, due to the inherent design of its API, the incurred software overhead is large, even compared to previous message passing libraries such as P4 or PVM. This is especially problematic when the hardware latency is low, because much of the benefits of fast networks are lost because of software overhead. This phenomenon not only applies to MPPs, but also to NOWs, where the availability of low-cost, low-latency networks such as the Myrinet is making low-latency communication possible.

As a result, application area of MPI has been somewhat restricted to regular, coarse-grained, and computation-intensive applications. In other words, attaining efficiency in fine-grained, irregular problems using MPI has been difficult. This is unfortunate, since standard message-passing libraries should encompass a wide variety of platforms and applications, including non-numerical applications, which are typically irregular and communication intensive.

There has also been a string of work that has focused on reducing software overhead in message passing as much as possible [9]. Notably, with Berkeley Active Messages, the incurred software overhead is in the order of several microseconds. The drawback is the relative lack of power and flexibility, and portability to some extent. Programming with native Active Messages library is much more difficult compared to

programming with MPI, because primitives are 'lower-level'. Furthermore, current Active Message does not support OS-level multithreading nor network heterogeneity well[1].

The question then is, would it be possible to have the best of both approaches, i.e., would it be possible to have a low-latency, high-performance message passing library, while retaining the flexibility and power of MPI? The answer is effectively *yes*---in this paper, we present our OMPI (Optimizing MPI) system, where much of the software overhead is eliminated with partial evaluation techniques, attaining performance which approaches that of Active Messages. C programs that contain MPI function calls are statically analyzed in order to determine which arguments are static, and specialized with respect to those arguments. Because the current partial evaluation techniques are not sufficiently powerful to eliminate all the software overhead, we propose a technique where partial evaluation is combined with selection of pre-optimized *template functions*. As a result, OMPI guarantees generality and portability of MPI programs, while allowing architecture-specific optimizations to be incorporated at compile-time. OMPI itself is also semi-portable, in that only template functions need to be reimplemented for a particular architecture. This is in contrast to traditional research on MPI implementation, where optimizations were highly architecture-specific.

To validate the effectiveness of our OMPI system, we performed some baseline benchmarks, and more extensive benchmarks on a set of application cores with different communication characteristics, on the 64-node Fujitsu AP1000 MPP. The basic point-to-point latency improved from 338 microsec. to 76 microsec., for communication intensive CG solver core, speedup of over a factor of two has been achieved. Even compared to traditional run-time optimization employing dynamic caching techniques, our OMPI system was consistently faster. The results show that our system is effective for various patterns of communication, significantly reducing the software overhead.

The rest of the paper is as follows: Section 2 analyzes the source of software overhead in MPI and opportunities of elimination. Section 3 describes our OMPI system. Section 4 presents results of the benchmark. Section 5 covers some related work, and we conclude in Section 6.

2. Analysis of Software Overhead in MPI and Opportunities of Elimination

We first analize the source of the software overhead, identifying problems pertinent to message passing libraries in general, and those that are specific to MPI. We then investigate the opportunities for optimization by removing the overhead when static information is exploited.

2.1. General Overhead

For most message passing libraries, information on messages can only be obtained at runtime. As a result, the receiver's buffer must either be somehow allocated dynamically, and the received message must be copied into the buffer. More specifically, the receiver has only two alternatives:

- If `receive` is posted before the actual message arrives, then the message is directly written into the receiver's buffers and not copied.
- Otherwise, the system allocates a buffer, and messages are written into the buffer; when `receive` is subsequently called, the contents of the system buffer is copied into the user's buffer.

Techniques to reduce the overhead typically has the sender specify the buffer address directly, such as is possible with the original Active Messages. However, software flexibility is lost as a result; for example, it will not be possible to filter message reception with message tags, as is possible with most message passing libraries. Also, programs must be strictly SPMD, in that addresses of functions must be the same among all the nodes.

2.2. MPI-Specific Overhead

The general philosophy of MPI design is to provide a rich set of features applicable to a variety of parallel applications. The characteristics of the resulting API of MPI is to have numerous arguments which are known to MPI only at runtime. For example, even the simplest point-to-point send/receive has six arguments, namely: 1) the address of the send/receive buffer, 2) message size, 3) message type, 4) ranks

of sender/receiver processes, 5) message tag, and 6) communicator. While such API complexity allows sophisticated support of different application communication patterns, the MPI library embodies the overhead of dynamic allocation of work area, error checking and handling, etc., in addition to the general overhead of message passing libraries.

```
MPI_Send(buf, count, type, dest, tag, comm)
     void *buf;              /* Pointer to Send Buffer */
     int count;              /* Data Count */
     MPI_Datatype datatype;  /* Data Type */
     int dest;               /* Rank (Target Process) */
     int tag;                /* Message Tag */
     MPI_Comm comm;          /* Communicator */

MPI_Recv(buf, count, type, source, tag, comm, status)
     void *buf;              /* Pointer to Receive Buffer */
     int count;              /* Data Count */
     MPI_Datatype datatype;  /* Data Type */
     int source;             /* Rank (Target Process) */
     int tag;                /* Message Tag */
     MPI_Comm comm;          /* Communicator */
     MPI_Status *status;     /* Status */
```

Figure 1: Example of MPI Calls `MPI_Send`, `MPI_Recv`

Here we identify the 4 sets of parameters, which we call *communication sets*, that are necessary for message passing. As described below, MPI incurs additional overhead compared to traditional, simpler message passing interfaces such as PVM, in order to obtain the communication sets. Later on we describe how OMPI utilizes the communication sets to optimize MPI code.

CommBuf:
Obtained directly.

CommSize:
Obtained from `datatype` and `count`. When `datatype` is a derived datatype, the processing becomes more complex because the datatype structure must be traversed to obtain the exact size of each unit of a datatype.

CommNode:
The physical node-id is obtained from the rank of the given process group specified by the communicator `comm`. When the process group is not the default, which contains all the nodes, some table lookup must be performed to obtain the rank --> node-id mapping.

CommTag:
Because MPI can specify both the `tag` argument and the tag specified by the communicator `comm`, the two must be combined to generate a unique tag.

2.3. Analysis of Opportunities for Eliminating Software Overhead

The problems with previous techniques have been that there are limitations in attempting to eliminate software overhead relying only on dynamic information available at runtime. By exploiting static information available at compile time, we can eliminate much of the software overhead as we will show in Section 3. Here, we analyze the various types of software overhead in detail, and discuss the opportunities of their elimination by utilizing static information.

The source of software overhead due to lack of exploiting static information can be categorized into the followings:

- Inapplicability of inter-buffer copy elimination techniques as seen in Active Messages.
- Cost of dynamic buffer allocation/deallocation management.
- Necessity of executing error checking and other dynamic conditionals.

- Cost of computing the communication sets themselves.

Thus, static determination of various MPI parameters will allow us to reduce the overhead by eliminating the computations costs incurred above. We further relate the cases when static information is known on the arguments and which of the costs above can be eliminated:

Case 1 --- When CommSize (datatype and count) are known:

- *Elimination of run-time checks and computation of message size:*
  This is especially effective for derived datatypes. Even for primitive datatypes, error checking can be eliminated.

- *Static allocation and re-use of message buffers:*
  When the DMA controller is employed for non-blocking transfers other than the ready mode, the system will require its own message buffer. By allocating or scheduling the re-use of such buffers statically, overhead involving buffer management can be drastically reduced.

- *Selection of optimal message passing procedures:*
  Various optimization techniques could be employed. As an example, packetization of small messages can be simplified; selection of push-based vs. pull-based messaging can be determined by message size[2]. When fast message passing hardware is available, such as Line-sending in AP1000 [4], sender buffering could be eliminated, etc.

- *Simplification/Elimination of Error Checking/Handling:*
  There are other benefits besides elimination of error checking/handling code. For example, argument range errors could be detected at compile time, increasing system robustness. When there is reliable transport for short messages, the heavyweight transport that handles all message sizes could be bypassed.

Case 2 --- When CommNode (the sender/receiver rank and the communicator) are known

- *Elimination of runtime computing of node-id:*
  Compared to simpler message passing libraries such as PVM, taking the correspondence of communication contexts is a significant part of MPI message passing, as mentioned earlier. Much of the overhead can be eliminated if both rank and the communicator are known. Even if the rank is not known, search procedures could be specialized to a fixed rank number, etc.

- *Conversion of self-sending into local memory copy/Elimination of handlers:*
  In order to take the correspondence between the sender and the receiver, communication contexts (usually called handlers) are created. For self-sending, which typically occurs in an SPMD program, message passing could be converted into local memory copy, and such handlers need not be created.

- *Simplification/Elimination of Error Checking/Handling:*
  For the same reasons mentioned above.

3. Optimizing MPI programs using partial evaluation

In order to exploit the optimization opportunities analyzed in Section 2, we propose OMPI: a system which optimizes MPI programs using partial evaluation techniques. OMPI works as a preprocessor to programs written in C + MPI, is semi-portable, and do not require customized C compilers, operating systems, or hardware. As the benchmarks will later show, our proposal eliminates much of the overhead analyzed so far, achieving the speed approaching Active Messages, while retaining the generality, flexibility, and portability of the MPI.

3.1. Overview of optimization architecture of OMPI

Optimization architecture of OMPI, which already has been implemented as a prototype on AP1000, performs the following optimizations automatically, without end-user interventions:

1. Static analysis is performed on the end-user source program written with C + MPI, obtaining static information of the arguments passed onto MPI.
2. Using partial evaluation techniques, the source program is specialized with respect to the MPI library functions.
3. The specialized source program is further optimized so that static information is fully exploited.

Unfortunately, we have found that the static analysis/partial evaluation techniques available today are not sufficiently powerful to effectively eliminate software overheads. Thus, we propose an alternative approach that requires machine-specific optimizations to be prepared by the MPI implementor for the particular machine. To be more specific, pre-specialized functions, called *template functions*, that correspond to each case of possible static/dynamic arguments, are prepared. Then, instead of automatic specialization of MPI functions, we instantiate the template functions with current function argument information, and inline expand the instantiated function. As we shall see, this technique works quite well in practice. Furthermore, it has the added benefit of being able to incorporate optimizations that are not possible with straightforward partial evaluation techniques. The drawback is that template functions must be tailored for each specific architecture; however, the burden of doing so could be substantially resolved by re-use of existing code and employing tool support.

3.2. Using SUIF for static analysis and partial evaluation

OMPI is currently built by extending the SUIF compiler toolkit proposed by Monica Lam et. al. at Stanford University [5]. The toolkit consists of a definition of an intermediate program representation called SUIF (from where the name is derived), and a set of extensible compiler passes. Each pass takes SUIF representation of the program, and outputs SUIF representations with additional information regarding the performed analysis, and/or a result of performing program transformation. The library is implemented as an object-oriented class framework using C++, and is extensible using inheritance. One could add new information to the SUIF node objects, and implement new passes by building on top of existing passes.

As mentioned above, OMPI is built as a preprocessor which passes the optimized MPI program to the backend C compiler of the target machine. We have found SUIF to be well-suited for the purpose, saving us considerable development time and achieving portability at the same time. Furthermore, as other research groups develop relevant optimizers, they could be integrated easily into our system to further optimize MPI programs.

3.3. Optimization Passes

Here we describe the optimization passes in more detail, as shown in Figure 2:

1. The source program written in C + MPI is transformed into SUIF by a tool called `scc` provided by the SUIF system.
2. An optimization pass called `peval` is applied to the SUIF representation, performing various static analysis and partial evaluation. `Peval` is OMPI's customized version of `porky` of SUIF, adding some features such as constant expression calculation and several specialized interprocedural analyses.
3. A template selection pass `tsel` is applied, selecting and instantiating the appropriate template functions.
4. `Peval` is further applied to the specialized SUIF program, propagating the static information inside the instantiated templates, and performing partial evaluation thereof.
5. The optimized SUIF program is converted back into a C program by using the `s2c` of SUIF.

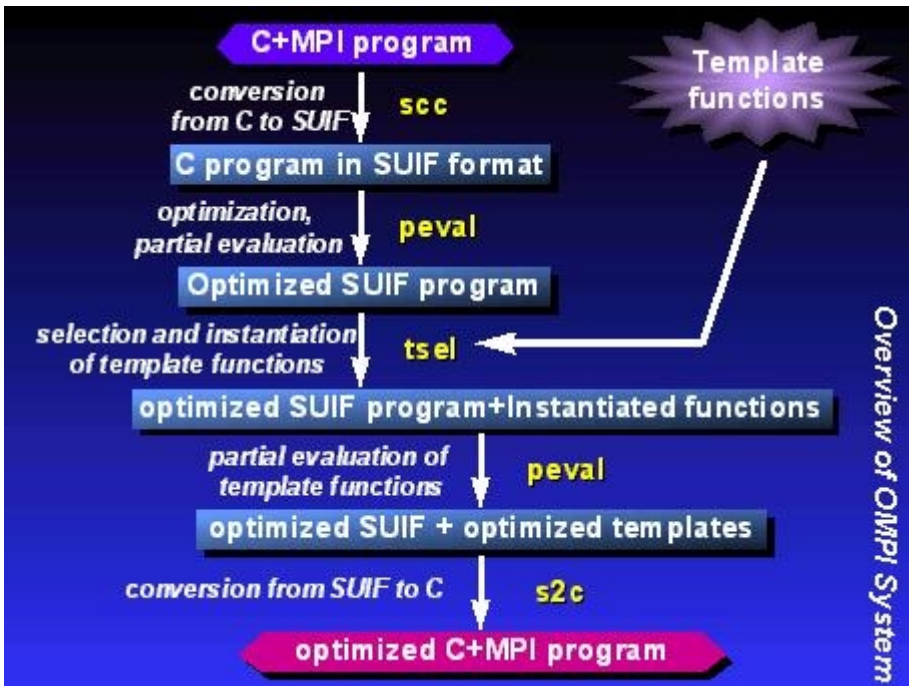


Figure 2: Overview of Optimization Passes

3.3.1. MPI Program Analysis/Optimization using `Peval`

`Peval` is our core analysis/optimization pass. It is implemented by re-using and extending upon many of the scalar optimizers available in SUIF. Traditional optimizations, such as constant folding, constant propagation, calculation of constant expression, forward propagation, induction variable detection, common sub-expression elimination, dead code elimination, unused symbols elimination, unused type definitions elimination, loop invariant expressions, loop invariant conditionals, control simplification, if hoisting, etc. are performed as source-to-source transformations. In our current system, we have already implemented interprocedural data-flow analysis[4] for basic datatypes, but array analysis is still in the (array analysis is important for determining *CommBuf* and *CommSize* inside loops.). As noted earlier, the `peval` pass is run twice, before and after the `tsel` (template instantiation) pass. The former is run to gather as much information as possible for `tsel`; because all the necessary arguments are passed to the MPI library via arguments and not through globals, we have found that current analysis and optimizations available in `peval` to be often sufficient for making as good a selection of templates as possible.

The post-`tsel` `peval` pass optimizes the internals of instantiated templates, which will be explained in the following sections.

3.3.2. Template Instantiation with `Tsel`

The `Tsel` pass scans the source program from the beginning, and when it encounters an MPI call, it performs the following actions:

1. Determine whether the arguments to the MPI call has been analyzed as a constant.
2. Select a template function depending on static/dynamic argument information.
3. Replace the MPI call with the instantiated template function call.
4. Append the instantiated template function.

In practice, however, forcing the MPI implementor to create template functions for every possible combinations of static/dynamic arguments is impractical even if we have semi-automated tools, due to combinatorial explosion. Instead, we employ a simpler policy for building and selecting template functions, based on communication sets described in Section 2. We group the arguments into those that determine *CommSize*, and *CommNode* and others, and only consider the entire communication set to be

static if all the member arguments are static; otherwise, the communication set is considered to be dynamic. We further partition the static case of *CommSize* into short and long messages, and *CommNode* into local and remote nodes. By eliminating meaningless combinations, we obtain 9 cases of template functions as seen in Table 1, where `X` is the name of an MPI function.

We note that not all the template functions need not be created. In fact, any function can be substituted by a conservative version w.r.t. its arguments being dynamic. For example, `X_long` could be safely substituted for `X_long_remote`.

Table 1: List of building template functions

			CommSize		
			Static		Dynamic
			Long message	Short message	
CommNode	Static	remote	`X_long_remote`	`X_short_remote`	`X_remote`
		local	`X_long_local`	`X_short_local`	`X_local`
	Dynamic		`X_long`	`X_short`	`X_generic`

The algorithm of selecting a template function depending on static/dynamic argument information is as below:

1. Identify whether *CommSize* and *CommNode* is static or dynamic.
   CommSize is determined to be static if both `datatype` and `count` are constants, otherwise dynamic. Likewise, *CommNode* is determined to be static if both `rank` and `comm` are constants, otherwise dynamic.
2. If *CommSize* is static, calculate the size of `datatype`. The actual `CommSize` value is obtained by the product of the size and `count`.
3. Identify *CommSize* with *long*, if the value is bigger than the architecture specific threshold. Otherwise, identify with *short*
4. If *CommNode* is static, the physical node-id is obtained from the rank of the given process group specified by the communicator `comm`. In OMPI runtime, the rank is numbered in order of the physical node-id, therefore, we can obtain the actual *CommNode* value easily.
5. Identify *CommNode* with *local* or *remote*.
6. According to the abstract value of *CommSize* (long or short or dynamic) and *CommNode* (local or remote or dynamic), select a template function from Table 1.

Instantiation of template functions selected by the above algorithm results in inline expansion of the function. In the expansion, all the static arguments appeared in the original MPI call is embedded inside of the instantiated template function definition. We should also note that even with conservative selections, we still may get the benefits of partial evaluation because the static value of the arguments will be propagated within the instantiated template functions in the subsequent `peval` pass.

3.4. An Example of MPI Optimization using Partial Evaluation

As an example, consider `MPI_Send` in Figure 3. Suppose that the preceding `peval` has output SUIF program [3] which includes the instantiated template call in Figure 3(a), where the message `count` is statically determined to be `10`, `type` to be `MPI_INT`, rank to be `2`, and communicator to be the default `MPI_COMM_WORLD`. In this case, `tsel` selects the template function `MPI_Send_Short_Remote`, which is optimized for sending short messages to a fixed node-id. The library call is transformed into code as shown in Figure 3(b), where `0001` is an ID which identifies each instantiation. Finally, the macro and the template function is expanded as seen in Figure 3(c). (Macro definition is for readability purposes.) The inline expanded template function has all the constants embedded within the function, which will be subsequently passed onto the next `peval` pass.

```
MPI_Send(buf, 10, MPI_INT, 2, tag, MPI_COMM_WORLD)
```

(a) original MPI call

```
MPI_Send_Short_Remote_0001(buf, 10, MPI_INT, 2, tag, MPI_COMM_WORLD)
```

(b) Transformed MPI call

```
#define MPI_Send_Short_Remote_0001(buf, count, type, dest, tag, comm)
    MPI_Send_short_remote_0001(buf, tag)

MPI_Send_short_remote_0001(_buf, _tag)
{
        /* preamble */
        const          int _count = 10;
        const MPI_Datatype _type  = MPI_INT;
        const          int _dest  = 2;
        const     MPI_Comm _comm  = MPI_COMM_WORLD;

        /* body */
        ....
}
```

(c) Specialized and expanded MPI call

Figure 3: An example of MPI optimization via Partial Evaluation

Instantiation of specialized template functions does increase code size. However, since the expansion of MPI functions is not recursive, the only potential problem is expansion of loops and recursive functions with embedded MPI calls. OMPI avoids this problem by limiting loop expansions to 4 iterations and not expanding mutually recursive functions more than once; as a result code increase is almost proportional to the original code size, and is minimal in practice so far.

4. Performance Measurements

In order to validate the effectiveness of our OMPI system, we performed some baseline benchmarks, and also more extensive benchmarks on application cores. The results show that our system is effective for various patterns of communication, and significantly reduces the software overhead, even compared with traditional optimization techniques.

We chose Fujitsu AP1000 as a target of our prototype implementation. As mentioned earlier, communication performance of AP1000 relative to its processor performance is considerably higher (25MBytes/sec node-to-node vs. 25Mhz Sparc IU + FPU). Thus, small software overhead will dominate loss in communication performance. AP1000 facilitates two modes of message communication. One is interrupt-based, and employs the DMA controller. Although the startup overhead is large, the send is nonblocking. The other is *Line-sending*, where values contained in a cache line could be sent directly with explicit cache flushing, eliminating the need for copying, interrupts, and DMA setup. On the other hand, the sender must block until the entire line is sent. Also, although the receive is transparently done into a ring buffer without interrupts, the value must be copied.

As a baseline for comparison, we chose MPICH [7], which is a public domain MPI implementation by Argonnne National Laboratory and Mississippi State University. In order to port MPICH onto AP1000, one only needs to implement to lower-level *ADI functions*. Since MPICH relies heavily on the performance of ADI functions, it is critical for performance comparisons that ADI functions are implemented efficiently. We reused the source code of tuned native AP1000 message passing libraries to achieve this requirement.

4.1. Comparison against dynamic caching optimization

As mentioned earlier, dynamic caching of arguments is a more orthodox and simpler implementation technique compared to ours. More specifically, each MPI function would have its own cache that holds the arguments of its previous calls, and when there is a cache hit, the following optimizations become possible in order to eliminate the overhead:

- Error checking could be eliminated.
- Parameter checks for other dynamic optimizations could also be eliminated.
- Message buffers could be reused

In order to validate the effectiveness of our approach against dynamic caching techniques, we also customized MPICH to incorporate such optimization for comparison purposes. To effectively implement dynamic cache in MPI calls, we employed the PCR (Persistent Communication Request) feature of MPI. The intended use of PCR is in inner loops, where the same MPI function is invoked repeatedly; there, instead of specifying the arguments each time in the iteration, a set of arguments could be registered with PCR using calls such as `MPI_Send_Init` and `MPI_Recv_Init`, and could be invoked repeatedly with `MPI_Start`. By using PCR, dynamic cache can be easily implemented as follows:

- *Preamble upon MPI function call:*
  Check if the arguments are cached; this is achieved with a *cache manager handle* as illustrated in Figure 4, and checking whether the stored arguments match or not. If it matches, it is a *hit*, and is a *miss* otherwise.
- *Cache Miss:*
  Register the called MPI function and the arguments with PCR. Create the communication handle, and store the arguments, the function ID, and a pointer to the PCR. Finally, invoke the communication with PCR.
- *Cache Hit:*
  Obtain the PCR from the cache manager handle, and invoke the communication.

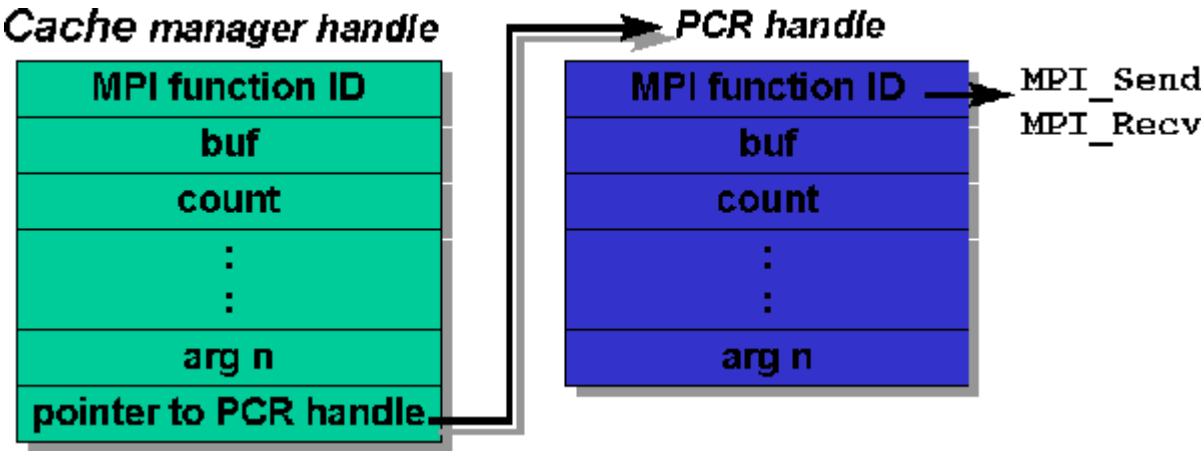


Figure 4: Cache manager handle

For derived datatypes, the argument comparison in the preamble could be costly, as the entire dynamic data structure must be traversed. The simple solution is not to cache such arguments; an alternative strategy is to use a fast but conservative matching function; for example, for systems where the operating system could trap on writes, any writes to a page containing a buffer could invalidate a match by setting some flag.

4.2. Implementation Specifics of OMPI on AP1000

In order to customize our system on AP1000, we need to create the template functions, and tailor the selection heuristics of `tsel`.

In the current prototype, template functions for the 9 cases involving the communication sets were hand-

created. Because we do not yet have tool support for creating template functions, only a small subset of MPI has so far been implemented. Here describe the specifics for `MPI_Send` and `MPI_Recv`. For `MPI_Send`, the templates for five cases have the specializations/optimizations described in Table 2. The remaining 4 cases (`X_long_remote`, `X_short_remote`, `X_long_local`, `X_short_local`) were created by combining the optimizations. Similarly, for `MPI_Recv`, the specializations/optimizations are described in Table 3, and likewise the remaining 4 cases were derived by their combination. Optimizations in other MPI functions are similar, which has given us confidence that at least some semi-automated tool could be created to greatly ease the task of template creation.

We also make a note here that, `X_generic`(i.e. unoptimized) template functions of OMPI are essentially the same as the corresponding MPICH implementations both in the robustness (e.g. error checking) and execution time.

Table 2: Specializing `MPI_Send` for AP1000

`MPI_Send`	`_generic`	do nothing (essencially same as MPICH)
	`_short`	use Line-sending method
		eliminate allocation/deallocation of send buffer
		eliminate error handling and overflow checking
		eliminate *CommSize* calculation
	`_long`	use DMA+Interrupt method
		eliminate *CommSize* calculation
	`_local`	use local copying from user buffer to system receive buffer
		eliminate allocation/deallocation of send buffer
		eliminate error handling and overflow checking
		eliminate *CommNode* calculation
	`_remote`	eliminate *CommNode* calculation

Table 3: Specializing `MPI_Recv` for AP1000

`MPI_Recv`	`_generic`	do nothing (essencially same as MPICH)
	`_short`	use Buffer-receiving method correspond to Line-sending
		eliminate allocation/deallocation of receive buffer
		eliminate error handling and overflow checking
		eliminate *CommSize* calculation
	`_long`	use DMA+Interrupt method
		eliminate *CommSize* calculation
	`_local`	use local copying from system receive buffer to user buffer
		eliminate allocation/deallocation of receive buffer
		eliminate error handling and overflow checking
		eliminate *CommNode* calculation
	`_remote`	eliminate *CommNode* calculation

We also tailor the selection heuristics of `tsel`. When there are multiple message transports, selection of the transport is dependent on *CommSize* and the characteristics of the underlying message passing architecture. On AP1000, in general it is better to employ Line-sending for short messages, whereas DMA+Interrupts is preferable for long messages. For the 64-node platform we employed, our tests showed that the threshold is at 60 Kbytes. This threshold is used for determining whether either `_short_` or `_long_` would be faster.

4.3. Baseline Performance Comparisons

We first compare the baseline performance by conducting the basic ping-pong benchmark. In order to obtain realistic figures for a multiprocessing environment, message reception is via interrupts and not polling. We tested both the latency and throughput; here, we only introduce the latency figures (Figure 5), as the throughput basically converges to be identical at approximately 60 Kbytes of message size (which was chosen as the threshold).

The leftmost two columns indicate the performance of native AP1000 message passing library for sending a null message. DMA requires 193 microsec. vs. 37 microsec. for Line-sending, and we can see, both hardware and software setup time of DMA is significantly greater. Line-sending latency is close to what one obtains from Active Messages (Latency for polling-based Active Messages on AP1000 has been reported to be approximately 9 microsec.[8]).

The middle three columns are dynamic cache optimized MPI. The software overhead for the initial setup time (i.e., cache miss) is significantly greater compared to DMA, but for cache hits involving basic datatypes, both software and hardware overhead is reduced significantly, coming close to that of DMA. Software overhead reduction is mainly due to elimination of error checks and dynamic computation of target nodes from *communication set*, as PCR allows such communication contexts to be cached and passed almost directly to the underlying DMA routine. On the other hand, software overhead for derived datatype is significant, due to the interpretation/traversal overhead for cache comparison check mentioned earlier, nullifying the gains obtained with caching.

The rightmost columns are OMPI results. The 'Generic' column is when there is no static information available; 'CommNode' and 'CommSize' indicate the cases where the respective communication sets are identified to be static; and 'Both' means that both are known statically. Here, we see that even with partial information, our optimizations result in significant overhead reduction. In particular, when *CommSize* is known, since our message is below the 60Kbytes threshold, the Line-sending was selected, which greatly reduces the mandatory hardware latency (from 112 microseconds to 24 microseconds). For the 'Both' case, the results are dramatic: both hardware and software overhead have been reduced to 1/4 of the unoptimized generic case, down to 76 microsec. from 338 microsec. Although the software overhead is still larger than the native AP1000 message passing library, we strongly believe that we could close this gap with further improvements in partial evaluation.

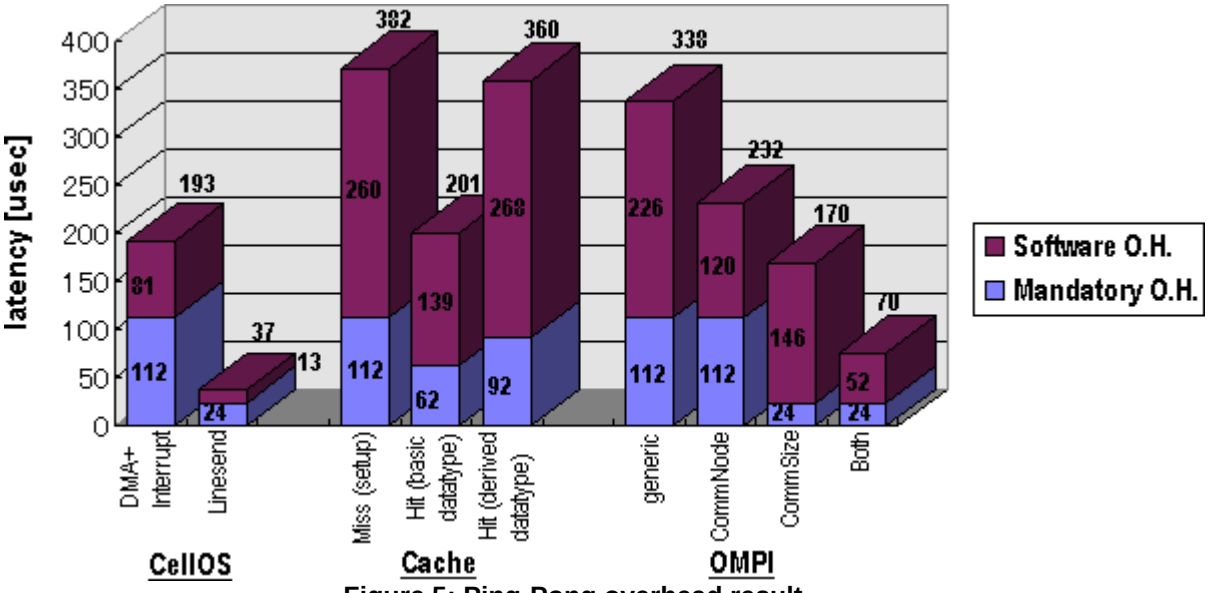


Figure 5: Ping-Pong overhead result

4.4. Numerical Applications Core Benchmarks

We next compare the performance of MPI optimizations in a typical numerical applications core. We chose three benchmarks with very different communication patterns and computation vs. communication

ratio. They are as follows:

- **MM**: 1024 x 1024 Matrix Multiply based on (Cyclic(16), Cyclic) submatrice distribution. Communication size is 16 Kbytes, communication pattern is regular, and is highly compute intensive.
- **LU**: Linpack 1000x1000 matrix LU-factorization with single pivot row selection and (Cyclic(8), Cyclic) distribution. Communication size is 1 Kbytes, and communication pattern cannot be entirely determined statically due to run-time pivot selection. Less compute intensive compared to **MM**.
- **CG**: Conjugate Gradient solver over a 128 x 128 sparse matrix. Dot distribution is employed. Communication size is one floating number, communication pattern is regular, and is highly communication intensive.

We compared the execution times of the above benchmarks for generic (unoptimized), dynamic cache optimized, and OMPI. Furthermore, the execution times were categorized into time spent for computation, time spent within MPI executing library code, and time spent within MPI waiting for barrier and communication synchronization. Figure 6 presents the results. For **MM**, speedup is minimal, because the speedup obtained within MPI library is only a very small fraction of the entire execution time. Still, OMPI won by a small margin. For **CG**, the other extreme, OMPI was able to win by a significant factor over both generic and cache implementations, with 53% reduction in execution time.

An interesting result was obtained for **LU**; we initially speculated that this benchmark would be disadvantageous for OMPI, as the communication pattern cannot be determined at compile time. The result surprisingly indicated otherwise, significantly improving over the generic MPI and winning over dynamic cache MPI by a notable factor. Closer analysis revealed that the PCR cache was being invalidated due to irregularity in the communication pattern of LU, and also that the cost of cache management was adding considerable overhead. By tuning the dynamic cache optimization, e.g., by not relying on the PCR, such overhead could be reduced. Still, the benchmark shows that, even for irregular communication, our partial evaluation strategy eliminates considerable portion of software overhead of MPI.

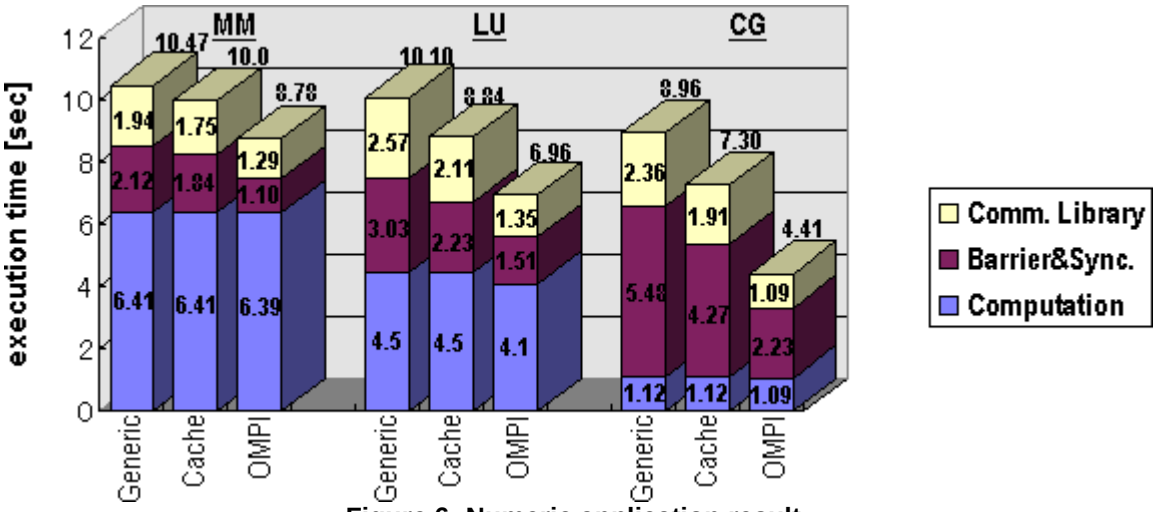


Figure 6: Numeric application result

5. Past Reports of Fast MPI Implementations on MPPs

As far as we know, all the efforts to lower communication latency in MPI have been to tune the libraries so that their software overhead becomes minimal, given the fact that all the arguments are dynamic. None have employed static compiler techniques to improve performance.

Franke and Hochschild report [1] that the lowest latencies achieved with their MPI implementations on SP1 and SP2 are 30 microsec. and 40 microsec. respectively, with throughputs of 9 Mb/sec. and 35Mb/sec. However, the figures are based on polling, and do not apply to multi-process environments in

practice. In such as case, interrupt-based implementation must be used, where the overhead increases to 200 microsec.

MPI/DE [2] is a implementation of MPI on NEC's Cenju-3 MPP, where the underlying operating system is Mach 3.0. Because Mach has kernel-supported threads which could be notified via kernel upcalls, interrupt handling could be made faster. Konishi reports that lowest latencies are 60 microsec., 90 microsec., and 140 microsec. for polling-based, upcall-based, and standard interrupt-based implementations, respectively. However, because Cenju-3 that network DMA controllers operate in physical space while MPI/DE works in logical space and no user-level facilities are provided for performing the necessary translation, the messages must always be copied between user buffers and kernel buffers, incurring significant overhead. Furthermore, polling-based implementation is not useable in a multi-process setting, and upcall optimization is not portable in that it relies heavily on Mach functionalities.

Sitsky describes the implementation of MPI on AP1000 [3], where the underlying CellOS is slightly modified, and the broadcast network is utilized to lower group communication. Still, the latency are reported to be 171.8 microsec. and 64 microsec. for the in-place method similar to DMA+Interrupts and the protocol method respectively, and the throughput are 2.69 Mbytes/sec. and 14.83 Mbytes/sec for the in-place method and the protocol method respectively, indicating that hardware performance is not well utilized.

6. Discussions and Future Work

We have presented OMPI, a compile-time optimizer for MPI that eliminates much of the communication overhead using partial evaluation techniques. Performance benchmarks show that, even compared to traditional dynamic optimization techniques, our system is faster by substantial margins, especially for communication-dominant computations in an high-performance hardware interconnect setting.

There are still technical issues to be worked out as future research. Some issues we share in common with elaborate compilation techniques, such as separate compilation and debugging of optimized code. We believe that solution techniques in advanced compilers could be applied. Furthermore, since the user can always fall back to non-optimized version of MPI, it is possible for the user to fully debug his code before applying OMPI.

There are other static optimization techniques that could be applied. For example, we could perform more extensive static analysis, such as variable range checking, which would be effective in eliminating many of the checks even if we do not have full static information. Another is communication rescheduling; even a simple algorithm would be effective in grouping the communication, and applying techniques such as message vectorization and piggybacking [10]. More elaborate communication rescheduling techniques will allow further optimizations. We are also considering combining our techniques with dynamic optimization techniques.

One of the current technical challenges with our MPI optimization is how to ease the effort of implementation of template functions. Currently, we are taking three approaches to this problem. One is classic software engineering, that is to separate out the machine-independent optimizations from machine-specific optimizations. Another is semi-automated tool support: a software tool could aid the user in specializing his code, by semi-automatically generating the code the user starts out with, given the static/dynamic distinctions of the arguments. The tool could also be supplied with the characteristics of the underlying hardware and operating system (latency/bandwidth of different network interfaces, polling/interrupt-driven/buffered, single/multiprocessing, etc.) and further select or eliminate parts of code, in a similar manner as the current partial evaluator.

Another interesting approach is to implement the core functionality of a subset of MPI, and implement more sophisticated functionalities be implemented in terms of the core subset, and optimized via our MPI optimizer by expanding them all out with partial evaluation. By taking care not to implement MPI functions to be mutually recursive, such recursive expansion via partial evaluation should terminate in a few iterations. Indeed, the MPI standard defines an official subset, whereby other MPI functionalities could be implemented---we must investigate whether the official subset will be just enough for our purpose, in terms of its functionality and the speed of the resulting implementation.

Finally, it is an interesting research and design issue how much of the new features currently proposed for MPI 2.0 could be superseded by optimization techniques such as ours. Indeed, some of the new proposals are fundamentally beneficial, such as threads, but there could be some features which might not be necessary, and would otherwise will have unsatisfactory effect on the current execution model and/or the MPI API.

7. Acknowledgement

This research was conducted under a grant from the Real World Computing Partnership of Tsukuba, Japan. We thank the valuable comments from Frank O'Caroll, Marc Snir, Yutaka Ishiakwa, Mitsuhisa Sato, Satoshi Sekiguchi, Umpei Nagashima, and Nayeem Islam.

References

[1] H. Franke, P. Hochschild, P. Pattnaik, J. Prost, and M. Snir. MPI on IBM SP1/SP2: Current status and future directions. In *Proceedings of 1994 Scalable Parallel Libraries*. October 1994.

[2] K. Konishi, Y. Takano, and A. Konagaya. MPI/DE: and MPI library for Cenju-3. In *MPI Developers Conference*, University of Notre Dame, June 1995.

[3] D. Sitsky and K. Hayashi. Implementing MPI for the Fujitsu AP1000/AP1000+ using Polling, Interrupts and Remote Copying. In *Proceedings of Joint Symposium on Parallel Processing '96*, University of Waseda, Japan. June 1996.

[4] T. Shimizu, T. Horie, and H. Ishihata. Low-latency message communication support for the AP1000. In *Proceedings of 19th Annual International Symposium on Computer Architecture*. pp. 288-297, May 1992.

[5] R. Wilson, R. French, C. Wilson, S. Amrasinghe, J. Anderson, S. Tjiang, S-W. Liao, C-W. Tseng, M. Hall, M. Lam, and J. Hennessy. *The SUIF Compiler System*. Computer Systems Laboratory, Stanford University, 1994.

[6] Message-Passing Interface Forum. MPI: A message passing interface standard, version 1.1. June 1995.

[7] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.

[8] K. Taura, S. Matsuoka, and A. Yonezawa. An Efficient Implementation Scheme of Concurrent Object-Oriented Languages. In *Proceedings of 4th ACM SIGPLAN Symposium on Principles and Practive of Parallel Programming*. pp. 218-228, May 1993.

[9] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechnisim for Integrated Communication and Computation. In *Proceedings of the 19th International Symposium on Computer Architecture*. pp. 256-266, May 1992.

[10] N. Islam, A. Dave, and R. Campbell. Communication Compilation for Unreliable Networks. In *16th International Conference on Distributed Computing Systems*. May, 1996.

Footnotes

[1] Some of these issues will be addressed with Active Messages 2.0.

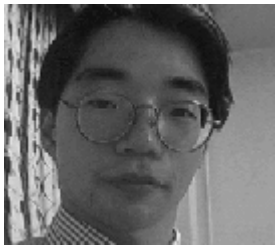
[2] This optimization has been suggested by Marc Snir, but has not been implemented yet in our current

system.

[3] As noted earlier, the SUIF representation is not textual; the described code has been retransformed back to C for readability.

[4] Interprocedural analyses can be easily implemented, using the Global Symbol Table of SUIF, which provides method to access symbols and procedures across program hierarchy.

Author Biographies



Hirotaka Ogawa

Received the B.E. degree from Dept. of Mathematical Engineering and the M.E. degree from Dept. of Information Engineering, the University of Tokyo. Currently doctral candidate in the Department of Information Engineering, at the Graduate School of Engineering, the University of Tokyo. His research interests include implementation technologies of Widely-Distributed/Parallel programming languages and systems, and parallel algorithms.

Satoshi Matsuoka

Received the B.S., M.S., and Ph.D. degrees from Dept. of Information Science, the University of Tokyo. Currently Assistant Professor at Graduate School of Information Engineering, the University of Tokyo. His current interests are implementation technologies of Object-Oriented/Parallel/Reflective programming languages and systems, and graphical user interface software. He had served as a Secretary for SIGPRG (Programming) of the Information Processing Society of Japan (IPSJ) during 1991-1995 and the secretary of the ACM Japan Chapter in June, 1993. He will serve as a program co-chair of ECOOP'97 (European Conference of Object-Oriented Programming).