Optimising MPI for Multicore Systems

Fall 2014

Instructor:

Dr. Ming Hwa Wang

Santa Clara University

Submitted by:

Akshaya Shenoy

Ramya Jagannath

Suraj Pulla

(Team 7 (Previously Team 8))

ACKNOWLEDGEMENT

This research paper is made possible through the help and support from everyone.

First and foremost, we would like to thank our Professor,
Dr.Ming Hwa Wang for giving us this opportunity and encouraging our research.

We would also like to thank the Santa Clara University for providing us good library and internet facilities.

Lastly, we would like to thank our family and friends who supported us.

Table of contents

List of figures

Abstract:

Multicore architectures are popular in both High performance Computing and commodity systems. MPI is used for communication between processes across distributed memory and are used for programming scientific applications .OSlevel process separations force MPI to perform unnecessary copying of messages within shared memory nodes. This project presents a approach that shares heap memory across MPI processes executing on the same node,allowing them to communicate like threaded applications. This approach enables shared memory communication and integrates with existing MPI libraries and applications without modifications.

1. Introduction

1.1 Objective

To improve the overall efficiency of MPI by sharing heap memory across MPI processes executing on the same node,allowing them to communicate like threaded applications

1.2 Why this project is related to the class

The project deals with inter process communication mechanism which is an important concept in Operating System. MPI is a standardised and portable message passing system used on variety of parallel computers. Optimising MPI helps in fast communication and improves the performance of processors.

1.3 Why other approach is not good

- OSlevel process separations without shared heap memory, force MPI to perform unnecessary copying of messages within shared memory nodes
- Thread based approach gives each MPI rank its own stack and heap within the shared address space. But one set of global variables is shared among all MPI ranks in a node. Application state becomes corrupted as different MPI ranks write to the common global variables.

1.4 Why our approach is better

Our approach shares memory among MPI ranks to implement communication between them more efficiently. This Optimised MPI resides on top of MPI interface and transparently improves the performance of existing MPI applications on modern and future multicore High Performance Computing systems.

1.5 Statement of the problem

MPI is the de facto programming model used for largescale computing in majority of scientific applications. However, it was originally designed for systems where singlecore compute nodes were connected by an internode network. The poor support of MPI implementations to manycore shared memory architectures has forced developers to use alternative programming models such as OpenMP or OpenCL and using MPI only for internode communication.

1.6 Area or Scope of investigation

This proposal focuses on the design of MPI libraries for manycore processors connected via shared memory hardware. Given the wide variety of applications that use MPI and systems on which they run, our goal is to ensure that peak shared memory communication performance is available to these applications. This experiment is performed on linux machine with manycore processors connected via shared memory hardware. We can enhance this further by considering different operating systems.

2. Theoretical background of the problem :

2.1 Definition of the problem

The performance and efficiency improvements in processor designs are achieved primarily by increasing the number of cores on a processing chip. MPI is used in communication between processes across distributed memory.The poor support of MPI implementations to manycore shared memory architectures has forced developers to use MPI only for internode communication.This proposal focuses to improve the overall efficiency of MPI by sharing heap memory across MPI processes executing on the same node,allowing them to communicate like threaded applications.

2.2 Theoretical background of the problem

Manycore architectures are becoming popular because of their high computing capacity. MPI's abstraction of explicit communication across distributed memory is very popular for programming scientific applications in supercomputers. Unfortunately, OSlevel process separations force MPI to perform unnecessary copying of messages within shared memory nodes. Thread based approach gives each MPI rank its own stack and heap within the shared address space. But one set of global variables is shared among all MPI ranks in a node. As a result the application state becomes corrupted as different MPI ranks write to the common global variables.The poor support that the MPI implementations provide for many-core shared memory architectures has forced developers to use alternative programming models such as OpenMP or OpenCL to parallelize computations on such hardware, using MPI only for internode communication.

Despite the limitations of today's MPI implementations, its programming model is actually highly compatible with the needs of future applications. MPI simplifies the communication by defaulting memory to be private to each computation thread. The challenge of MPI implementations is to provide developers the efficient abstraction across a wide range of architectures, including those where memory is actually shared or where only restricted communication primitives are available.

2.3 Related research to solve the problem

Different approaches were used for intranode communication using MPI

2.3.1 Processbased MPI

MPI was originally designed for systems where singlecore compute nodes were connected by an internode network. This implementation considered each MPI rank as an OS process with its own private memory. Figure 1 illustrates this layout.

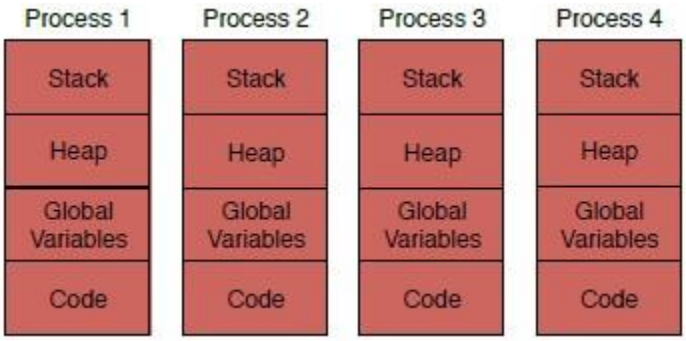


Figure 1: Memory layout in the traditional processbased MPI design.

Advantage of this approach

- Processbased design makes it easier to coordinate internode communication by multiple cores.
- Since each core is used by a separate process, their MPI libraries maintain separate state and thus require no synchronization

Disadvantage of this approach

- Unnecessary copying of messages among different ranks that are executing within the same shared memory node.
- MPI libraries often use a FIFO connection for small messages, and one or more shared memory regions mapped by multiple ranks for larger messages. Either case inherently requires two copy operations per message.(The sender copies from its private memory send buffer into the FIFO queue or shared memory region, and the receiver copies back out into its separate privatememory receive buffer)
- FIFOs are a pairwise connection, and shared memory regions may also be created on a pairwise basis to simplify communication.
- Thus, the number of resources grows as the square of the number of MPI ranks per node, which is often the number of cores per node. As a result will consume too many resources

2.3.2 Threadbased MPI

The limitations of processoriented MPI implementations has motivated research on implementations where MPI ranks are implemented as OS threads, all of which execute within the same process. Figure 2 illustrates this approach. Threads are an excellent choice since they share all their memory by default.

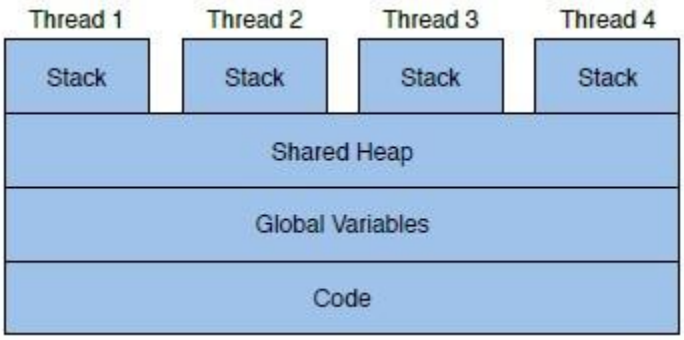


Figure 2: Memory layout in the threadbased MPI design.

Advantage of this approach

- Threads share their memory and hence solved the problem of unnecessary copying of messages among different ranks that are executing within the same shared memory node.

Disadvantage of this approach

- However, many MPI applications are written with the assumption that global variables are private to each MPI rank. While threading gives each rank its own stack memory, but heap within the shared address space and one set of global variables is shared among all MPI ranks in a node.
- Application state becomes corrupted as different MPI ranks write to the common global variables, which may exist within the application and in any libraries they link with.

- Developers of threadoriented MPI implementations have attempted to resolve this problem by privatizing global variables so that each thread is provided its own copy. But privatization is complex, especially in library code.

2.4 Our solution to solve this problem

The goal of our work is to develop an implementation of MPI that is
(i) optimized for shared memory hardware,
(ii)works on existing operating systems with no root access,
(iii) provides peak performance for intranode communication.
We have chosen OS processes as MPI ranks.The challenge of this approach is to share memory across processes by sharing the heap memory.

2.5 Our solution different from others and better than others

We differ from the others in the following ways

- Our approach focusses on design of MPI libraries for manycore processors by considering OS processes as MPI ranks
- It assigns each rank to its own process but shares heap memory among all ranks in a node
- Eliminates unnecessary copying of messages for intra node communication
- Our approach works on existing operating systems with no root access

Our approach combines the benefits of processbased and threadbased MPI design. Hence it is a better solution.

3. Hypothesis

Our goal is to improve the overall efficiency of MPI by sharing heap memory across MPI processes executing on the same node,allowing them to communicate like threaded applications. This experiment is performed on linux machine with manycore processors connected via shared memory hardware. Our proposed solution optimises MPI for intranode communication.

4. Methodology

4.1 Input data:

Compare performance of two programs(one with optimised MPI and other traditional MPI using two copy paths).

4.2 How to solve the problem

In the proposed solution we optimise MPI by sharing the heap memory across MPI ranks. We override the system's default memory allocator to allocate memory from a specially crafted shared memory pool. Normally, the memory allocator incrementally requests memory from the operating system using the sbrk or mmap system calls. We implement our own version of sbrk that requests memory from a shared memory region mapped on all MPI processes. We modify malloc library to transparently provide shared memory from malloc and related routines.To provide memory for a shared heap, we allocate and map a large shared memory region to the same address on each MPI rank using mmap.The shared region is divided evenly among the ranks on the node, and each rank allocates memory only from its part of the region. This approach eliminates the need for any synchronization between processes within the

memory allocator. Figure 4 illustrates how shared memory allocator connects multiple processes together.

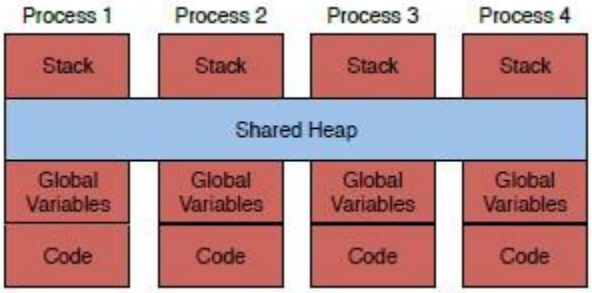


Figure 3: Memory layout of processes with shared heap allocator.

Stack, global variables, and code are private to each process, but the heap is shared. Our memory allocator provides the same sharedheap benefits as threadbased MPI and systems with kernel extensions. However, we incur none of the global variable privatization challenges encountered by thread-based MPIs and do not rely on specific operating systems, resulting in maximum portability.

We implement two incoming message queues per receiver. Figure 6 illustrates our design.

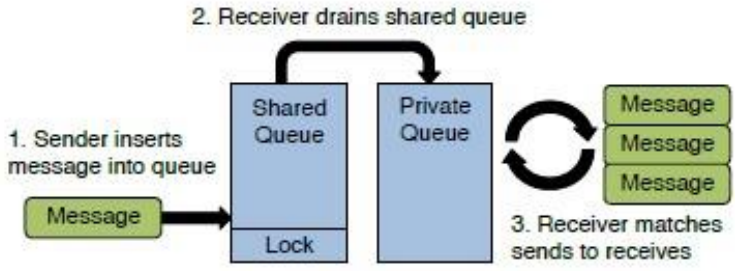


Figure 4: Optimised MPI’s message matching design.

One queue is globally accessible by all ranks. Senders add messages to the global queue owned by the rank for which the message is destined. Each global queue is protected by a lock.
The second queue is private. When a receiver attempts to match incoming sends to local receives, it drains its global queue and adds incoming sends to its private queue. The receiver then attempts to match sends on its private queue to local receives. A second private queue enables the receiver to loop many times without need for synchronization, and ensures that messages cannot be matched out of order due to senders adding new messages to the queue. Our dual queue technique minimizes contention between processes.

4.3 Tools used : GDB for debugging

4.4 Output:

Two programs producing same output. But one using optimized MPI performing better than other program not using optimized MPI.

5. Implementation

5.1 Code changes:

- Doug Lea's malloc is used (malloc.c and malloc.h)
- Wrapper is written to main, MPI_Send, MPI_Recv, malloc, calloc, realloc and free
- Two protocols are used, Immediate for small messages and synergistic for large messages

Wrapper to main function creates two shared regions using mmap, by calling openSharedRegion function call. This is a shared region accessible to all processes. Calls create_mspace_with_base() function of Dlmalloc.

```
int __wrap_main(int argc, char **argv) {
MPI_CHK(__real_MPI_Init(&argc, &argv));
MPI_CHK(MPI_Comm_size(MPI_COMM_WORLD, &numRanks));
MPI_CHK(MPI_Comm_rank(MPI_COMM_WORLD, &myRankId));

NULL_CHK(openSharedRegion("/msgQueueDescsregion",
MSG_QUEUE_DESCS_REGION_BASE_ADDR, sizeof(messageQueueDesc) * numRanks),1);
NULL_CHK(openSharedRegion("/sharedheapregion", SHARED_HEAP_BASE_ADDR,
HEAP_SIZE_PER_RANK * numRanks), -1);

msgQueueDescs = (messageQueueDesc *)MSG_QUEUE_DESCS_REGION_BASE_ADDR;
mySharedHeapPortionBaseAddr = SHARED_HEAP_BASE_ADDR + (myRankId *
HEAP_SIZE_PER_RANK);
mySharedHeapPortionMspace =
create_mspace_with_base(mySharedHeapPortionBaseAddr,HEAP_SIZE_PER_RANK,1);
NULL_CHK(mySharedHeapPortionMspace, -1);

MPI_CHK(MPI_Barrier(MPI_COMM_WORLD));

return __real_main(argc, argv);
}
```

Wrapper to malloc calls mspace_malloc() to allocate memory within the shared memory space.

```
void * __wrap_malloc(size_t size) {
void *ptr;

if (size == 0) {
ERR_PRINT("Invalid size specified for malloc");
return NULL;
}

ptr = mspace_malloc(mySharedHeapPortionMspace, size);
if (allocIsInMySharedHeapPortion(ptr, size) == false) {
ERR_PRINT("Malloc of size %ld failed", size);
mspace_free(mySharedHeapPortionMspace, ptr);
return NULL;
}

return (void *)ptr;
}
```

Wrapper to MPI_Send checks if the message is less than immediate threshold. If it is less uses immediate protocol.

```
if ((count <= MSG_SIZE_IMMEDIATE_THRESHOLD)) {
for (int i = 0; i < count; i++) {
msgHdr->immBuf[i] = ((const char *)buf)[i];
}
insertMsgIntoDestRanksSharedQueue(msgHdr, dest);
}
```

If the message is larger than synergistic threshold and if the buffer is in shared heap then uses synergistic protocol.

```
static void doSynergisticCopy(volatile char *dstBuf, char *srcBuf, int count, volatile int *blockIdxCounter,
volatile int *numBlocksDone) {

int numBlocks, blockSize, blockIdx, offset, size;
blockSize = SYNERGISTIC_COPY_BLOCK_SIZE;
numBlocks = (count + blockSize - 1) / blockSize;
while (true) {
blockIdx = __sync_fetch_and_add(blockIdxCounter, 1);
if (blockIdx >= numBlocks) {
break;
}
offset = blockIdx * blockSize;
size = blockSize;
if ((blockIdx == (numBlocks - 1)) && (count % blockSize)) {
size = count % blockSize;
}
memcpy((char *)dstBuf + offset, srcBuf + offset, size);
__sync_fetch_and_add(numBlocksDone, 1);
}
while (*numBlocksDone != numBlocks) {
sched_yield();
}
}
```

Wrapper to MPI_Recv transfers message from shared queue to private queue and then does message matching. After matching the message on private queue. It uses immediate protocol for copying small messages and synergistic protocol for large messages.

```
static messageHeader * findMatchingMsgInPrivateQueue(int count, MPI_Datatype datatype, int source, int
tag, MPI_Comm comm) {
messageHeader *cur, *prv;

prv = cur = msgQueueDescs[myRankId].privateQHead;
while (cur != NULL) {
if ((cur->count <= count) && (cur->datatype == datatype) && (cur->
source == source) && (cur->tag == tag) &&          (cur->comm == comm))
{
if (cur == msgQueueDescs[myRankId].privateQHead) {
msgQueueDescs[myRankId].privateQHead = cur->next;
if (cur == msgQueueDescs[myRankId].privateQTail) {
msgQueueDescs[myRankId].privateQTail = NULL;
}
}
else if (cur == msgQueueDescs[myRankId].privateQTail) {
msgQueueDescs[myRankId].privateQTail = prv;
}
prv->next = cur->next;
cur->next = NULL;
return cur;
}
prv = cur;
cur = cur->next;
}

return NULL;
}
```

5.2 Communication protocols and Flow charts

Although singlecopy message transfer was our goal with optimised MPI, simply using memcpy to transfer the data is often not the fastest method possible. We use an ‘immediate’ protocol for small messages less than the immediate threshold and a ‘synergistic’ protocol for messages larger than the synergistic threshold. Before queuing a message, the sender goes through a series of checks as shown in Figure5.

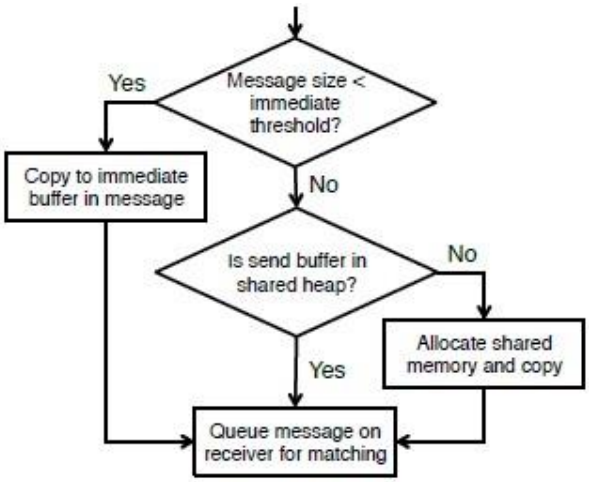


Figure 5 : Sender protocol flow. The sender ensures its buffer is in the shared heap and uses the immediate transfer protocol for small messages.

- If the message is small, we go into the immediate protocol, in-lining the message data with the message's matching information.
- If the application's send buffer is not located in the shared heap, we allocate a shared buffer and copy the data over.
- Finally, the message is queued on the receiver's shared queue.

Once a message is matched, the receiver decides how to transfer data from the send buffer to the receive buffer. Figure 6 shows the decision process

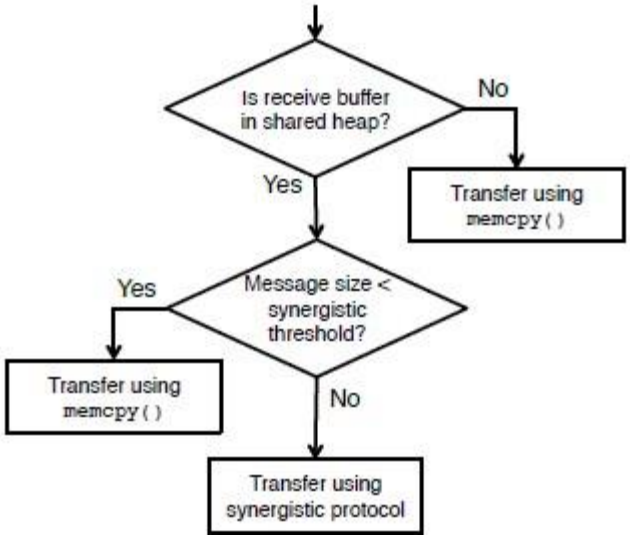


Figure 6: Receiver protocol flow. A single memcpy is used if the receive buffer is not in the shared heap or if the message is too small for the synergistic protocol.

- If the receive buffer is not on the shared heap, memcpy to transfer the data
- If the message is too small to use the synergistic protocol, we use memcpy to transfer the data.
- For larger messages we enter the synergistic protocol

5.2.1 Immediate Transfer protocol

When a message is matched, the receiver accesses the source message information (source rank, tag, communicator), incurring a cache miss. With a singlecopy data transfer approach, copying the message data will incur another cache miss, since that data has not been seen by the receiver. Inlining the message after the sender's matching information causes the hardware to bring the data into cache at the same time as the matching information, avoiding the second cache miss when copying the data.

5.2.2 Synergistic transfer protocol

For large messages,bandwidth is most important. We can achieve higher data transfer rates than possible with a single memcpy by having both the sender and receiver participate in copying data from the send buffer to the receive buffer. To do this, we break the data into blocks and utilize a shared counter that is atomically updated by the sender and receiver. When the receiver matches a message, it initializes the counter (used as a byte offset) and begins copying data one block at a time.

Before copying each block, the counter is incremented. If the sender enters the MPI library and sees that the receiver is copying in block mode, it also begins incrementing the counter and copying blocks of data until the entire message has been copied.

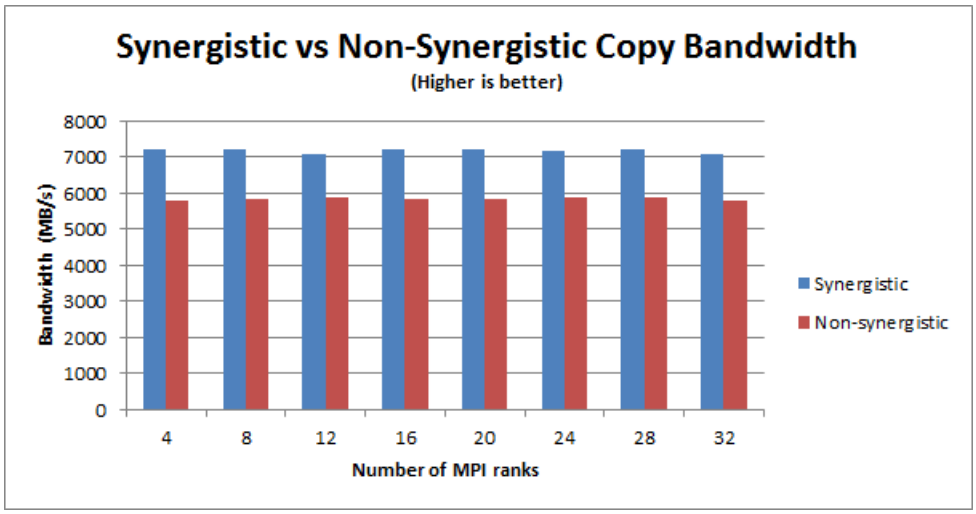
In the worst case, the sender does not participate (it is either executing application code or helping with other transfers), and we see the same bandwidth as a memcpy, which is the peak bandwidth achievable by one core. The sender can enter and assist the transfer at any point. Bandwidth improvement then depends on when the sender begins assisting and on the peak bandwidth achievable by two cores.

6. Data Analysis

6.1 Synergistic vs Non-Synergistic copy

x-axis represents number of MPI ranks and y-axis represents bandwidth(MB/sec).

The test is repeated for ranks varying from 4 to 32. The blue bars represents synergistic and red represents non-synergistic copy path.



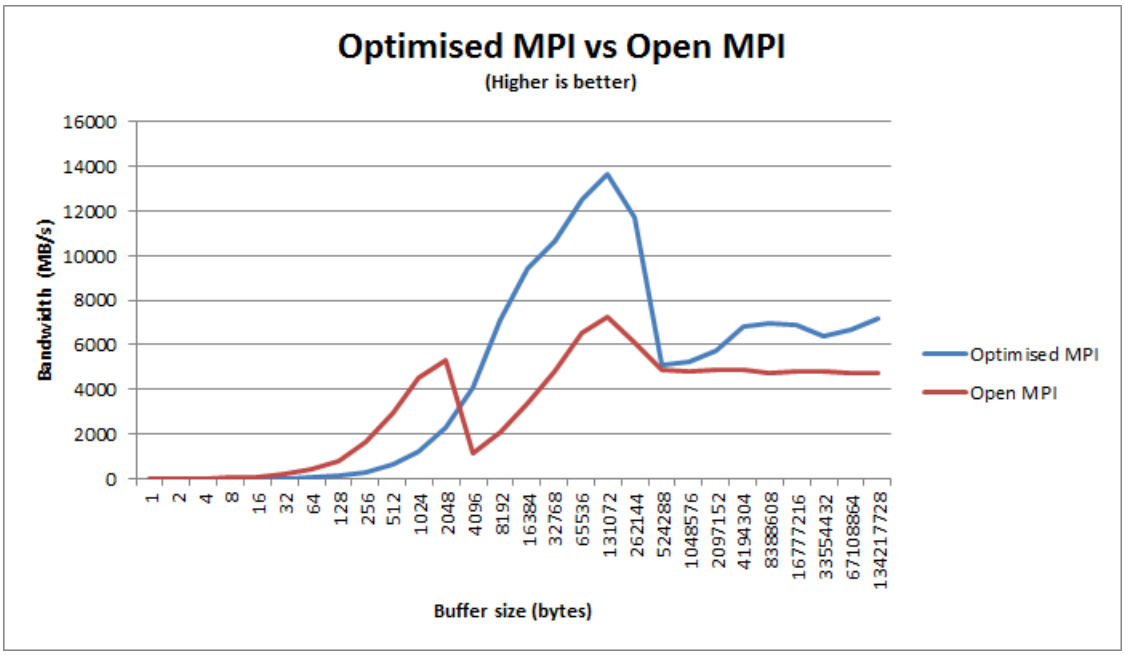
From the above graph it is clear that synergistic copy path provides better bandwidth that is faster communication.

6.2 Optimized MPI vs Open MPI

x-axis represents buffer size and y-axis represents bandwidth(MB/sec).

The test is repeated for ranks varying from 4 to 32. The blue lines represents optimized MPI and red represents Open MPI.

From the graph it is clear that optimized MPI performs better than Open MPI. The improvement in performance is significant in higher buffer sizes.



7. Conclusion

Our approach focuses a novel scheme for exploiting shared memory hardware in MPI programs. This mechanism shares selected subsets of memory among MPI ranks to implement communication between them more efficiently. This project presents a approach that shares heap memory across MPI processes executing on the same node, allowing them to communicate like threaded applications. This approach enables shared memory communication and integrates with existing MPI libraries and applications without modifications.

Future work includes developing optimized MPI on other platforms. This project uses LINUX operating system. More optimized solutions for faster communication can be explored.

8. Bibliography

1. http://dl.acm.org/citation.cfm?id=2503210.2503294&coll=DL&dl=GUIDE&CFID=453965786&CFTOKEN=16906363
2. http://dl.acm.org.libproxy.scu.edu/citation.cfm?id=2503210.2503294&coll=DL&dl=ACM&CFID=453965209&CFTOKEN=21797586
3. http://dl.acm.org/citation.cfm?id=2493123.2462903&coll=DL&dl=GUIDE&CFID=453965786&CFTOKEN=16906363
4. http://en.wikipedia.org/wiki/Message_Passing_Interface
5. https://computing.llnl.gov/tutorials/mpi/