Design Considerations for Shared Memory MPI Implementations on Linux NUMA Systems: An MPICH/MPICH2 Case Study.

-or-

There, but not back again: A story about the importance of being in the right place at the right time.

Per Ekman (pek@pdc.kth.se)
PDC/KTH, Stockholm, Sweden
http://www.pdc.kth.se/~pek/linux/

-with-

Philip Mucci (mucci@cs.utk.edu)
PDC/KTH, Stockholm, Sweden, ICL/UTK, Knoxville, Tennessee
Chris Parrott, Bill Brantley
Advanced Micro Devices, Austin, TX

Introduction/Abstract

This paper describes our work to make MPICH and MPICH2 more tolerant of Non Uniform Memory Access architectures (NUMA). MPICH and MPICH2 are freely-available implementations of the MPI and MPI-2 standards from Argonne National Laboratory. The goal was to increase the delivered performance of MPI applications by improving memory placement of the application and the MPI library. The target platform was the AMD Opteron microprocessor running SuSE Linux 9.1 Pro. Due to an on-chip memory controller, Opteron-based systems have NUMA properties in any multiprocessor configuration[1]. In the following pages, we discuss our modifications to the Linux kernel to do monitoring of page placement and our NUMA modifications to the MPICH and MPICH2 source code bases. We present some benchmarks demonstrating the performance impacts of these changes. We conclude with a discussion of the results and some suggestions regarding extensions of this work.

Some Notes on Terminology

The term "node" is used here as it is in the Linux kernel; a physical CPU that may possibly have some local memory attached. The term should not be confused with nodes in a cluster; those are referred to as "cluster nodes". A "system" refers to an entire NUMA machine, often a cluster node.

The MPICH devices use different terms to denote the data structure used to communicate data between processes. That type of data structure is uniformly referred to as a "packet" here. A "message" is the data buffer to send, and possibly, the data structure encapsulating it.

1 The notable exception being a single chip, multi-core configuration.

The term "local memory" does not dictate that the memory is non-shared, merely that it lives on the same node as the process referencing it. Non-shared memory is called "private memory". "Remote memory" is memory in another node and can be either shared or non-shared (private). The term "global" signifies that the data it refers to is shared by all processes.

Unless otherwise stated, "page-aligned" means that the start address of the memory area coincide with a page address AND that the size of the area is a whole number of pages so that the first byte after the end of the area lies in a new page.

Linux Kernel and NUMA

The three main characteristics of the Linux 2.6 kernel that are of interest to us are:

- Processes have weak processor affinity.

  The scheduler attempts to keep processes running on the same node. It will move processes between nodes to maintain adequate load balancing.

- Memory is physically allocated upon first touch.

  Memory pages owned by a process are allocated on the node where they are first referenced. By first reference, this means a write to anonymous pages (like from malloc()), or a read/write from/to for named pages (swapped or mmap'ed files). If the node on which the process is running has no free memory left, the memory will be allocated from the "topologically closest" node that has enough free memory. The determination of which node is closest is made at boot time.

  There are three cases to consider:

  1. Copy-on-Write (COW) inheritance.

     Anonymous virtual memory that has been inherited from the parent process is mapped to the same physical page frames used in the parent process. These frames are marked as Copy-on-Write. When either the parent or the child process writes to a page, it breaks the COW and a new page frame will be allocated for that process. This means that, if the parent process breaks COW for a page, both copies of that page will end up living on the node that the parent process runs on.

  2. Initial allocations map to the "zero" page.

     Anonymous virtual memory, that has been newly allocated from the kernel, is initially mapped to the Copy-on-Write "zero"-page. A write to a virtual address then breaks the COW and a physical page frame will be allocated for that address on the node on which the touching process is running.

  3. Named pages (pages that are backed by a file or swap device).

Named pages are allocated on the node where the process doing the reference (read or write) is running.

- Pages are not migrated.

  Pages that have been touched will stay on the same node that they were allocated on until they are deallocated or paged out to secondary storage.

The behavior of the Linux kernel can be summarized quite simply: Processes have soft affinity and memory pages have hard affinity. There are two factors that determine whether memory ends up on the "right" node or not :

- Is the first write to each page issued on the "correct" node (the node that will compute on them later)?

- Are the data structures inside shared memory regions properly page aligned?

Monitoring Memory Placement in the Linux Kernel

In order to adequately diagnose poor memory placement, it is necessary to figure out which node a page of virtual memory lives on. In order to do that, the virtual address must be translated to the corresponding physical address.[2] Once this mapping is obtained, the physical address of the page can be compared to the physical memory map to figure out which node the page lives on. However, doing this mapping for all active virtual pages on the system results in information overload. A better way is to obtain this mapping on a per-process basis and aggregate the data into per-node groups of contiguous page ranges. This allows clear discontinuities to be spotted. As part of this work, we have implemented the above functionality in two separate patches to Linux 2.6.8. The patches provide information about the memory placement via special files in the /proc filesystem.

- /proc/<PID>/pagemap

  This file contains a listing of all pages used by the process. Each page is marked with the corresponding node number and copy-on-write status, meaning that it may change node later if one of the processes sharing the page breaks the copy-on-write status.

```
# cat /proc/self/pagemap
VMA 0 start 400000 end 404000
00400000-00400fff node 1
00401000-00401fff node 1
...
```

- /proc/<PID>/nodemem

  This file contains aggregated virtual memory ranges that reside on the same node.

2 Unfortunately, we were unable to find a documented interface in our Linux installation to perform this mapping. Andi Kleen's **libnuma** apparently has an undocumented set of flags to get_mempolicy() where it can perform virtual address to physical node mappings.

```
# cat /proc/self/nodemem
0000000000504000-0000000000525fff node 0
...
0000002a9576a000-0000002a9576afff node 3
Process running on node 0
Process has 2392064 bytes (584 pages) on node 0
Process has 122880 bytes (30 pages) on node 1
Process has 122880 bytes (30 pages) on node 2
Process has 90112 bytes (22 pages) on node 3
```

To further help our analysis, a perl script was developed that parsed the output of the /proc/<PID>/nodemem file, as well as the listing from /proc/<PID>/maps. This allowed precise attribution of pages to different regions of the executable.

```
 # ./maps.pl 16518
Process running on node 1
Process has 12341248 bytes (3013 pages) on node 0
Process has 397312 bytes (97 pages) on node 1
Process has 180224 bytes (44 pages) on node 2
Process has 172032 bytes (42 pages) on node 3
Process has 0 bytes (0 pages) on node 4
Process has 0 bytes (0 pages) on node 5
Process has 0 bytes (0 pages) on node 6
Process has 0 bytes (0 pages) on node 7
Process has 0 bytes (0 pages) not in core
VMA 2a95556000-2a9556a000 /lib64/ld-2.3.3.so
        5 pages on node 1
        5 pages on node 3
        39 pages on node 0
        4 pages on node 2
VMA 2a95b12000-2a95c18000 /lib64/tls/libc.so.6
        33 pages on node 1
        31 pages on node 3
        447 pages on node 0
        35 pages on node 2
VMA 400000-472000 /bin/bash
        114 pages on node 0
VMA 2a955a4000-2a955c5000
        32 pages on node 0
...
Processed 3196 pages
```

The MPICH 1.2.6 shared memory implementations.

Here we examine the two MPICH devices that are most likely to be used on a NUMA-system running Linux; the pure shared memory device, ch_shmem, and the combined shared memory/socket device, ch_p4.

MPICH ch_shmem Internals

With the ch_shmem device, all processes communicate through a structure in shared memory called MPID_shmem. This structure contains an array of packet queues, an array of stacks of free packets, various locks and packet “pool”, where the data actually resides. There is one packet queue and one stack of free packets for each process. Packets contain a a "next" pointer that is used to construct the linked lists that implement the queues and stacks. To alleviate pressure on the MPID_shmem structure, each process has a private structure called MPID_lshmem. This contains copies of pointers to the elements in MPID_shmem.

Initialization

When the application starts, a shared memory region is set up for use by the MPICH's internal memory allocator. This is accomplished by creating an anonymous, shared memory region with the mmap() system call.[3] Memory for MPID_shmem is then allocated and the packet pool is partitioned. Each packet is pushed onto one of the per-process stacks of available packets. Finally, the child processes are spawned using the fork() system call.

Sending

Sending is accomplished by first popping a free packet from the process' stack of available packets. If a packet isn't available, the process blocks. Then the data from the user buffer is copied into the packet using memcpy(). Finally, the packet is inserted into the incoming packet-queue of the destination process.

Receiving

When receiving a packet, the data is copied from the packet to the user buffer and the packet is removed from the incoming-queue. It is then placed in private cache of packets to be returned to the sender's stack of available packets. The cache is flushed after a certain number of packets has been put on it.

Problems with ch_shmem

Even though the ch_shmem device keeps separate queues for each process, the location of the packets in memory is still a performance problem in Linux/NUMA systems. With the default memory affinity policy of Linux, the first process to write to a page, owns that page for the duration of execution. In the initialization of ch_shmem, all packets in the pool are touched when their "next" pointers are set and the packets are partitioned. This causes all the memory pages in the pool, and therefore all packets, to live permanently on the node on which this setup code was executed. Consequentially, all processes in the application are forced to access memory on the

3 Early versions of Linux did not support the MAP_ANON flag to mmap().

startup node every time a message is sent or received. This can be a performance limiter, with respect to both bandwidth and latency. Furthermore, with the default soft processor affinity, processes may move between nodes but the memory will not, compounding the problem and giving non-deterministic performance.

Suggested Modifications to ch_shmem

In order to get predictable performance and make sure that processes don't migrate away from their data, processes needed be locked to nodes so that they display the same hard affinity as memory. In newer Linux kernels, this can be accomplished with the sched_setaffinity() system call. It is important that the affinity be set as early as possible in the startup process, before any private memory has been touched. If a process is migrated from one node to another before the processor affinity is set, all memory it touched while running on the initial node will be in the wrong place. In order to make sure that the processes are evenly distributed among the available nodes, the affinity cannot be set until the process has discovered its own rank (MPID_myid). Only after that point, can the processes be scheduled to their own rank modulo the number of processors available. For the child processes, this can then be done in p2p_create_procs() (ch_shmem/p2pprocs.c) right after MPID_myid is set from the sequence number MPID_shmem->globid. For the initial process, affinity can be set to node 0 in MPID_SH_InitMsgPass() (ch_shmem/shmeminit.c) before the shared memory segment is initialized. To force a rescheduling of the process, a sched_yield() call should be made directly following the call to sched_setaffinity(). Otherwise, the process could continue to execute on the current (but wrong) node until the current scheduler time-slice runs out.

Any time a message is sent, data is copied from a user buffer on the sender (A) to a packet in shared memory, and then from shared memory into a user buffer at the receiver (B). When the packet resides in either A's (or B's) local memory, the latency will be minimal[1] thus we need to make sure that the shared memory area used by process A's packets gets allocated to the node on which process A is running. Furthermore, by distributing the packets such that the sender's packets live on the sender's node, we improve total bandwidth, as more memory interfaces are be utilized for communication. Given a regular communication pattern, the exchange of data will not stress one node's memory system more than another.

Most of the code needed to get processes to use packets in local memory is already present in MPICH. Any given process will always use the same set of packets for sending data and those sets are partitioned among processes. Since memory gets assigned to nodes at the granularity of pages, we need to make sure that:

1. The packet pool is page aligned.

2. The packet pool partitions occupy a whole number of pages in memory.

3. The pages used in any pool partitions are first written to by the process that should own them, **after** that process has been started on the node where it is supposed to live.

Given 1 and 2, 3 will cause the Linux kernel to place the memory pages used by the packets in physical memory on the correct node. To accomplish the above, MPID_shmem needs to be allocated with a memory allocator that returns memory that is page-aligned. Also, the MPID_shmem structure must contain internal padding such that the packet pool begins on a page boundary. Furthermore, the size of the pool divided by the number of processors must be a whole multiple of the page size. This allows each partition to contain a whole number of pages worth of packets in the pool. Finally, each child process must initialize it's own stack of available packets from the pool. The case where the last packet in a partition spans a page boundary between partitions can be solved by only initializing the number of packets that fit completely within the processors partition.

MPICH p4_shmem

With the p4 device, processes communicate via shared memory when on the same machine or via sockets between systems. Shared memory communication is implemented through a global data structure called p4_global. p4_global contains:

- One queue of incoming packets for each process.
- A queue of cached available packets, initially empty.
- An array of cached available message buffers, initially empty.

Each process also has a private structure called p4_local which contains a private message queue for the process.

Initialization

When the application starts, a shared memory segment is set up for the internal memory allocation interface using the System V IPC interface (shmget/shmat). In Linux this is implemented in the same way as shared anonymous memory mappings; by mapping a file in a RAM-filesystem using the mmap() system call. Memory for p4_global is allocated from the segment and some variables are initialized. Then, an array is set up as a cache of available message buffers with a power-of-two sizes up to 1MB. Memory for p4_local is allocated using malloc() and is initialized. After this, the child local processes are spawned by the library using fork(). The child processes free p4_local, reallocate it with malloc() and reinitialize it.

Sending

On sending via shared memory, a buffer for the message is fetched from the available message buffer cache in p4_global, or allocated from the shared memory segment if no free buffer of sufficient size can be found. The data is copied from the user buffer into the message buffer and a packet is fetched from the global queue of available packets. If the available packet queue is empty, a new packet is allocated from the shared memory segment. The message is linked into the packet and the packet is put at the end of the incoming packet queue of the destination process.

Receiving

When receiving a packet, the private queue of received messages is first searched for a matching message (one with a sender and type that matches those of the MPI receive call that triggered the receive process). If a match is found, the data is copied into the user buffer and the message is either cached in the global queue of available message buffers (if the size of the message matches one of the saved sizes), or it is freed. If no matching message is found, a packet is removed from the head of the global incoming packet queue for this process if one is available. The message is retrieved from the packet and placed on the private, unexpected message queue. The packet is then placed at the head of the queue of available packets.

Problems with p4_shmem

The situation with p4_shmem is more complicated than with ch_shmem. When a message is being prepared, it could end up using a packet cached in the global queue of available packets. The physical memory used for such a packet will reside on the node that originally allocated it, which may be any node in the system. When no previously allocated packet is available, the process allocates a new packet from shared memory, but where in physical memory that packet ends up depends on the page-alignment of the area allocated. If the area is part of a page that has already been partially used for other packets it could already have been allocated to physical memory on a remote node. This makes the performance of any given message transaction hard or impossible to predict. The same problems exist for message buffers: The queue of free messages is shared among the processors and messages can be allocated on demand. p4_shmem also has the same problems with soft processor affinity as ch_shmem, processes may move between nodes as the OS scheduler responds to temporary load imbalances.

Changes to p4_shmem

First and foremost, p4 processes needs to be given hard processor affinity in the same way as with ch_shmem.

Second, the queue of available packets should be split up so that each process has its own queue. When a process receives a packet it needs to return the packet to the queue of available packets belonging to the sending process. In this way any given node will only send packets allocated on physical memory local to it. For this to work the individual partitions of the queue needs to be page aligned.

Third, packet allocation must happen either in page-aligned chunks or from a page-aligned per-process shared memory segment. This is also true of message buffers. A new shared memory allocator must be implemented for both types of allocations; one that allocates page-aligned, page-sized blocks of memory, or takes a process ID as an argument and keeps per-process “arenas”. If the memory allocated keeps per-process arenas, then the free-routine must also keep track of where to return the memory being freed.

Lastly, the list of available message buffers needs to be split up into page-aligned, per-processor partitions and the process of returning message buffers after use must be modified to return the buffers to the sending process' list.

The MPICH2 Shared Memory Implementation

MPICH2 uses the Abstract Device Interface (ADI) version 3 to implement communication. The current MPICH2 devices all use the CH3 ("channel") implementation design of ADI3. CH3 supports asynchronous, non-blocking communications using MPI requests to keep track of incomplete operations. MPICH2 has two shared memory channels, the shm channel and the "scalable shm" channel sshm. We look at the sshm channel here.

Process startup in MPICH2 occurs external to the library. Unlike MPICH the MPI library code never forks, even in the shared memory case. In the MPD, processes are started by forking the mpd process already running on the host and then calling exec() on the binary to be started.

Communication occurs through a "virtual connection", or vc, a structure describing a point-to-point connection between two processes. A virtual connection is represented by a vc-structure private to each process. vc contains, among other things:

- A pointer to a shared memory write queue.
- A pointer to a shared memory read queue.
- A send request queue.
- A active send request.
- A active receive request.
- A current request .

Asynchronous communication is implemented by storing the pointer to the user buffer submitted in a MPI call in a MPID_Request structure that is queued in the virtual connection or in the MPIDI_Process struct (depending on the type and status of the request). A "progress engine" updates pending requests and sets a completion flag so that completed requests can be returned to the application. The progress engine can operate synchronously in a single threaded mode or asynchronously as a separate thread of execution. The current MPICH2 implementation of the ADI (CH3) does not yet seem to fully support a multi-threaded implementation.

On a Linux system, sshm allocates memory for all shared memory allocation requests directly using mmap() with the MAP_SHARED flag. No shared memory arena is preallocated for small allocations. Since mmap() creates a new VMA (which is always page-aligned) in Linux the area it allocates will be properly page-aligned.

SSHM Initialization

The virtual connections (vcs) for a process are allocated in an array in the MPI process group. The vcs are not connected until they are actually used. Furthermore, a vc can be connected in one direction. A process ever only connects a vc for sending. A vc in process A, representing the connection between processes A and B, is connected for reception when process B connects its own vc representing the connection for sending. When connecting a vc the write queue for the vc is allocated from shared memory. The vc read queue is initialized when the process at the other end connects. It is initialized to point to the shared memory area allocated for the write queue at the other end of the connection. When vc is a connected virtual connection between process A and B then the read queue in process A is a pointer to the same shared memory area as the write queue in process B, and vice versa. The queues contains a fixed array of packets that are used for communication on the channel. The MPIDI_Process struct maintains two lists of virtual connections, shm_reading_list for vcs that have been connected for reading and shm_writing_list for vcs that have been connected for writing. The same vc can be on both lists at the same time. Only the vcs on those lists will be checked for pending transmissions by the progress engine.

A "bootstrap queue" is used to transfer the shared memory identifier that is needed to attach to already allocated shared memory areas (such as when a process attaches the pointer to the read queue to the memory area allocated to the write queue of another process). The bootstrap queue is a local linked list of message nodes living in shared memory. The shared area is memory mapped and thus page-aligned. The linked list itself lives in private memory.

Modifications to SSHM

The only modification that the MPICH2 SSHM device needs is for the mpd to set hard CPU affinity for the forked child process prior to exec'ing the executable. One way to do this is to start the executable under the numactl-program to set the CPU affinity to the rank of the process modulo the number of available CPUs on the system. This is done by modifying the clientPgm and clientPgmArgs in src/pm/mpd/mpdman.py:mpdman() in the source tree, or by modifying the installed mpdman.py in an existing MPICH2 installation directory (bin/mpdman.py).

Benchmarks

We implemented the above changes and evaluated the performance of MPI using MPI_BENCH, an MPI Benchmark found in the LLCBench benchmark suite. The execution platform was a 4 processor AMD Opteron system, model 848 processors at 2.2GHz. The system was fully populated with 16 1GB DDR 400 memory modules. The operating system was Suse Linux 9, Service Pack 1. Both MPI_Bench and MPICH were compiled using the standard options with gcc 3.3.3.

mpi_bench was run with the following options 40 times for each data point. The arithmetic mean and the standard deviation was then computed. The test was run on 4 processors with the first processor and the last processor communicating, the inner two were idle.

-d	bidirectional send/recv mode (mpi_irecv()/mpi_isend()/mpi_waitall()).
-i1000	1000 iterations per message size
-e1	do only 1 run of 1000 iterations
-f	flush cache before start and message sizes
-F	flush cache before starting and between repeat cases
-M<size>	use this message size

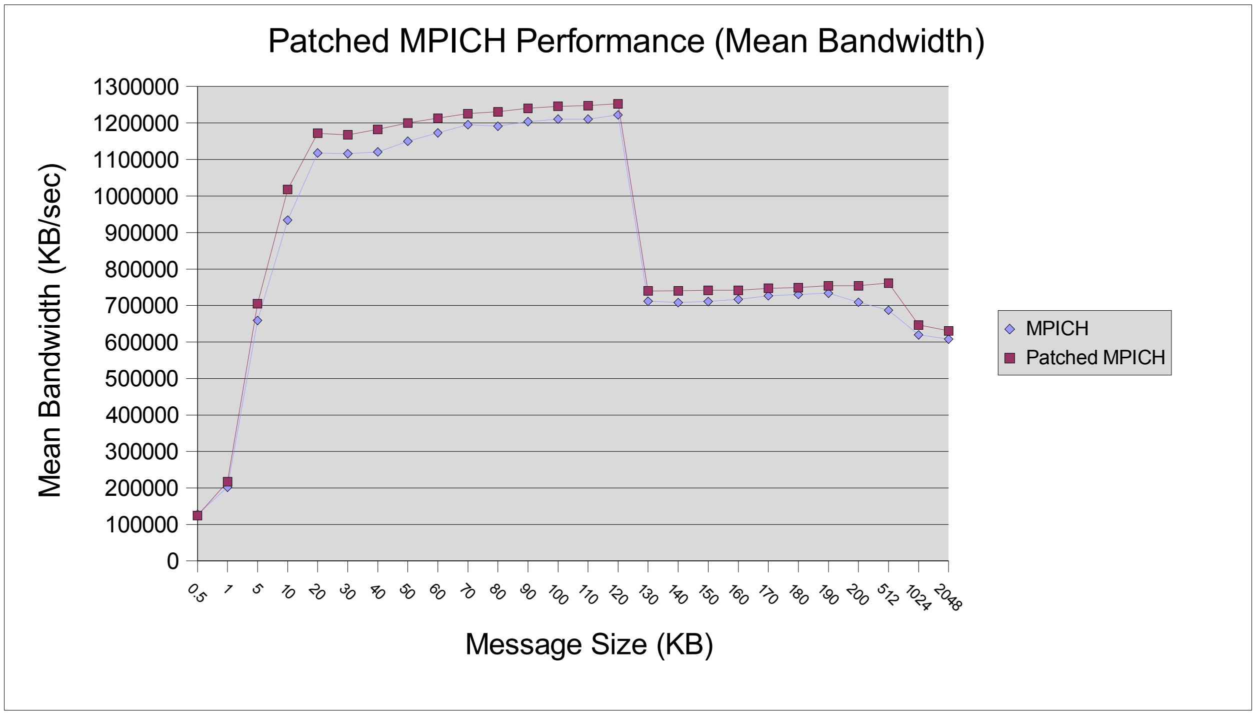

Figure 1: Mean Bandwidth of Patched MPICH 1.2.6

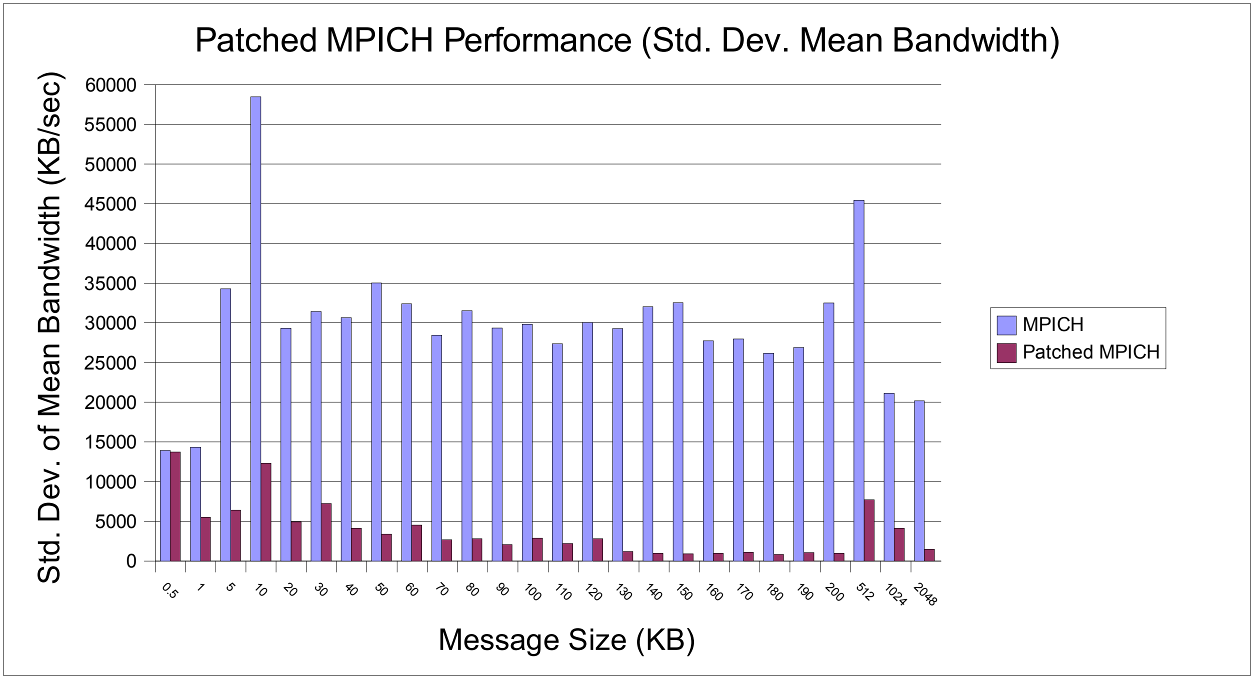

Figure 2: Standard Deviation of Mean Bandwidth of Patched MPICH 1.2.6

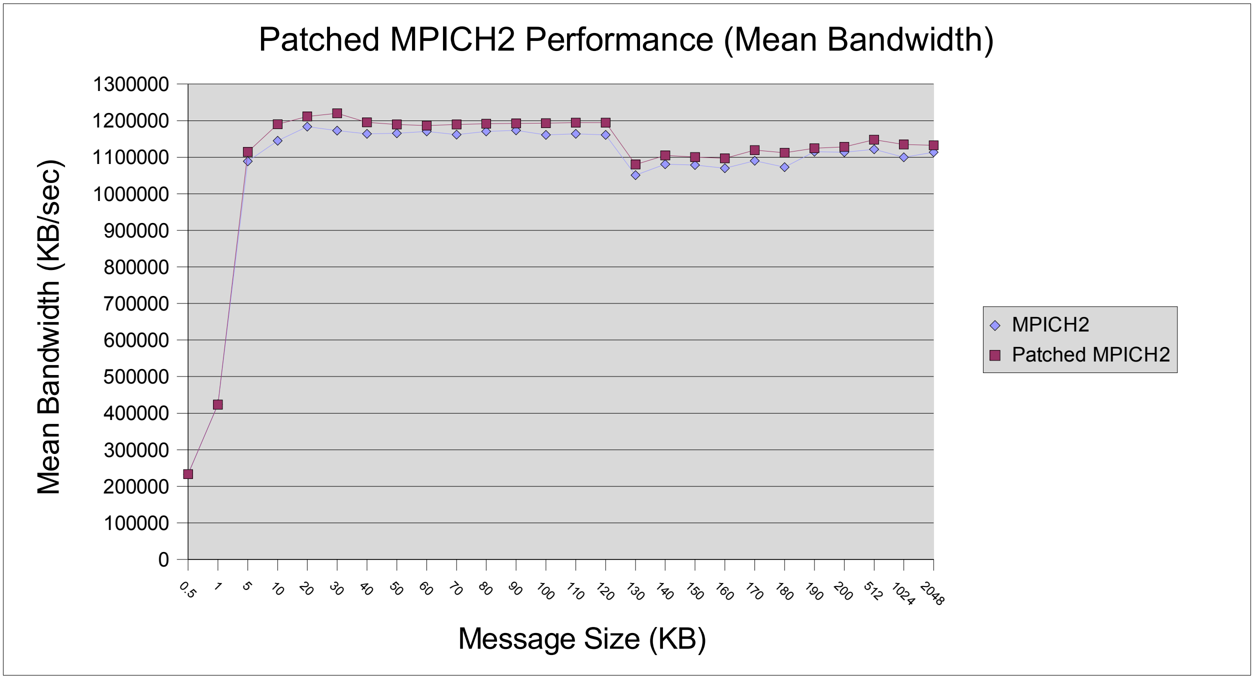

Figure 3: Mean Bandwidth of Patched MPICH2

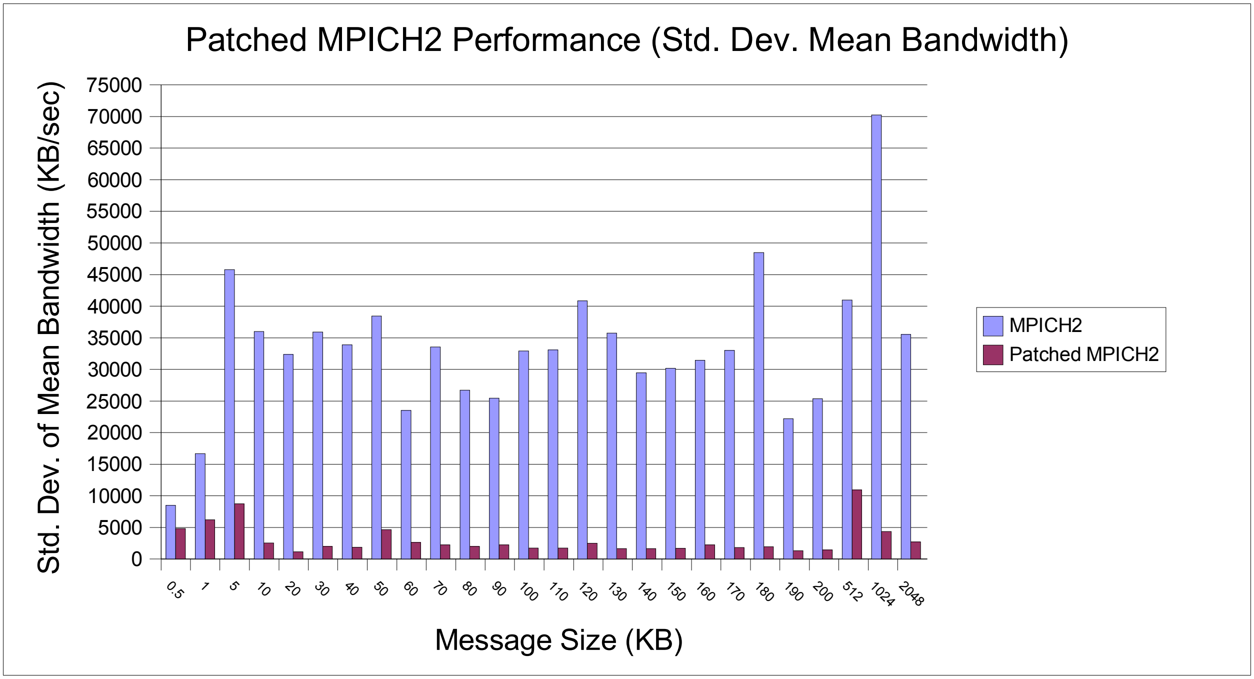

Figure 4: Standard Deviation of Mean Bandwidth of Patched MPICH2

Discussion

We see that in both Fig. 1 and Fig. 3, that the average bandwidth has improved a 2-10% percent in most cases. This is to be expected since the cost of interprocessor communication is current dominated by the cost of the two copy implementation (one to the shared memory, one to receiver). The effects are more pronounced in the MPICH1 implementation because the packets now almost always live in the “right” place, on the sender's node. Because of the 4 processor run, usually we are communicating between nodes that are farthest apart in the HyperTransport topology. As HyperTransport only exhibits a small percent decrease in bandwidth with each additional hop, we are able to “make up” this difference by proper memory placement of the packets and data structures in the shared memory region. The humps at the beginning part of each curve are from the cache effects on repeated use of the same message buffers in the benchmark. The big benefit of these modifications becomes apparent when one observes the differences from run to run of the same benchmark. The patched versions show a tremendous decrease in variation from run to run, even for messages that live entirely in cache. The variation stems both from the original choice of process to node mapping and from subsequent rescheduling. The patch guarantee's the same placement of processes for every run, delivering highly predictable performance.

It should be noted here that no runs were done of actual application code. However, we believe that the effects of this work will have an even greater impact on application performance. This is due to the soft processor affinity of Linux processes. Once the user has issued an MPI_Init (), his process will be on the same node for the duration of the run. Depending on the initial process placement and the type/frequency of memory accesses, it's possible to see a significant increase in the performance of application code. A test user of the patched MPICH 1.2.6 reported that a 4 processor run of EGDE, a computational electromagnetics code that performs sparse matrix-vector arithmetic, ran 30% faster.

Future Work

AMD is currently implementing our suggested changes to the MPICH1 ch_p4 device. The most difficult modifications to perform are the per-page and per-process allocation of the shared memory region. My using a custom memory allocator and proper padding of the global structure, virtually all remote references due to MPI references can be eliminated.

It should be noted that while we have drastically reduced the variation in run-time and increased average performance in most cases, there still is a lot to be gained by eliminating the copies done by both these MPI implementations. It should be noted that for unidirectional transfers, it's faster to send a message to yourself over an Infiniband/PCI-X adapter than it is over shared memory due to Infiniband's zero copy infrastructure.

Linux provides a mechanism to do single process to process memory copies via the /proc/<PID>/mem device. MPICH2 appears to have remnants of code that attempted to use this at one time, but it is commented out in the 1.0.1 release. A single copy could easily be built using the current infrastructure that uses the shared memory segments to exchange pointers and offsets. The actual data transfer could be done using the above device. Furthermore, the current implementation uses System V semaphores (a system-call) to guard against concurrent accesses to shared data structures. Semaphores are known to be highly inefficient and numerous high performance, user-level equivalents exist. Even better, the implementations could be made lock-free by using atomic compare-and-swap instructions available on most platforms.

For a true high performance implementation, a kernel level zero-copy mechanism should be designed. This would allow true zero copy between cooperating processes by intelligent use of the TLB, page-tables and the Copy-On-Write mechanism. There have been other experiments in this area, most notably the zero-copy pipe() patch that was circulated on the Linux Kernel mailing list. Also, correspondence with the developers of MPICH-VMA from NCSA, indicated that they had also experimented with a kernel module to do true zero copy. With the advent of high performance zero-copy network stacks, the time has come to provide this capability for interprocessor communication.

End Notes

1. This argument applies to data movement through the processor, this appears to be the way memcpy() does things on linux.

Let us denote the time taken to move a data object from memory into the processor over the memory bus locally on a node T, and the time taken to transfer the object over the node-interconnect from node A to node B $T_{A,B}$. We assume that the local memory transfer time is equal on all nodes and furthermore that writing the same object from the processor to memory is the same as for reading it.

In the case where all packets live on node 0, the transfer time for a message between nodes X and Y, when X and Y are not equal to zero is:

$$T+T_{X,0} + T + T + T_{0,Y} + T = 4T+ T_{X,0} + T_{0,Y}$$

If the packets live on node X the time becomes:

$$T+T+T+T_{X,Y}+T = 4T + T_{X,Y}$$

Assuming that $T_{X,Y}$ is always shorter or equal to $(T_{X,0} + T_{0,Y})$, which is a reasonable assumption since the path from X to Y could pass node 0, the latency in the case where the packets live on one of the involved nodes will be smaller or at worst equal to the case where all packets live on the same node.

References/Links

Kernel Patches, MPICH Patches and the most recent version of this document.
http://www.pdc.kth.se/~pek/linux

Undocumented Virtual Address to Node Mapping in Andi Kleen's libnuma.
http://article.gmane.org/gmane.linux.lse/3366

Understanding the Linux Virtual Memory Manager by Mel Gorman
http://www.skynet.ie/~mel/projects/vm/

Linux Device Drivers, 2nd Edition by Alessandro Rubini and Jonathan Corbet
http://www.xml.com/ldd/chapter/book/index.html

MPICH2 Design Document, Draft of October 14, 2002 by David Ashton, William Gropp, Ewing Lusk, Rob Ross, Brian Toonen
http://www-unix.mcs.anl.gov/mpi/mpich2/docs/mpich2.pdf

A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard, by William Gropp, Ewing Lusk, Anthony Skjellum
http://www-unix.mcs.anl.gov/mpi/mpich/papers/mpicharticle/paper.html

MPICH Abstract Device Interface Version 3.4 Reference Manual by William Gropp, Ewing Lusk, David Ashton, Rob Ross, Rajeev Thakur, Brian Toonen
http://www-unix.mcs.anl.gov/mpi/mpich2/docs/adi3man.pdf

MPICH Home Page
http://www.mcs.anl.gov/mpi/mpich

MPICH2 Home Page
http://www.mcs.anl.gov/mpi/mpich2

Zero Copy Pipes
http://lse.sourceforge.net/pipe/pipe-results-large-zero

Efficient MPI on SMP Cluster
http://ipdps.eece.unm.edu/1999/pc-now/takahash.pdf

MPIBench and LLCBench
http://icl.cs.utk.edu/llcbench