Improving Parallel Performance Using MPI One-Sided Communication and MPI-IO

Jon Gibson
HPC Consultant, Manchester Computing, University of Manchester

Two major bottlenecks for codes running on HPC machines are communication time and time to do I/O. Both of these are becoming more significant with time, given that processor speed is increasing relative to communication speed, and as the trend is to run jobs on larger and larger numbers of processors. In this article, we consider how the MPI 2 standard provides techniques for improving performance in these two areas.

In the communication model assumed by the MPI-1 standard, each processor has its own local memory, whose contents can only be modified by the process running on that particular processor. Hence, in order to change the memory associated with a remote processor, there must be explicit send and receive calls by the processes involved. This requirement for both a sender and receiver is referred to as two-sided communication. Although this approach works very well for many problems, it is not always ideal for a given algorithm. For example, it requires both the sender and receiver to know how many messages are being sent, as well as the type and amount of data. It may also need an excessive amount of synchronization, especially where a large number of small messages are involved.

For codes with a communication bottleneck, alternative approaches need to be considered. If the code is using blocking communication, then a move to non-blocking calls allows greater overlapping of communication and computation, so reducing the synchronization overhead. However, further improvements can be made with a move to one-sided communication. As the name suggests, one-sided communication allows a given process to directly access the memory of another for the purposes of reading and writing. This leads to both a reduction in the synchronisation overhead and possible algorithmic simplification, in that only the process making the remote memory access needs to know the type and amount of data. Although there are a number of one-sided communication models (e.g. SHMEM, LAPI, Co-array Fortran), for portable code, as with message-passing generally, the MPI paradigm is the one to choose.

Programming one-sided MPI communication does require a new way of thinking, with the programmer having to decide what bits of memory should allow remote access and at what times. There are also new types of bug to be avoided. Since we don't have the space to go into the details of using one-sided communication here, we'll illustrate its potential with a simple example. CSAR's Kevin Roy has re-implemented the MPI_Allgather function using one-sided communication and compared its performance with the standard two-sided implementation. Figure 1 shows the significant improvement in performance this provides. A further improvement in performance is expected with improved synchronisation within this algorithm.

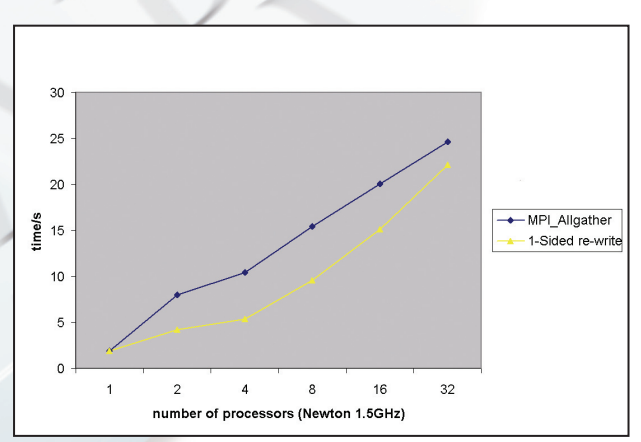

Figure 1: Improving the Performance of MPI_Allgather using MPI One-Sided Communication.

As we've already mentioned, I/O can also be a major bottleneck on parallel machines. In fact it is often said that "a supercomputer is a device for converting a compute-bound problem into an I/O bound problem." Any code that uses a single process to perform I/O on behalf of all processes is serialising that part of the application and hence limiting the overall scalability according to Amdahl's law. Alternatively, if the I/O can be performed in parallel on a parallel file system, then the performance and scalability of the code can be greatly increased. A simple approach to parallelising I/O could be to get each process to output one file. However, this restricts the ability to change the number of processes and the separate output files may still have to be combined in some way in order to post-process or analyse the data. Parallel access to individual files is

really what we want. There are a number of I/O libraries that can provide this, two popular ones being NetCDF and HDF5. Both of these have parallel versions of the serial libraries and allow architecture-neutral files to be created. However, the parallel versions of these libraries are still at the developmental stage. They are built on top of MPI-IO and it is necessary to understand aspects of MPI-IO in order to use the libraries effectively.

MPI-IO allows the input and output of binary files using all the processes within an MPI group and offers the advantage of output to a single file. File access is performed using MPI derived datatypes, allowing fast I/O using collective operations. Data is packed into a file in a manner consistent with a serial program and so data can be read out on any number of processors and hence files are re-usable on a given machine. There are in fact three different format options for writing data: native, internal and external32. The first of these is the native format of the machine; internal is understood by the whole MPI environment, even if it happens to be heterogeneous; and external32 is a completely portable, machine-independent format. Unfortunately, external32 is not currently available on Altix machines but once it is, it will make MPI-IO an even more attractive option.

Again, the details of using MPI-IO are beyond the scope of this article. However, if your appetite has been whetted, then we'd like to point you in the direction of our "MPI One-Sided Communication and MPI-IO" course. In this one day course, we delve into the mysteries of these important features of the MPI-2 standard and explain how they can be used to improve the performance and scaling of your code. A course is likely to be scheduled in the near future. Please contact jon.gibson@manchester.ac.uk to register your interest now, as places are likely to be limited.

Technical Symposium on Reconfigurable Computing with FPGAs, 21-22 February 2005

Kevin Roy and Mike Pettipher
Research Support Services, Manchester Computing, University of Manchester

In February 2005, the University of Manchester hosted a 2 day symposium on Reconfigurable Computing with FPGAs (Field Programmable Gate Arrays). This meeting was sponsored by Cray and SGI, and supported by the Ohio Supercomputer Center (OSC), who hosted a similar meeting in October 2004. The focus of the symposium was the use of FPGAs for High Performance Computing.

For people that have not come across FPGAs before, they are essentially hardware that can be programmed to do whatever they are tasked with. The millions of logic gates on the chip allow a flow of data or bits; flows can be constructed into algorithms to solve complex problems. The real benefit is that, because of their reconfigurability, an algorithm written for an FPGA allows you to create a processor to solve your particular problem rather than using the main CPU which has a rather rigid structure (set numbers of floating point units, integer units, loads/stores per cycle etc). This benefit is highlighted by one of their main uses in prototyping digital circuit designs.

Two of the major HPC vendors are now actively pursuing this technology - Cray have been marketing a HPC system with (optional) FPGAs, the Cray XD1, and SGI are soon to be offering optional FGPA bricks that can be accommodated in an Altix.

Other vendors from the FPGA market are also targeting the HPC community – the symposium had speakers from XIlinx on the underlying hardware and future chips, Nallatech who spoke on the history and commercial realities of FPGAs, Celoxica who spoke about implementing algorithms in FPGAs, Mitrion who spoke on programming for FPGAs and Star Bridge systems who spoke about development environments for FPGAs.

This is not the new area it seems as was highlighted by many talks from the research community, including the University of Durham, University of Saarland and NASA.

It was clear from the event that FPGAs have a long way to go to achieve a more widespread adoption in the HPC community. The potential benefit for some applications seems impressive but this is less clear for floating point arithmetic, and the development environments seem immature with algorithms needing to be coded in lower