Introduction to MPI

Jake Zhao

September 25, 2012

The message passing interface (MPI) was developed in the early 1990s to run parallel code on supercomputers. The MPI paradigm assumes that the system has distributed memory, i.e. each processor has access to a separate memory block. Even though modern systems generally have multiple cores sharing a single memory block (or even higher level caches), distributed memory can be simulated by allocating a partition for each core. This induces some inefficiency but, on the other hand, there is a host of additional problems that must be considered when working under the shared memory framework.

At this point, you should have a Fortran compiler already installed in your system. Open MPI[1] is a free open source MPI implementation that should work with any Fortran compiler (gfortran, ifort, g95, etc...).

MPI_example.f90 introduces you to the basics of MPI and performs a sum along a large matrix's columns. To compile the program, type the command "mpif90 MPI_example.f90 -o mpi_example -O2" where the -o flag specifies the output file name and -O2 is the level of compiler optimization. The matrix is set to 8000×8000 so the number of cores needed to run the program must divide into 8000 (you can modify the matrix size and the number of cores to evenly split the work on your system which may have a different number of cores). The supercomputer I used has 8 cores to a node so that I can type the command "mpirun -np 8 mpi_example" to run this program where -np is a flag indicating the number of cores

[1] http://www.open-mpi.org/

to use. The main thing to notice in the program is that even though the 8 cores share a single memory block, I had to call the MPI_scatter subroutine to "scatter" the submatrices to each processor's local memory partition (which is implicitly managed by the interface). Each allocation, initialization, or computation on a variable without a specified processor condition will use all the processors to perform the operation on their own local variable. As you may guess, debugging in a parallel environment can be incredibly challenging. For instance, if I had initialized the random matrix for each processor separately instead of using just the root, they would all be different and therefore the sums would be different. After all the sums are computed, they must be gathered back into one large 8000 element vector. Finally, the program checks that indeed the parallel implementation gives the same answer as the serial implementation.

HW2ns_parallel.f90 computes the non stochastic growth model with MPI parallel processing. The same set of commands are used here as in the initial example except for MPI_bcast[2]. MPI_bcast, uses the root processor and broadcasts the variable value in the root partition to the rest of the processors and their respective partitions. As an interesting excercise, you can turn off MPI_bcast to see what happens.

Once you finish programming the parallel stochastic growth model, please compare it to the speeds of the various other programming approaches (Matlab, serial Fortran, or some other language).

[2]The following link is a great reference to all the available MPI calls and necessary arguments: http://www.open-mpi.org/doc/v1.6/