Laplace Exercise Solution Review

John Urbanic
Parallel Computing Specialist
Pittsburgh Supercomputing Center

Finished?

If you have finished, we can review a few principles that you have inevitably applied. If you have not, there won't be any spoilers here. It you want spoilers, you should look at in ~training/Laplace at

laplace_mpi.f90

laplace_mpi.c

These are also good starting points for following us into OpenMP or OpenACC optimizations.

Two things I know you did.

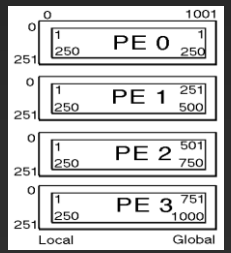
Even though I may not have been looking over your shoulder, I know that you had to apply the domain decomposition process that we discussed is universal to MPI programming. In this case you had to:

1) Identify the main data structures of the code:

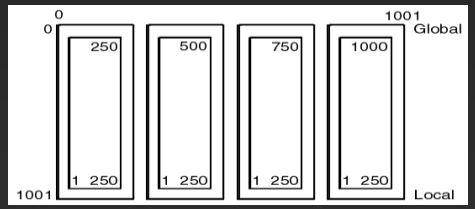
Temperature and *Temperature_old*

2) Decompose both those data structures with a forward looking strategy:

C



Fortran



Digging through the code

This code has subroutines, like all real codes. And, like all real codes, you have to follow the main data structures that you are modifying into those subroutines.

You had to modify the boundary conditions and you had to modify the IO. This is very common to an MPI port.

On the other hand, you did *not* have to modify the kernel, or real math, or effectively the science of the code. This is also typical of a real MPI port.

Running to convergence

3372 iterations to converge to 0.01 max delta.

	Serial (s)	4 PEs (s)	Speedup
C	13.9	3.80	3.66
Fortran	6.05	1.56	3.88

Better Fortran performance in this case is because the Fortran max() intrinsic vectorizes automatically. C tuning would take a little more effort.

Note that all versions converge on the same iteration. This kind of repeatability should be expected. However, exact binary repeatability is usually not possible due simply to floating point operation reordering.

Scaling off the node will typically be much better than scaling on the node for a well written problem of this type run at normal scale.

Vs. OpenMP

3372 iterations to converge to 0.01 max delta.

	Serial (s)	4 threads (s)	Vs. MPI
C	13.9	3.50	1.08X faster
Fortran	6.05	1.48	1.05X faster

Of course, for these small cases that fit on one node (16 PEs), the straight threading approach will obviously be faster, right? Above are the numbers using our OpenMP Workshop Laplace solution.

How can this be? With the MPI version, we have gone to the trouble of segmenting our data in memory. This is a big help to the cache and memory system and can often offset the advantage that the threading works in place on shared memory. More on this increasingly important topic in the Outro talk.

Improvements

The Laplace code is really a very realistic serial-to-MPI example. We can extend this example even further into the world of real application codes with some modifications that you could pursue.

1) 3D. To make this code into a full 3D application would be a straightforward, if tedious, addition of indices to the arrays and loops. However, you would have to reconsider your data decomposition and your communication routines would now have to pass 2D "surfaces" of ghost cell data instead of just strips.

2) Data Decomposition….

Data Decomposition

Whether we go to 3D or not, it is desirable, and usually necessary, that the code be able to run on variable numbers of PEs.

However, it is not bad form to limit the PE counts to some reasonable numbers. For a 2D code you might impose a demand that it be square number, and for a 3D code you might demand a cube. Often schemes lend themselves best to powers of 2 (FFTs). Machine "chunks" are often easily accommodated in powers of two.

Domain decompositions can be quite complex in applications with irregular shapes.