Parallelization Tutorial

Daniel Stubbs

McMaster University/Sharcnet

July 8, 2004

Introduction

- Overview of parallelizing a “real” PDE code, in Fortran 90, using MPI
- Assumes knowledge of programming, as well as MPI basics
- Code and slides will be available on the Sharcnet website after the talk

Motivation

- Do you need to parallelize?
- Serial codes are much easier to write, debug and maintain
- MPI may not be the best solution to your problem: OpenMP, script-based serial job submission
- Will you be able to analyze results from much larger problems?

Preparation 1

- Assuming we do need MPI, we often begin from an existing serial code
- If so, it’s much easier if this serial code is clean, modular and well-documented
- Ideally, one can isolate much of the parallel communications and reuse some of the original serial code

Preparation 2

- We will consider the following problem,

$$\frac{\partial f}{\partial t} = \sigma \nabla^2 f + \alpha f + \beta f^3$$

in two dimensions, with appropriate initial and boundary conditions

- Though not necessary, we'll also assume that all coefficients are constant
- In the serial version, we have used an explicit 4th order Runge-Kutta solver

Preparation 3

- We employ the usual finite difference stencil for the spatial derivatives:

$$\nabla^2 f = \frac{1}{\Delta x^2}\left(f_{i+1,j} + f_{i,j+1} + f_{i-1,j} + f_{i,j-1} - 4 f_{i,j}\right)$$

- Such an approximation means that we need to know the value of the function at the nearest neighbours to compute the Laplacian at a point

Preparation 4

- With such a choice, it’s easy to see that the principal bottleneck for the serial code is computing the right-hand side of the equation
- This is a triply nested loop, and so our goal in parallelizing this serial code is to somehow break apart this “loop nest”

Preparation 5

- In general, parallelizing the temporal loop is out of the question, for obvious causal reasons
- The result is the classic strategy for parallelizing PDEs, to decompose the spatial domain
- Each CPU will handle a given region of the total space in which we want to solve the equation

Caveats

- Clearly, having a very regular domain like a square, and using a uniform rectangular mesh, makes things much simpler
- With adaptive grids or irregular spatial domains, much more care and work has to go into this decomposition, to minimize communications and balance the processor load

Parallelization 1

- With the parallelization technique chosen, we can begin thinking how to implement it in code
- As a first approximation, we can begin by breaking up the x axis in two
- With the usual five point stencil for approximating the second order spatial derivatives, we have a single shared region

Parallelization 2

- At each time step, the two processes will need to exchange the value of $f(t_n, x, y)$ along their mutual border
- This essentially establishes the communication pattern for us, which we can naively implement by a pair of matching MPI_Send and MPI_Recv calls

Parallelization 3

- There’s a potential problem here though, in using blocking MPI calls, namely “deadlocking”
- If process 0 is waiting for its send to complete at the same time that process 1 is waiting for its send to complete, the program will appear to hang indefinitely
- So, need to consider some changes…

Parallelization 4

- We can avoid this with by sequencing the send and receive, which works fine with just two processors but can’t really be generalized
- A better alternative is to use some MPI calls that don’t block, such as by the sequence MPI_Irecv, MPI_Send and MPI_Wait
- For now, we stick with the simple solution of alternating the send and receive statements

Parallelization 5

- This brings us to our first parallel version of the original serial code
- While this MPI version is alright, it does have some room for improvement:
  - It only works with two processors
  - All the constants are hard-coded
  - No checkpointing

Testing

- Having created a parallel version of your program, it’s important to remember to verify its correctness
- Particularly if a program involves random numbers, this will require some work
- It’s important not just to check average quantities or rely on visual inspection

Generalization 1

- To support more than two processors, we’ll just divide the axis into n equal parts
- We will assume that the number of CPUs divides evenly the number of points on each axis
- Now most processors will have two sets of ghost points to send and receive, which does complicate the programming

Generalization 2

- This makes the use of non-blocking MPI calls even more important
- The parameters are now read in from a header file
- We checkpoint every 100 time steps, by writing the current field state to disk
- This isn’t the complete program state, but it’s good enough for our purposes

Generalization 3

- This is certainly a much better, flexible program than our first MPI-based code
- Still, there are some performance issues that could arise, mainly in terms of how we are decomposing the square
- Since we don’t know how the CPUs are wired together, we have no idea if our virtual topology is optimal for this hardware configuration

MPI Communicators

- A solution to this issue lies with using a communicator other than the usual default MPI_COMM_WORLD
- This puts MPI in charge of distributing the logical data layout over the actual hardware
- It can also simplify using higher dimensional decompositions of the spatial domain
- Now each processor has a rank within the new communicator, as well as the standard MPI_COMM_WORLD

Cartesian Communicator

- In this version of the program, we have a call **MPI_CART_CREATE**
- This creates a new communicator for us, in addition to the usual default communicator, MPI_COMM_WORLD
- We also need a new technique to find out the ranks of the two neighbouring processes, **MPI_CART_SHIFT**, and of our own rank via **MPI_CART_GET**
- Most times in 1-D, the ranks won’t change

Boundary Conditions

- At either end, this **MPI_CART_SHIFT** call will return MPI_PROC_NULL
- This ensures correct behaviour here as we have specified non-periodic boundary conditions for this axis
- The actual data exchange is similar to what we had before
- How does this code behave in terms of its scaling?

Scaling & Decomposition

- As we use more processors, how much communication will be required?
- With two processors, each one will need to send 8*N bytes/iteration
- With four processors, this becomes 16*N bytes/iteration, and in general, this remains the same as we add processors
- In short, the amount of communication will scale linearly with the number of processors, very undesirable behaviour

Scaling, continued

- What about a two-dimensional decomposition?
- In this case, with P processors, we have 2*N/Px + 2*N/Py with P = Px*Py
- We see that the amount of communication starts to shrink as we increase the number of processors, with this decomposition
- For such a finite difference algorithm, the communications will be proportional to the surface area of the processor cells

2-D Decomposition

- Happily, using the MPI Cartesian communicator makes it relatively painless to switch from a 1-D to a 2-D decomposition
- The principal changes now are adding a **MPI_CART_SHIFT** call to get the two additional neighbours, and the added MPI messages to swap ghost points

Conclusion

- With this change, we now have a parallel solver for this nonlinear PDE that is robust, flexible and scales well
- Some obvious improvements to the present code:
  - to handle situations in which N mod numprocs isn’t zero
  - to rewrite the code as a Fortran 90 module
  - to make the disk I/O and checkpointing less of a bottleneck

Conclusion, cont.

- For more complicated and realistic geometries and meshes, there are various libraries and toolkits available that may help
- Additional resources include Sharcnet staff, books and the possibility of more extensive courses on this and similar topics, such as in Montreal this year
- At McMaster next summer?