Send and Receives in Time-step Sub-cycles

1 Abstract

This paper comments on the efficiency of both blocking and non-blocking I/O implementations of a domain decomposed radiation transport problem. Due to synchronization and timing issues, blocking receives are necessary for the sake of simplicity and efficiency.

2 Introduction

Radiation transport problems require, at each time-step, every particle to run to completion. Within a domain decomposed problem this entails a separate looping construct within each time step to loop until all particles have been passed and received and have run to completion. A single particle can pass between multiple domains numerous times before exceeding its time limit or running out of energy. One obvious question is how should the send and receives be structured within the sub-cycles to achieve maximum efficiency. I explore this question in the following sections.

3 Problem and Discussion

Before I describe the problem, I will first cover some of the basic terms used in this paper.

- **Domain** A portion of the problem description on an individual node of a distributed cluster.
- **Frozen Particles** Particles waiting to be sent to another domain before they have run to completion in the current time step.
- **Mesh** Geometric problem space in which the particles move. In a domain decomposed problem, the mesh is partitioned into separate **Domains** and loaded onto unique nodes of a distributed cluster.

Consider the following pseudo-code for a single time-step:

```
\\begin time-step
0. run_particles();//run the initial set of particles

1. while(there_are_still_particles())
   {
2.     physics_send_particles_to_other_domains();
3.  receive_particles_from_other_domains();
4.  run_particles();//perform physics on received particles
   }
```

The algorithm goes as follows. Given an initial set of particles (whose generation is not demonstrated in the above code snippet), run the particles until all particles are either **Frozen** or have run to completion. This is done in the method in line 0. The next step enters the sub-cycle loop which continues until there are no active particles in the system. The sub-cycle sends particles to their respected destination domains, receives all the particles which have been sent by other nodes, runs the received particles until they are **Frozen** or have run to completion, and repeats.

However, one must be careful with the nature of the send and receives used in lines 2 and 3. For example, assume that the method receive_particles_from_other_domains() uses a nonblocking receive. Immediately you are confronted with the following problems.

The first can be shown in the following example. In a 2 domain problem, assume domain 0 sends some number of particles to domain 1 and then progresses to line 3 before domain 1 has a chance to send its particles. Domain 0 then skips line 4 because there are no particles to run and goes up to line 1. Domain 1 in the meantime sends its particles and receives the particles sent by domain 0. Domain 1 then runs its particles to completion such that none of them want to cross over to domain 0. When domain 1 reaches line 1 where domain 0 is waiting to check if the system has zero particles, a reduce message (the sum of all frozen particles) lets both of the domains know that the system is free of frozen particles and both problems exit. However, there are still particles waiting in the mpi buffer of domain 0 that have not been received! The problem exacerbates if this happens to occur during a run with multiple-time-steps as there is a possibility the particles waiting in the buffer will be used in the subsequent time-step, compromising the correctness of the solution.

One can avoid the problem described above by placing an MPI_Barrier() construct between lines 2 and 3. However this is not a desirable solution because the overhead in calling Barrier requires communication between all nodes, on top of the nonblocking receives. Simply employing a blocking receive on line 3 would be much more efficient than a Barrier construct. A second solution may be to have the nonblocking receives sit in a "busy wait" until all domains have checked in with their particles or sent a message that there are no particles to send. This solution mirrors a blocking receive in functionality but it has the following drawback, forcing the code to sit in a "busy wait" tells the resource scheduler that it is indeed active and therefore unnecessarily ties up cpu time. A blocking approach, on the other hand, allows the system to schedule other processes until the message is received.

4 Conclusion

The sub-cycles to ensure all particles pass between domains to completion within a time-step should be implemented with blocking receives. Blocking receives inherently synchronize all domains without the unnecessary overhead of using barriers or busy waiting loops.