MPI Proto: Simulating Distributed Computing in Parallel

Omari Stephens

1 Introduction

MIT class 6.338, Parallel Processing, challenged me to, vaguely, do something with parallel computing. To this end, I decided to create an MPI-parallelized implementation of the simulator for the Proto language, a product of the Space/Time Programming Group[1] at the MIT Computer Science and Artificial Intelligence Lab (CSAIL).

This paper describes the design and implementation of the MPI-parallel simulator, dubbed *MPI Proto*. Section 2 quickly introduces Proto and describes the design of the simulator. Section 3 covers important details and caveats of the serial implementation of the design. Finally, Section 4 details the changes in design and implementation which I performed to create the parallel implementation.

2 Proto

Proto is a programming language aimed at ensembles of agents, such as those in a sensor network or a swarm of robots (generally, "spatial computers"). Proto is, essentially, a distributed streaming language. A single program is executed repeatedly for the life of the agent, and the values calculated by an agent and each of its neighbors during a round of computation may affect the subsequent round of computation. Thus, the relevant computational state consists of the program, the computed values of the agent of interest in the previous round of computation, and the values computed by that agent's neighbors in the previous round of computation.

To program the ensemble, the programmer writes a program, which is compiled to bytecode. Each agent runs a virtual machine which receives the bytecode, and then exe-

[1]`http://people.csail.mit.edu/jrb/stp/stpg.htm`

cutes a round of computation, followed by a broadcast during which that agent's computed values are sent to all of the other agents in listening range. This cycle of computation and communication continues until the death of the agent.

2.1 Simulation

To test the language without having to purchase, maintain, and physically handle a an army of hardware units, a simulator exists which places devices in a cartesian plane and provides methods for them to communicate with one another. Since one major goal of Proto is to function well on ensembles in the dense limit, like droplets in a computational puddle, programs generally use primitives which do not allow them to identify individual neighbors — instead, they deal with proportions, thresholds, and other aggregative measurements which remain feasible when the number of neighbors grows unboundedly.

To maintain this abstraction, the simulator has two decoupled layers for computation and communication, the interaction of which is managed by a scheduler and a buffer for both incoming and outgoing computed data ("exports"). The computational portion of each simulated device grabs the exports from the buffer, executes a single round of computation, then writes its own updated export to a similar buffer. After computation, the device schedules a broadcast and then idles.

Note that the computation layer corresponds to a physical device in a spatial computer — it grabs information from a buffer, percolates, and then sends a broadcast. It does not (necessarily) know who its neighbors are, whether its broadcasts were successfully transmitted, or the age/latency of the data it is computing on. To complete the physical analog, the communication layer approximates the electromagnetic physics of the physical system. This layer *does* know what radio messages should go where and handles the delivery of those messages. It keeps track of distance approximations to each of the neighbors (which accompany each respective neighbor's entry in the export buffer), and can, for instance, probabilistically simulate radio transmission (TX) and receipt (RX) errors to better approximate true radios.

The communication layer accomplishes this delivery by first keeping track of each device's neighbors — the other devices within the so-called radio range of the TX device. When the scheduler tells a device's communication layer to execute a broadcast, the layer grabs the device's export from the buffer and sends it to the communication layers of all of the neighbors (the other devices within the radio range). Finally, it schedules a round of computation with the scheduler and then idles. Conversely, when the communication layer receives an export, it stores the export in the buffer along with the estimated distance to the neighbor.

Thus, each device continually performs interleaved stages of computation and communication. The devices have a pre-set desired period of operation, and schedule each consecutive action a half-period away from the end of the prior action. So a broadcast will be scheduled to begin a half-period after the *end* of a round of computation, and vice-versa.

3 Proto in Serial

3.1 Architecture and Design

The simulator is written in a very modular fashion in C++. The SpatialComputer class is a container which represents the ensemble of machines in both their physical and computational aspects. Of importance for the subsequent parallelization of the simulator, the SpatialComputer contains a scheduler, a physics (implemented either as points and forces, or farmed out to the Open Dynamics Engine for better realism), and a population of devices (each of which houses its own computation and communication layers). It is important to note that while movement is supported by the serial simulator, it is not yet supported by the parallel implementation, so I will not discuss it further.

The SpatialComputer controls simulation by incrementing the simulator time, and then calling the “evolve” function with the simulator time. The evolve function pops events off of the scheduler (which is implemented as a priority queue on time) and processes those events until the scheduler has caught up to the simulator time, at which point the function returns. Event processing generally consists of telling devices to perform a round of computation or a broadcast.

3.2 Communication Implementation Details

Unfortunately, the communication infrastructure functions in a manner that is tightly tied to uniprocessing. As a reminder, when a device’s communication layer executes a broadcast, it sends the TX device’s export to the communication layers of all of the RX devices. This is actually implemented as a function call — the transmit method of the TX device iterates through all of the RX devices, and for each of them, calls the recipient’s receive method with the export contents as function arguments.

The communication layer’s receive method is actually a simple pass-through to the a kernel (virtual machine) function of the same name. This kernel method places the export into the aforementioned buffer, where the computation stage can access the data. For the kernel state to remain consistent, however, this means that computation and message receipt may not happen concurrently at all. Thus, the timings of all of the devices in a neighborhood require tight coupling of a magnitude that serial processors cannot help but provide, and that parallel systems have great difficulty with.

4 Proto in Parallel

4.1 Architecture and Design

Derived classes were the theme of the parallel implementation of Proto. The parallel implementation introduces a supervisor class, SCCluster, which handles time advancement, and the SCCElement class, which is just an parallel-environment-savvy derivative of the SpatialComputer class. Each process of the MPI program contains a single instantiation of one of the classes — the SCCluster instance lives in the rank 0 process, and every other process contains an instance of SCCElement.

After initialization, the SCCElement class waits in an idle loop for MPI messages. The messages it is prepared to handle include an “evolve” message, whose contents will be the simulator time. When the idle loop receives an evolve message, it simply calls the evolve method of its superclass (SpatialComputer). Furthermore, the supervisor waits for an ack from each SpatialComputer before advancing the simulation time again and sending more evolve messages. This two-way communication keeps the SpatialComputers (SCs) approximately in sync, and also provides for a method of rate control in case one of the SCs takes a lot longer to evolve than the others. Consequently, when the SCCluster makes progress (as measured by the advancing simulation time), it is guaranteed that the SCs are making appropriate progress to stay temporally synchronized with the SCCluster, and consequently, with each other.

The SpatialComputers are presumed to be in a Cartesian grid of runtime-settable width. They fill the grid in row-major order, from left to right across the columns. The physical area controlled by each SC is presumed to be larger, in both width and height, than the radio range of any device, and consequently, each SC only needs to communicate with up to 8 touching neighbors (4 adjacent and 4 corner).

4.2 Implementation Challenges

4.2.1 Device Numbering

In the serial version, device numbers were only required to uniquely identify the devices in a single process. Consequently, the Device class contained a static member holding the number of devices currently instantiated. Any new devices would take this device ID and increment the member by 1, ensuring that device IDs remained unique.

With multiple processes (and, more specifically, a different address space for each process), this method no longer maintains global uniqueness. Consequently, device IDs were created as a convolution of this number with the rank of the processor. To allow for large quantities of devices per process, in addition to large numbers of processes, the device ID is now an unsigned 32-bit number, with the top 14 bits identifying the rank of the host process, and the low 18 bits identifying the specific device on that host process.

Thus, simulations can span up to 16,384 processes (which is reasonable, given the availability of machines like the almost 6,000-node SiCortex machine), as well as up to 262,144 devices per process (which is currently many more than any machine we have access to can handle). Thus, this split is flexible enough to run on a single, very fast machine as well as a very distributed, slow machine (like the SiCortex). Of course, a very distributed, very fast machine will also work, but would also likely be able to handle more devices than can be easily be represented in 32 bits.

4.2.2 Discovery of Remote Devices

Advancing from the serial implementation, the first task is for a SpatialComputer to discover devices that its own devices may want to communicate with, but that may be hosted by another SpatialComputer. For instance, if the four quadrants of the Cartesian plane represented four SCs, devices located at coordinates $(5, 1)$ and $(5, -1)$ should likely be able to communicate, despite being hosted by different SCs.

To accomplish this, each SCCElement (a derivative class from the SpatialComputer) constructs a summary list of its devices, which it subsequently communicates to its (up to, omitted henceforth) 8 neighboring SCCElements. Each element in the list consists of a device ID and device coordinates. Thus, since each SCCElement knows the width of the spatial computer cluster, the number of SpatialComputers, and the width and height of each SpatialComputer in the cluster, each SCCElement can construct a map of devices in the area covered by itself and its 8 neighbors.

Consequently, this constructed map allows an SCCElement to determine which remote devices it needs to communicate with. Furthermore, since the device ID of each device is unique, and since that ID identifies the device's host process, the SCCElement can treat remote devices identically to local devices. When a distinction between the two needs to be made (such as for communication), it can be made.

4.2.3 Communication with Remote Devices

As discussed in the previous paragraph, communication is an instance where local and remote devices need to behave differently. To accomplish this bifurcation, all devices are instantiated from the RemoteDevice class, which is a derivative of the standard Device class. This class provides a communication API which does the right thing in both cases — it simply calls a function in the local address space for local devices (where the device's host process is identical to the rank of the process executing the code), and sends an MPI message to the host process for remote devices.

When the SCCElement idle loop receives an export MPI message, it simply calls the RemoteDevice receive method again in the local address space. Again, this does the right thing and simply passes the message through to the kernel, as in serial Proto.

4.3 Analysis of This Architecture

This architecture works very well. It allows for tremendously large simulations while still conforming to the tight-coupling constraints of the Proto kernel and infrastructure. The convoluted (in the mathematical sense :o) device IDs allow the SpatialComputer to work correctly without having to know or care about whether devices are local or remote Furthermore, these device IDs make debugging easier than it might be — the IDs can easily be deconvoluted to determine the host process, and when written in hexadecimal, the host process can often simply be read off by inspection.

4.4 Unresolved Challenges/Future Work

In the initial implementation, contention was a big problem as network sizes increased. The problem was that blocking MPI sends to remote neighbors were being done from inside of the evolve call, which was itself called from the idle loop. Thus, messages could not be both sent and received concurrently by a single process since the message receipt also happens in the idle loop.

4.4.1 MPI Send Queue

To alleviate this problem, I implemented a circular MPI send queue. Instead of executing an MPI send command directly, RemoteDevices would simply add the messages to the queue, and a separate thread (running in the same process, and consequently, the same address space) would dispatch the messages.

The two main advantages of this change are that SCCElements now take considerably less time to evolve, and messages can be batched. That is, in the vast majority of cases, a span of sends in the queue will have the same destination (since the communication layer queues a send for each neighbor that should receive the message, and the remote devices from a certain remote process are contiguous on the list of neighbors). When it detects this situation, the send queue sends a large message to the remote process rather than a number of smaller, less-efficient messages.

This change has a single major disadvantage, as well as an implementation problem. The disadvantage is that asynchronous sends allow for latency between message transmission and receipt. Depending on how fairly or unfairly the MPI subsystem queues messages, this has the potential to lead to constantly-growing latency. To deal with this problem, the implementation waits to acknowledge the SCCluster's evolve message until it has dealt with incoming exports. Thus, the rate control protocol prevents starvation even though it is still susceptible to certain amounts of latency.

The implementation problem is that, on the system where this was tested, OpenMPI was compiled without thread-safety (likely to avoid performance problems). Thus, running at full speed, the simulation rarely completes without receiving a Segmentation Fault at the OpenMPI level. Trying to synchronize MPI communication has lead to deadlocks which I have not yet been able to fix. Thus, while MPI Proto is tantalizingly close to working, it does not yet.

4.4.2 Extra Features

The serial Proto simulator supports movement and visualization, neither of which is supported by this parallel implementation for lack of time. Adding visualization back should be fairly straightforward, either by sending state messages to a visualization process, or using a library such as the MPI Parallel Environment[2], which provides primitives for distributed use of a single X display.

Movement will likely be trickier. If SCCElements are only to have 8 neighbors to deal

[2]Developed by the Argonne National Lab

with, moving devices will need to be able to migrate from one host process to another. If no migration happens, then processes may, potentially need to speak with every other process. Furthermore, recalculating appropriate neighborhoods (to determine where a broadcast should be received) will likely be very expensive.