Parallel Processing: A KISS Approach

Ronald Marsh, Ph.D.

Computer Science Department
University of North Dakota
July 2006

rmash@cs.und.edu

This material was developed with support from North Dakota EPSCoR.
# Table of Contents

Chapter 1: Introduction to Parallel Processing .............................................................3

Chapter 2: CRCRAM ..................................................................................................16

Chapter 3: Serial Template #1 (serial) .................................................................21

Chapter 4: Master-Slave Template #1 (MST) .........................................................51

Chapter 5: Coordinating Peers Template #1 (PTP) ..............................................60

Chapter 6: Serial Template #2 (serial_2) ..............................................................80

Chapter 7: Master-Slave Template #2 (MST_2) ....................................................90

Chapter 8: Coordinating Peers Template #2 (PTP_2) ........................................119
Chapter 1: Introduction to Parallel Processing

1. Introduction.

The intent of this tutorial is to provide design heuristics and sample programs to help UND researchers port their compute intensive research to the Computational Research Center’s (CRC) High Performance Computing Cluster (HPCC) also known as “Shale”.

What is Computational Science why it is important for UND? Computational Science is the application of mathematical simulations and models to understand complex phenomena. When combined with high performance computing, the use of simulations and models allow the scientist to manipulate the phenomena in ways that are beyond any reach using traditional scientific methods. Thus, according to the National Science Foundation, Computational Science which was once viewed as a mere adjunct to theoretical (1st branch) and experimental (2nd branch) scientific approaches, is now emerging as a principal means of scientific research and considered to be the 3rd branch of scientific endeavor. It is an excepted fact that without access to high-end computational capability, many important discoveries in the fields of chemistry, biology, oceanography, meteorology, and many others could not have been made. Finally, according to heralded science journalist George Johnson “As research on so many fronts is becoming increasingly dependant on computation, all science, it seems, is becoming computer science.”

Unfortunately, like many modern scientific disciplines, Computational Science has become a diverse field and it would take volumes to properly describe each domain. Even scientific computing, the domain that focuses on simulation and modeling software development, includes many paradigms of software development, including:

- Iterative parallelism – where a program has several identical processes wrapped in a loop.
- Recursive parallelism – where a program uses recursion to work on different parts of the data.
- Producers and Consumers – where processes communicate or share data.
- Clients and Servers – where clients request services and servers provide services.
- Interacting peers – distributed systems that support many other paradigms:
  1. Master-slave – where 1 machine coordinates the processing.
  3. Heartbeat algorithms – where interacting processes periodically exchange data (with forced timing constraints).
  4. Pipeline algorithms – where data flows from 1 process to another.
  5. Broadcast algorithms – used for decentralized decision making.
  6. Token-passing algorithms – used for decentralized decision making.
  7. Probes and receives – to disseminate and gather data from trees and graphs.
  8. Replicated server processes – used to manage multiple instances of resources.
Obviously, the development of scientific simulation and modeling software is a very complex task. Fortunately, many scientific simulation and modeling tasks intended for execution on a HPCC can be represented by the interacting peer paradigm. Therefore, I will use the KISS approach to describe the development of two common HPCC software architectures, have provided code examples for both architectures, and the associated documentation. I have also provided a utility for implementation of a special class of HPCC problem and have provided the associated documentation.

2. Background.

Before we can delve into the software development process we must have a basic understanding of the different computing platforms that exist, the intraprocess communication requirements, and the intraprocess management styles that are common.

2.1. High performance computing platforms.

Each platform type requires the developer to abide by a particular set of design criteria and may put limitations on the software’s performance that the software developer must understand. In keeping with the KISS approach, we will briefly review only 2 classes of platforms:

- The first class is the classic “supercomputer.” A multiple CPU shared memory platform. Supercomputers tend to be expensive due to extreme memory bandwidth requirements. However, supercomputers are very flexible and the software development is much less constrained.

- The second class includes the cluster, Network Of Workstations (NOWs), or GRID. Clusters are much less expensive than supercomputers because they use commodity personal computers linked together with commodity networking. However, clusters cannot deliver the level of performance of a supercomputer and have more software development design constraints. Yet, for many scientific problems, clusters provide an affordable way to achieve high performance computing. Since the Computational Research Center’s High Performance Computing Cluster is a cluster, we will assume this class of machine for all software development covered by this tutorial.

To make the best use of a cluster, the researcher must be able to spread their workload across the different computers (we will refer to the individual computers making up a cluster as nodes) making up the cluster. It is this parallelism that achieves high performance computing. However, when developing an application for a cluster, the communication costs associated with having to spread the computations across many different nodes and the synchronization of the computations across those different nodes introduces several software design constraints that are unique to clusters.
2.2 *PVM vs MPI.*

When developing an application for use on a cluster, one must adopt/implement some form of communication to pass data and provide for the synchronization of the software across the different nodes that comprise the cluster. While many techniques may be implemented, the most common is to use some form of *message passing*. While several message passing methodologies have been developed, the two most common are arguably:

- **MPI (Message Passing Interface)** - a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementers, and users. A free version of MPI (MPICH) is available at [http://wwwunix.mcs.anl.gov/mpi/mpich/](http://wwwunix.mcs.anl.gov/mpi/mpich/)

- **PVM (Parallel Virtual Machine)** - a software package that permits a heterogeneous collection of Unix and/or Windows computers linked together by a network to be used as a single large parallel computer. The freely available package has been compiled on everything from laptops to CRAY's. See: [http://www.csm.ornl.gov/pvm/pvm_home.html](http://www.csm.ornl.gov/pvm/pvm_home.html). Finally, PVM is distributed as a RedHat RPM, as part of the standard RedHat distribution and is, thus, very easy to install.

The methodology that best suits your requirements depends on what you want to do. Here are 4 comparisons:

1. **Homogeneous vs. Heterogeneous cluster** - If your cluster includes several different system types (e.g., Linux, NT, Solaris, Irix) then PVM is would be better as it has the innate ability to handle conversions between binary data formats and data type size differences. MPI assumes a more homogeneous environment and competing implementations of MPI will not interoperate. That is, a group of Linux boxes running MPICH will not talk to a group of Irix boxes running SGI's native MPI.

2. **Dynamic vs. static number of processes** - PVM allows one to spawn processes as one needs them. MPI has only recently added this capability, thus most MPI implementations ask for the number of processes when the job is launched.

3. **Simple vs. complex programming interface** - PVM has a relatively few function calls; MPI has a rich variety of function calls. When one of the MPI constructs, such as Cartesian communicators, fits your problem very well, then PVM appears to be much harder to use. In contrast, when your problem only needs a simple data distribution, MPI appears to be overly complex.

4. **Trendiness (for lack of a better term)** - A lot of researchers are using MPI now, as opposed to their use of PVM 3-4 years ago. A lot of papers are being published, MPI implementations are being improved, and vendors are optimizing their MPI
libraries more than their PVM libraries. How valuable this is to you as an individual isn't obvious.

Both packages do essentially the same thing. Thus, I used both, PVM for one example and MPI for the other. A more complete comparison can be found at:  
http://www.epm.ornl.gov/pvm/PVMvsMPI.ps

Master-slave vs coordinating peers.

Thinking back to our introduction of the paradigms of software development for HPCC, we recall that many scientific simulation and modeling tasks can be represented by the interacting peer paradigm. The two most common paradigms used are the master-slave paradigm or the coordinating peers (or heartbeat algorithm) paradigm:

- The master-slave paradigm is commonly used for *embarrassingly parallel* processes, for processes where the process time of each datum varies greatly, and for processes where the partitioning (into a parallel process) has resulted in much more data partitions than nodes. It is generally accepted that the master-slave paradigm is the simplest algorithm to implement due to the much more flexible synchronization needs. In many cases, the master-slave paradigm is very scalable and variations in node performance have little impact. However, the master-slave paradigm requires an extra node to cover the master’s duties.

- The coordinating peers paradigm is commonly used for *tightly bound* processes, for processes where the process time of each datum varies little, and for processes where the partitioning (into a parallel process) has resulted in fewer data partitions than nodes. The coordinating peers paradigm is frequently used with a *heartbeat algorithm* to synchronize the peers. It is generally accepted that the coordinating peers paradigm is the most difficult algorithm to implement due to the more stringent and complex synchronization requirements. Variation in node performance usually has significant impact (due to heartbeat synchronization, the algorithm will execute at the rate allowed by the slowest node). In many cases, the coordinating peers paradigm exhibits limited scalability.

Note that in many cases, a problem can be reformulated into one paradigm or the other. Finally, in developing the examples, I execute the master-slave and coordinating-peers versions on idle nodes and busy nodes. When executed on idle nodes little performance difference was noticed. However, when executed on busy nodes, the master-slave version always outperformed the coordinating-peers version. Remember, a coordinating-peers version is only as fast as its slowest node.

3. Case studies.

3.1 *Embarrassingly parallel*.

An embarrassingly parallel problem is one for which no particular effort is needed to partition the problem into a number of parallel tasks and there is no essential dependency (or communication) between those parallel tasks. Each task can be computed
independently from every other task, thus each task could be run on a separate node to achieve quicker results. Examples of embarrassingly parallel problems include:

1. Distributed rendering of non-real-time computer graphics or of computer animation. Each frame may be rendered independently.
2. Brute force searches in cryptography. A notable real-world example is Distributed.net.
3. BLAST searches in bioinformatics.
4. Computer simulations comparing many independent scenarios, such as climate models.

Embarrassingly parallel problems are ideally suited for distributed computing over the Internet (e.g. SETI@home), on a GRID, and are commonly assigned to server farms (e.g. a very large NOW). Embarrassingly parallel problems lie at the “easy” end of the spectrum of parallelization. Let’s look at an example in some detail: The development of a computer animated movie.

During the 2006 spring semester computer graphics course (CSci446) at UND, teams of students were each tasked with developing a 4 minute computer animation movie. The students used OpenGL (a standard graphics library for developing 3D animations such as video games) to create their components (scenes, characters, etc). However, since computer animations developed using OpenGL do not have the level of realism that was desired they had to port their animations to POVRAY (Persistence of Vision Raytracer) to achieve the level of realism that was desired. To port their OpenGL images to POVRAY, the students used RayGL (an OpenGL to POVRAY conversion library developed by UND CS graduate student Kris Zarns).

In order to achieve realistic motion, a computer animation requires about 25 frames per second. Thus, each 4 minute movie required about 6000 frames to be generated. To achieve a high quality rendering, the POVRAY raytracing time was as high as 45 minutes per frame, with an average time of 20 minutes per frame. Thus, using a single computer, the rendering time would have been nearly 3 months! If the rendering could be ported to the Computer Science department’s 16 processor cluster (the CrayOwulf), the rendering time would be reduced to 5 days. Since, each frame is stored as a separate and independent file, since POVRAY can only process 1 frame/file at a time, and since it does not matter what order the frames/files are processed in (only their final accumulation), this is an embarrassingly parallel problem as all we need to do is start 16 versions of POVRAY and instruct each to render a different frame until all of the frames have been rendered.

Note that if we were to extend the movie length to 1 hour, the single computer rendering time would jump to 3.7 years! Embarrassingly parallel does not imply computationally easy.
To summarize, if you have a problem that either consists of many separate and independent data files or datum, you probably have an embarrassingly parallel problem. Furthermore, if you a program to process those individual files, you should be able to use a master-slave utility I ported to Shale called CRCRAM (originally developed for rendering the CSci 446 movies). CRCRAM is covered in detail in chapter 2.

3.2 Loosely coupled.

We define a loosely coupled problem as one where the frequency of the synchronization between the parallel tasks is low, such that it is possible to perform many tasks before synchronization must occur. Let’s look at a simple example in some detail: The multiplication of a vector by 2 matrices.

In computer graphics it is common to achieve affine transformations by multiplying each 3D point (represented by a vector) by a transformation matrix. If multiple transformations are required, the point is multiplied by multiple transformation matrices. We can depict this as:

\[
\begin{bmatrix}
 x' \\
 y' \\
 z' \\
 1
\end{bmatrix} = \begin{bmatrix}
 1 & 0 & 0 & T_x \\
 0 & 1 & 0 & T_y \\
 0 & 0 & 1 & T_z \\
 0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
 x \\
 y \\
 z \\
 1
\end{bmatrix}
\]

Where \( <x', y', z', 1> \) describe the 3D point’s transformed location, the leftmost 2D matrix describes the translation, the rightmost 2D matrix describes the rotation, and where \( <x, y, z, 1> \) describe the 3D point’s original location. Note that we will perform the multiplications right to left, such that we rotate the point before translating it. Thus, algorithmically, the operation will look like:

\[
\begin{bmatrix}
 X \\
 Y \\
 Z \\
 1
\end{bmatrix} = \begin{bmatrix}
 1 & 0 & 0 & 0 \\
 0 & \cos(\phi) & -\sin(\phi) & 0 \\
 0 & \sin(\phi) & \cos(\phi) & 0 \\
 0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
 x \\
 y \\
 z \\
 1
\end{bmatrix}
\]

followed by:

\[
\begin{bmatrix}
 x' \\
 y' \\
 z' \\
 1
\end{bmatrix} = \begin{bmatrix}
 1 & 0 & 0 & T_x \\
 0 & 1 & 0 & T_y \\
 0 & 0 & 1 & T_z \\
 0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
 X \\
 Y \\
 Z \\
 1
\end{bmatrix}
\]

where \( <X, Y, Z, 1> \) describe a vector for storing the intermediate results.
We now have what could be considered a pair of embarrassingly parallel problems as for each vector-matrix multiplication there is many different parallelization partitions possible and no synchronization. The only synchronization that we have with this problem overall is that the first matrix-vector multiplication must be completed before the second matrix-vector multiplication is begun. Therefore, we have infrequent synchronization with no intranode data exchange, hence a loosely coupled problem.

Since the intuitive approach for any vector-matrix multiplication is to multiply each matrix row by the vector forming an intermediate vector value and since this approach results in a parallelizable operation, we will take this approach. Finally, it does not matter what matrix row is operated on first, only that all 4 rows eventually get multiplied (before we begin processing with the second matrix).

Unfortunately, this is a “toy” problem and unsuited for parallel processing. We can, however, make two revisions and turn it into a problem well suited for parallel processing:

1. We will assume that each vector has 100,000 elements and that each matrix has 100,000x100,000 elements.
2. We will assume that the row-by-row multiplication has very non-uniform computational costs.

Given the above revisions, this problem is now well suited for parallel processing using the master-slave paradigm.

To summarize, if you are developing a program that can be expressed as, or nearly as, an embarrassingly parallel problem, that partitions into many more data segments than available nodes, that may have non-uniform computational cost per data segment, and that requires only occasional synchronization, you should be able to derive a program using the two Master-Slave examples that I wrote (see Chapters 4 and 7).

3.3 Tightly coupled.

We define a tightly coupled problem as one where the frequency of the synchronization between the parallel tasks is high and where it is impossible to perform many tasks before synchronization must occur. These tend to be the most difficult (logically and programmatically to develop). So let’s look at a simple example in some detail: A 1D heat transfer simulation.

For this example, we will model the environment by a 12 element 1D array “M”. On the left side there is a source of constant heat, while at the right side there is a source of constant cold. Pictorially, this looks like:

```
Heat   Cold
```

9
The algorithm we will use to simulate the heat transfer is a simple 3 element neighborhood averaging, where each array element becomes the average of itself and its 2 neighbors. Equation (1) depicts the algorithm:

\[ M_i' = \frac{1}{3} \sum_{x=-1}^{1} M_{(i+x)} \]

To facilitate the simulation of the constant heat and constant cold sources we will extend array “M” to 14 elements with the outer elements having fixed values. The constant heat source is colored red and the constant cold source is colored blue:

The extended version of “M” has array indices of 0 to 13 (C/C++) and since the two end points are points of constant value, this extension allows us to directly implement equation (1) by allowing “i” to range from 1 to 12. We no longer have to deal with the (original problem’s) end points separately.

Before we partition this into a parallel problem, note that the algorithm is iterative. A single pass of the neighborhood averaging procedure will not produce correct results. We must apply the neighborhood averaging procedure repeatedly until convergence occurs (until no significant changes occur from iteration to iteration).

We will arbitrarily partition the problem space into 3 non-overlapping equal sized segments and assign each segment to 1 of 3 nodes on the cluster:

Immediately a problem arises. The calculation involves a neighborhood average, yet the array elements (shown in gray) along the segment edges do not have all of the required neighbors as these “missing” neighbors have been assigned to other nodes. In order to retain usage of our equation and insure identical results to the sequential version, the necessity for “halo” elements (shown in green) arises:

We now have a problem that is uniform across the 3 nodes, in that all local arrays are of the same size (1x6) and whose outer elements have fixed values (at least with regards to the “local” neighborhood averaging procedure). If this problem
only required a single pass of the neighborhood averaging procedure, it would be an embarrassingly parallel problem. However, this algorithm is iterative and requires the application of a heartbeat algorithm to manage the synchronization of updating the “halo” node values. Prior to each local application of the neighborhood averaging procedure, the halo elements must get their appropriate values from the neighboring node:

![Image of three nodes with heat and cold elements]

It is the frequency and complexity of the “halo” updates that make this a tightly bound process. Each node must wait for all of its neighbors to complete their own processing, before it can continue on to the next iteration (the “heartbeat”). As a result, the performance of this algorithm (any many other coordinating peers /heartbeat algorithms) is limited by the slowest node.

To summarize, if you are developing a program that must be expressed as a set of neighborhood processes and that requires frequent updates of “halo” elements, you should be able to derive a program using the Coordinating Peers example that I wrote (see Chapters 5 and 8).

4. How to partition.

The first question the budding HPCC user must ask is: “Is my problem parallelizable at all?” Many problems simply are not parallelizable. The second question is: “How should I partition my problem such that it is parallel?” The heuristic is to partition the problem into as many pieces as possible and then to aggregate the pieces into something that is feasible for the cluster it will be run on. The primary concern is that there is always communication overhead in any parallel processing application and you want to make sure that your processing time per datum is much greater than the communication costs per datum. Another concern is that many parallelizable problems are only parallelizable to a certain extent. In other words, the problem may be parallelizable over 4 processors, but not 8 (at least not effectively).

Ian Foster has developed a 4-step design process that is useful when developing parallel algorithms. The steps are:

1. Partitioning
2. Communication
3. Agglomeration
4. Mapping
4.1. *Partitioning.*

The goal here is to divide the problem into as many small parts (called primitive tasks) as possible as the number of primitive tasks is the upper bound to the parallelism possible. There are 2 decompositions possible:

- Domain decomposition (data decomposition) – divide the data into pieces and associate data with computations. Each association is called a primitive task and we want to maximize the number of primitive tasks.

- Functional decomposition (algorithm decomposition) – divide the algorithm into pieces and associate computations with data. We can often derive pipelines (sequences) of processes, if nothing else. Again, each association is called a primitive task and we want to maximize the number of primitive tasks.

4.2. *Communication.*

We want to minimize all communications as it is overhead that is not present in a serial algorithm. Thus, we must determine the communication requirements of our algorithm by considering the following:

- Are the communication operations balanced among tasks? *Preferably they are.*
- Does each task only communicate with a small number of neighbors (local communication) or must it communicate with many other tasks (global communication)? *This will depend on the application.*
- Can tasks perform their communications concurrently? *Preferably, yes.*
- Can tasks perform their computations concurrently? *Preferably, yes.*

4.3. *Agglomeration.*

The goal here is to combine tasks to reduce communication, improve performance, and reduce the software development time. Agglomeration should achieve the following:

- Reduce communication – If we have many simple tasks that communicate, we can reduce communication costs by combining these tasks into 1 large task. This is especially true if any of these tasks is being held up by any other task. Another method is to combine tasks that send/receive a lot of information into 1 large task as it is better to send 1 large message than many small messages.

- Improve performance – We want to keep our design parallel enough such that it will scale up (if required) to a larger problem and/or onto a machine with more processors. As the problem size increases the number of tasks should increase. Also, tasks should have similar computational and communication costs.

- Manage software development time – Assuming that we are parallelizing a serial algorithm we want to use as much of the existing serial code as possible as this
will reduce the time and cost of developing the new code. We want the trade-off between the cost of developing the new code and the cost of modifying the old code to be reasonable.


The goal here is to assign tasks to processors without greatly increasing communication costs:

- When using domain/data decomposition the resultant tasks are often of similar size and if the communication pattern is regular you would map each task to a processor. If the communication pattern is local and frequent (tightly bound) we would use the coordinating peers paradigm. If the communication pattern is global and infrequent (embarrassingly parallel or loosely bound) we could use the master-slave paradigm.

- If the number of tasks is fixed and the communication pattern is regular but the tasks take greatly differing times to complete, we would look for a pattern of computation cost and develop a cyclic allocation of tasks to processors (load balancing via a master-slave paradigm). If we do not know the pattern of computation cost, we could still use the master-slave paradigm using a naive load balancing algorithm (such as that provided in the Master-Slave Template).

- Some problems result in an irregular communication pattern and it is best to statically map tasks to processors to reduce communication.

- Some problems require many short lived tasks that have little communication needs. These have few mapping constraints.

5. *Compiler optimizations.*

If one is to spend the time developing code for a HPCC, it would be foolish not to also consider optimizing the code for the intended architecture (or to use a language not intended for high performance computing!). Thus, here is a brief tutorial on code optimizations.

Modern compilers are extremely good at optimizing code, yet they operate under several restrictions:

- They must not alter correct program behavior.
- They have limited understanding of the problem.
- They need to complete the compilation task quickly.

Since the compiler only optimizes limited sections of code at a time, it has a limited understanding of the code. Therefore, situations can occur that will force the compiler to abandon the optimization. Function calls are one such situation. Of course some function
calls are required (to libraries, etc) and software engineering principles tell us to use functions to improve code readability and maintainability. Another issue is the development of loops. It is well known that most programs spend most of their time within a loop. In staying with the KISS approach we will concentrate our discussion on loops with function calls as this generally has the greatest impact on performance.

Many programs spend much of their execution time in loops. Therefore, it is especially important to be able to write loop code effectively. There are 3 basic practices:

1. Simplifying the loop construct.
2. Removing non-dependent function calls from within a loop.
3. Removing excessive non-dependent memory accesses.

Here is an example of a poorly designed loop:

```plaintext
for (I = 0; I < (s+2); I++) {
    j = cos(\phi);  // function call
    k = array_element[6];  // memory access
    new_data[I] = (j+k)*I;
}
```

The loop construct is not simple (I<(s+2)), there is a loop independent function call within the loop (j=cos(\phi)), and there is a loop independent array access within the loop (k=array_element[6]).

Here is the redesigned example where we used a temporary variable (stop) to simply the loop construct, moved the loop independent function call outside of the loop, and moved the loop independent array access outside of the loop. This loop is now much more efficient.

```plaintext
stop = (s+2);
j = cos(\phi);
k = array_element[6];
for (I = 0; I < stop; I++) {
    new_data[I] = (j+k)*I;
}
```

Note that we could further improve this loop by replacing the (j+k) with another temporary variable, but a good optimizer should do this for us. This leads to our next topic: compiler optimization options.

Many compilers offer different levels of optimization controlled by command line parameters. However, we must be somewhat careful with compiler optimization parameter selection as in some cases (programs) increasing the level of compiler optimization may actually decrease performance. For gcc/g++ the O2 optimization generally produces code more efficient than the O or O1 optimizations. However, the O3 optimization may or may not produce code more efficient than the O2 optimization, the same holds true for the O4 and O5 optimizations. Another set of optimizations to
consider are those that specify a specific CPU or machine architecture such as the -mcpu or the stronger –march gcc/g++ options. Regardless of the compiler and optimization options available, one should really conduct a trial and error study to see what works best for the intended platform and the specific program.

6. Conclusion.

I have tried to summarize the steps required to develop parallel algorithms for use on a HPCC such as Shale. Obviously, I left out many details. However, what is provided should empower the reader with enough information to get something running on Shale without too much effort. Even a poorly optimized parallel process is usually more effective than a serial process.

I have provided three C/C++ examples (a serial implementation, a PVM Master-Slave implementation, and MPI Coordinating Peers implementation) that perform an N-body simulation. My hopes are that given the straightforward serial implementation and the two parallel implementations, the reader may be able to understand the process of partitioning the problem into a parallel one and how to code it.

I have provided a utility (CRCRAM) that allows the user to process many data files in parallel using an existing (serial) program. I tried to design the source code and include sufficient comments (in the source code) such that anyone familiar with C/C++ would be able to modify the code to fit their requirements. Note that any sections of code that I expect would require modifications by any user of this example are clearly indicated in the comments. Note that the code is NOT the most efficient, as it is intended to educate, not impress.

Should you have questions, comments, need help applying an example, or need help with CRCRAM contact Ron Marsh at: rmarsh@cs.und.edu. Time permitting, I will try to help.
Chapter 2: CRCRAM

CRCRAM was designed for the embarrassingly parallel situation. Specifically, a situation where you have many independent data files (files that can be processed independently of each other) and where you have an existing program to process those individual data files.

Using parameters specified in the CRCRAM.ini file (see #2 below), CRCRAM scans the directory containing the data, builds a list of data files to process, and automatically load balances the processing across the nodes requested. As each data file is processed the resulting output is saved to a new file whose name is derived from the input file name with a second extension appended on (the idea is to make it easy to track which result came from which input). Therefore, if your input file is named test1.data, the resulting output file will be named test1.data.something (where the “something” is specified in your CRCRAM.ini file).

In the remainder of this document, I will work through an example showing how to set up and use CRCRAM:

1. Changes to .bash_profile:

   The command language interpreter is the interface between the user and the operating system. The command language interpreter executes commands read from standard input or from a file and sets paths to commonly used utilities and libraries. Since most accounts on Shale have Bash set as their default command language interpreter, I will use Bash in the example. As with most command language interpreters, Bash gets user configurable parameters from a .profile (pronounced “dot profile”) file. However, with Bash it’s a .bash_profile (pronounced “dot bash profile”) file.

   Note that the .bash_profile file has a period (the “dot”) for its first character which makes it invisible to “ls” (use “ls –la” to see it). The intent of the leading “.:.,” is to hide the file from normal use.

   Since this example uses PVM, you must change your .bash_profile file to specify paths required by PVM. To view or edit your .bash_profile file:

   1. Login (remain in your “home” directory).
   2. Use an editor to edit .bash_profile (ie vi .bash_profile).

   Below is an example .bash_profile file (yours is probably very similar) modified to include the paths required by PVM (note the text in bold). You should add the same text to your .bash_profile file.

   ```bash
   # .bash_profile
   # Get the aliases and functions
   if [ -f ~/.bashrc ]; then
   ```
. ~/.bashrc
fi

# User specific environment and startup programs
PATH=$PATH:$HOME/bin
export PATH

# PVM specifics
PVM_ROOT=/usr/share/pvm3
PVM_DPATH=/usr/share/pvm3/lib/pvmd
PVM_RSH=/usr/bin/ssh
export PVM_ROOT PVM_DPATH PVM_RSH

unset USERNAME

Once you have made the changes, logout and log back in to have the changes take effect.

2. The CRCRAM.ini file:

Since CRCRAM obtains all of its parameters via an initialization file called CRCRAM.ini, you must create such a file. An example CRCRAM.ini file is shown below:

```
In_extension_: .data
Out_extension: .output
Data_path___: /home/rmarsh/CRCRAM/BIN
Slave_path___: /home/rmarsh/CRCRAM/BIN/CRCRAMSlave
#
#Data_process_script:
#
sort -g $1 > $2
```

The CRCRAM.ini file **must** specify 5 parameters:
1. The “In_extension” specifies what the extension will be for the input data files (.data in this example).
2. The “Out_extension” specifies the extension to be appended on to the input filename for the resulting output files (.output in this example).
3. The “Data_path” specifies the directory where the data is located and where the results will be located (/home/rmarsh/CRCRAM/BIN in this example).
4. The “Slave_path” specifies the directory where the CRCRAMSlave program is located (/home/rmarsh/CRCRAM/BIN/CRCRAMSlave in this example - note that CRCRAMSlave is the actual name of the slave program and not a directory name). **The “Data_path” and “Slave_path” must be the same** (ie. CRCRAM and CRCRAMSlave must be in the same directory as the data). P.S. This is a “feature” not a “bug”.
5. The “Data_process_script” specifies the program to process the data files. In this example the script is a LINUX utility (sort) that takes an input file (indicated by the $1 variable), sorts the file, and writes the result to an output file (indicated by the $2 variable). CRCRAM populates the $1 and $2 variables with actual file
names as it processes the data. The "Data_process_script" must specify an input and output using $1 and $2.

Therefore, given the above CRCRAM.ini file, if you have an input file named test1.data, the resulting output file will be named test1.data.output.

Another example CRCRAM.ini file is shown below:

```
In_extension_: .pov
Out_extension: .png
Data_path___: /home/rmarsh/MOVIE/BIN
Slave_path___: /home/rmarsh/MOVIE/BIN/CRCRAMSlave
#
#Data_process_script:
#/usr/local/bin/povray +L/usr/local/share/povray-3.6/include +A0.3 +W640 +H480 +I$1 +FN -D +O$2
```

In this example the “Data_process_script” refers to an open source raytracing package (POVRAY) that takes a .pov scene description file as input and generates a .png image file as output (in actuality, when using CRCRAM we will get .pov.png image files as output). Note the $1 and $2 (in bold) in the script. The rest of the parameters in the “Data_process_script” set parameters specific to the POVRAY program (CRCRAM allows you to pass flags to the program).

A simple way to check the validity of your “Data_process_script” is to replace the $1 and $2 with valid filenames and run it from the LINUX prompt. If it works there it should work via CRCAM as well.

Note that the format of the CRCRAM.ini file is very specific. Do NOT change the ordering of parameters or any text to the left of the colons!

3. The batch submission script:

I will assume that you will use the batch scheduler on Shale to submit your job and have provided a simple script to configure the scheduler to use CRCRAM:

```
#!/bin/bash
#PBS -S /bin/bash
#PBS -N test
#PBS -q long
#PBS -d /home/rmarsh/CRCRAM/BIN/
#PBS -o test.o
#PBS -e test.e
#PBS –m abe
#PBS -l nodes=8:ppn=2
#
./CRCRAM –v
```

In order, the items this script configures are:
1. “#PBS –S” Sets the execution shell to bash (can leave as is).
2. “#PBS –N” Sets the name of the job to “test” (you can change or leave as is).
3. “#PBS –q” Sends the job to the “long” queue (can leave as is).
4. “#PBS –d” Sets the working directory to “/home/rmarsh/CRCRAM/BIN” (you must change this to reflect your directory organization).
5. “#PBS –o” Writes the output to a file (in the working directory) called “test.o” (you can change or leave as is).
6. “#PBS –e” Writes errors to a file (in the working directory) called “test.e” (you can change or leave as is).
7. “#PBS –m” Instructs the scheduler to send email when the job begins execution (b), terminates (e), or is aborted by the scheduler (a). You can use any combination of “abe.” If you do not want email sent, remove this line (change as desired).
8. “#PBS –l” Sets limits on the job execution. This example specifies 8 nodes with 2 processes per node. Note that there are many options. Do a “Man pbs_resources” for more info (see text box below).
9. “./CRCRAM –v” Instructs the batch scheduler to run CRCRAM with the verbose option (-v) on (to run CRCRAM without the verbose option change ”CRCRAM –v” to “CRCRAM”). You must change this argument to specify your process.

In general, parallelizing a job across a number of nodes and processes per node is not as obvious as it may first seem. Fortunately, if you have a task that CRCRAM was designed to manage, you can probably specify as many nodes as you have data files (however, the heuristic is to specify, at most, half as many nodes as data files). Specifying the processes per node is not as obvious. Here’s another heuristic: If your processing requires mainly computations and little IO, specify 2 processes per node and reduce the number of nodes requested. If your processing involves a lot of IO, you may want to only specify 1 process per node.

Finally, remember that CRCRAM is a master-slave paradigm. Therefore, 1 entire node (regardless of the number of processes per node specified) will be allocated to the master. For example, in this example only 14 slave processes will be created even though 16 processes have been requested via “#PBS –l”

4. Running CRCRAM:

Once you have PVM configured, CRCRAM.ini configured, and a batch submission script configured you can execute CRCRAM using the command:

```
qsub ./name_of_batch_script
```

Note that when using the batch scheduler no output is displayed on the computer screen, but is written to an output file (see “#PBS –o” and “#PBS –e” above).

To summarize, modify your .bash_profile file to include PVM paths, copy CRCRAM and CRCRAMSlave to the directory where your data files are, create a CRCRAM.ini and
batch submission script applicable to your job, and submit your job. *CRCRAM* will then check the allocated nodes to ensure PVM is available, scan the directory creating the data files to process, parse the “Data_process_script” specified in the *CRCRAM.ini* and replace the $1 and $2 variables with actual data file names, and call the specified program to process the data (file by file). Since CRCRAM scans the directory creating a list of data files to process each time CRCRAM is executed, you can increase or decrease the number of data files for each run by adding or removing files from the directory without having to make any changes to *CRCRAM.ini* or the batch submission script. Similarly, since CRCRAM gets the list of nodes to execute on from the batch scheduler each time CRCRAM is executed, you can change the number of nodes or processes per node in the batch submission script and CRCRAM will adjust the load balancing accordingly.

Finally, the load balancing mechanism is master-slave and naïve. Since it does not have any information regarding the processing time of any data file, it may very well end up waiting for a single long-to-process file to be completed while all of the other files have long finished. An informed scheduler would process long-to-process files first.
Chapter 3: Serial Template #1 (serial)

This program was designed to illustrate a serial situation where you have to write your own code. The program is a simple C/C++ program that performs a 2D heat transfer simulation.

We will model the environment by a 9x9 element 2D array “M”. On the upper left side there is a source of constant heat, while at the lower right side there is a source of constant cold. This is shown in figure 1.

![Figure 1](image_url)

The algorithm we will use to simulate the heat transfer is a simple 3 element weighted neighborhood averaging, where each array element becomes the average of itself and its 2 neighbors depending on a weighting factor. Equation (1) depicts the algorithm.

\[
M'_{j,i} = \frac{1}{scale} \sum_{y=-1}^{1} \sum_{x=-1}^{1} M_{(j+y,j+x)} \cdot K_{y,x}
\]  

Equation 1 is a common convolution, where K is a 3x3 matrix (a kernel) that defines the weighting values and scale is a normalization term derived from K.

To facilitate the simulation of the constant heat and constant cold sources we will extend array “M” to 11x11 elements with the outer elements having fixed values. This allows a simple implementation of equation 1 as the edges (of the original 9x9 region of interest) are no longer a special case that must be dealt with separately. The constant heat source is colored red and the constant cold source is colored blue in figure 2.

Note that the algorithm is iterative. A single pass of the neighborhood averaging procedure will not produce correct results. We must apply the neighborhood averaging procedure repeatedly until convergence occurs. To keep things simple, we will define convergence as being when no significant changes occur from iteration to iteration in the total value (e.g. sum) of the 9x9 region of interest.
Finally, note that M’ and M are **NOT** the same array and that the actual algorithm is:

1. Initialize K
2. Initialize M
3. Perform equation (1) with i and j ranging from 1 to 9
4. Update M by coping M’ to M (do not copy halo elements)
5. Check for convergence
6. Repeat from step 3 until convergence occurs

1. **The serial.cpp file:**

   The *serial.cpp* file is the program.

   Obviously, the processing done in this example is trivial, but the intent is to show how the basic algorithm was implemented such that the reader has a starting point for parallelizing the algorithm.

2. **The serial.h file:**

   The *serial.h* file is used to define include paths required by C/C++, to define global variables that would be common to any program using this example, and to define User definable parameters that govern the problem size (see USER DEFINES and USER GLOBAL VARIABLES).

3. **Compiling code based on this example:**

   Like many UNIX systems, Shale has a “make” utility for compiling programs. The Makefile for the examples is shown below.

   ```
   C  =gcc
   CC  =g++
   OFLAG=-O2
   LIBP=-L/usr/lib64/ -L/usr/local/lib/
   LIBS=-lm
   INCP=
   ```
serial:
    $(CC) $@.cpp $(OFLAG) $(INCP) $(LIBP) $(LIBS) -o $@

However, “make” is a bit fussy and will not compile a program if a valid executable
version currently exists in the current directory (talk about lazy!). Therefore, you have
to remove the old executable each time before running “make”. However, if you are
typing efficient (i.e. lazy) and don’t want to type all those commands to get
something compiled, try this script (requires the above *Makefile*):

    rm serial
    make serial

Use an editor to type the above text into a file (I like to call mine “build”) and use
chmod to make the file executable (*chmod 755 build*). You can then compile the
serial program simply by entering “./build” and the UNIX prompt.

4. **The batch submission script:**

    Not required.

5. **Running the program:**

    You can execute the program using the command:

    ./serial

6. **Source code follows:**
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <unistd.h>

/* GLOBAL PARAMETERS & VARIABLES */
/* */
/* ********************************************** */
/* SYSTEM DEFINES */
/* */
/* !No changes should be made to these! */
/* */
/* ********************************************** */
define version 1.1
#define yes 1
#define no 0

/* USER DEFINES */
/* */
/* Define the dimensions of the data to be */
/* processed. Note that the actual data to be */
/* processed in this template is 9x9. However, */
/* it will make the algorithm much simpler if */
/* I add an outer layer of elements (halo */
/* nodes) to the data, thus I will increase the*/
/* problem size to 11x11. */
/* */
/* I would expect the user would want to */
/* modify this code to fit their application. */
/* ********************************************** */
define Height 11 // Height of problem (with halo nodes).
define Width 11 // Width of problem (with halo nodes).

/* USER GLOBAL VARIABLES */
/* */
/* Define global variables. To be consistent */
/* across templates, I will define a global */
/* array to hold the problem and the matrix */
/* (or kernel) that will be used to weight the */
/* elements during the neighborhood averaging.*/
/* I also need to declare a second array to */
/* hold intermediate results. */
/* */
/* Verbose is used to display log messages   */
/* Turn on display with (verbose = yes) or  */
/* turn off display with verbose = no).      */
/*                                           */
/* I would expect the user would want to     */
/* modify this code to fit their application. */
/***********************************************/
double scale;
double K[3][3];
double M[Height][Width];
double Mprime[Height][Width];
int verbose = yes;
### serial.cpp

* Template for a generic serial program.


---

**Include the header file which includes any required global variable definitions, any user global varible definitions, and any required functions.**

See the serial.h file for details.

---

```c
#include "serial.h"
```

---

**DATA PARTITION/GENERATION ROUTINES**

---

**The INITIALIZE_K routine initializes the 3x3 kernel matrix which is used to adjust the weight of any array element during the neighborhood average operation.**

**I would expect the user would want to modify this code to fit their application.**

```c
void INITIALIZE_K(void) {
    int i, j;

    // Set kernel weights.
    K[0][0] = 0.5;  K[0][1] = 1.0;  K[0][2] = 0.5;
    K[1][0] = 1.0;  K[1][1] = 1.0;  K[1][2] = 1.0;
    K[2][0] = 0.5;  K[2][1] = 1.0;  K[2][2] = 0.5;

    // Calculate scale parameter.
    scale = 0.0;
    for (j = 0; j < 3; j++) {
        for (i = 0; i < 3; i++) {
            scale += K[j][i];
        }
    }
}
```
/***********************************************/
void INITIALIZE_M(void) {
int i, j;
for (j = 0; j < Height; j++) {
  for (i = 0; i < Width; i++) {
    M[j][i] = 0.0;
  }
}

// Set location of constant heat source.
for (j = 1; j < 4; j++) {
  M[j][0] = 100.0;
}

// Set location of constant cold source.
for (j = Height-2; j > Height-5; j--) {
  M[j][Width-1] = -10.0;
}
}

/***********************************************************/
/* DATA PROCESSING ROUTINES                                */
/***********************************************************/
/***********************************************/
/* The PROCESS_M routine performs the          */
/* neighborhood averaging on the data using the*/
/* kernel to weight the individual data element*/
/* contributions. Note that this particular    */
/* algorithm is a simple convolution.          */
/*                                                */
/* I would expect the user would want to       */
/* modify this code to fit their application.  */
/***********************************************/
void PROCESS_M(void) {
int i, j;
int x, y;
double sum;
for (j = 1; j < (Height-1); j++) {
  for (i = 1; i < (Width-1); i++) {
    sum = 0.0;
    for (y = 0; y < 3; y++) {
      for (x = 0; x < 3; x++) {
        sum += M[j+y-1][i+x-1]*K[y][x] ;
      }
    }
    Mprime[j][i] = sum / scale;
  }
}

/***********************************************************/
/* The UPDATE_M routine copies the iteration's */
/* result (held in Mprime) back into M.          */
/*                                                */
/* I would expect the user would want to        */
/* modify this code to fit their application.  */
/***********************************************************/
/***********************************************/
void UPDATE_M(void) {

    int    i, j;
    for (j = 1; j < (Height-1); j++) {
        for (i = 1; i < (Width-1); i++) {
            M[j][i] = Mprime[j][i];
        }
    }
}

/***********************************************/
/* The SUM_M routine calculates the sum of the */
/* data.                                       */
/*                                             */
/* I would expect the user would want to       */
/* modify this code to fit their application. */
/***********************************************/
double SUM_M(void) {
    int    i, j;
    double sum;
    sum = 0.0;
    for (j = 1; j < (Height-1); j++) {
        for (i = 1; i < (Width-1); i++) {
            sum += M[j][i];
        }
    }
    return sum;
}

/***********************************************************/
/* GENERATION ROUTINES                                     */
/***********************************************************/
/***********************************************************/
/* LOAD_BALANCE                                */
/* This routine does the following:            */
/* 1. Call functions to inialize the data.     */
/* 2. Call the function to process each        */
/* iteration of the data.                     */
/* I would expect the user would want to       */
/* modify this code to fit their application. */
/***********************************************************/
void LOAD_BALANCE(void) {

    // Define required parameters.
    int    i, j;
    int    startTime, stopTime;
    int    iteration;
    double newSum, oldSum;
    double convergence;

    /***********************************************************/
    /* Define user specific parameters. */
    /* It is expected that the user will replace */
FILE  *log;

// Records the start time of the program.
startTime = time(NULL);

/*****************************/
// Initializes the user's data.  *
// It is expected that the user will replace  *
// this call with their own.  *
/*****************************/
INITIALIZE_K();
INITIALIZE_M();
oldSum = 0.0;

/*****************************/
// Here we start the data processing.  *
/*****************************/
if (verbose == yes) {
    printf("*************************************************
");
    printf("* MESSAGE - Processing data.                    *
");
    printf("*************************************************
");
}

// Initialize iteration counter.
iteration = 0;

// Start infinite loop for iterative data processing.
while (1) {

    /*****************************/
    // Call the routine to perform 1 iteration of  *
    // data processing.  *
    // It is expected that the user will replace  *
    // this call with their own.  *
    /*****************************/
    PROCESS_M();

    /*****************************/
    // Call the routine to update M with the  *
    // results contained in Mprime.  *
    // It is expected that the user will replace  *
    // this call with their own.  *
    /*****************************/
    UPDATE_M();

    /*****************************/
    // We will assume convergence occurs when no  *
    // more changes occur and we will determine  *
    // this by monitoring the total sum (energy)  *
    // of the environment.  *
    /*****************************/
newSum = SUM M();
convergence = fabs(newSum - oldSum);
if (verbose == yes) {
    printf("Iteration [%d] convergence: %lf.\n", iteration, convergence);
}
if (convergence < 0.00001 || iteration == 10) break;
oldSum = newSum;

// Update iteration counter.
iteration++;

// Write results to log file.
log = fopen("./log", "w");
for (j = 0; j < Height; j++) {
    for (i = 0; i < Width; i++) {
        fprintf(log, "%6.2lf ", M[j][i]);
    }
    fprintf(log, "\n");
}
fclose(log);

// Records the stopping time of the program.
stopTime  = time(NULL);

// Display the program's running time.
if (verbose == yes) {
    printf("Processing time: %d.\n", (stopTime - startTime));
}

int main(int argc, char **argv) {

    // Process data.
    LOAD_BALANCE();

    // Shut down.
    return 1;
}
Chapter 4: Master-Slave Template #1 (MST)

This program was designed to illustrate a loosely coupled parallel situation where you have to write your own code. The program is a simple C/C++ program that performs a 2D heat transfer simulation and is based on PVM and thus, you must modify your .bash_profile file to set the paths required by PVM. You must also modify the mst.cpp program to initialize, partition, and accumulate the data specific to your task, you must modify the mstSlave.cpp program to process your data, and you must modify the mst.h file to specify your problem size (otherwise, I do not think mst.h will require any changes).

As with the serial program, for this program, we will model the environment by a 9x9 element 2D array “M”. On the upper left side there is a source of constant heat, while at the lower right side there is a source of constant cold. This is shown in Figure 3.

![Figure 3.](image)

The algorithm we will use to simulate the heat transfer is the same simple 3 element weighted neighborhood averaging, where each array element becomes the average of itself and its 2 neighbors depending on a weighting factor. Equation (1) is shown again below.

\[
M'_{j,i} = \frac{1}{\text{scale}} \sum_{y=-1}^{1} \sum_{x=-1}^{1} M_{(j+y,j+x)} \times K_{y,x}
\]  

Equation 1 is a common convolution, where K is a 3x3 matrix (a kernel) that defines the weighting values and scale is a normalization term derived from K.

To facilitate the simulation of the constant heat and constant cold sources we will extend array “M” to 11x11 elements with the outer elements having fixed values.

The height and width of array “M” are defined in the file mst.h via the 2 lines:

```c
#define Height 11  // Height of problem (with halo nodes).
#define Width 11   // Width of problem (with halo nodes).
```
This allows a simple implementation of equation 1 as the edges (of the original 9x9 region of interest) are no longer a special case that must be dealt with separately. The constant heat source is colored red and the constant cold source is colored blue in Figure 4.

![Figure 4.](image)

Note that the algorithm is iterative. A single pass of the neighborhood averaging procedure will not produce correct results. We must apply the neighborhood averaging procedure repeatedly until convergence occurs. To keep things simple, we will define convergence as being when no significant changes occur from iteration to iteration in the total value (e.g. sum) of the 9x9 region of interest.

This problem is nearly embarrassingly parallel as there are many parallelizations possible. Furthermore, since we are using a master-slave approach, we do not need to consider the number of processors that may be available nor any increase in future problem sizes (this is a real strength of the master-slave approach). Thus, we will adopt the simplest partitioning scheme possible: a row-by-row approach. However, given the 3x3 neighborhood averaging process, we will in actuality partition the problem into overlapping 3 row segments as shown in figure 5.

```
The height and width of the partitions are defined in the file mst.h via the 2 lines:
#define cellHeight  3     // Height of partition (with halo rows).
#define cellWidth   11   // Width of partition (with halo nodes).
```

Note that the example allows you to define partitions of multiple rows as well. However, the partitioning scheme you use MUST uniformly fit the problem space.

Since we have partitioned the problem, we must modify equation (1) to fit the partitioning scheme. The algorithm is still the same basic 3 element weighted neighborhood averaging, where each array element becomes the average of itself and its 2 neighbors depending on a weighting factor, but we are performing the processing on each partition “R”, as shown in equation (2).
\[ R'_{j,i} = \frac{1}{scale} \sum_{y=1}^{1} \sum_{x=1}^{1} R_{(j+y,i+x)} * K_{y,x} \]  

Note that while we have 3 row partitions, only the center row (of each partition) is processed/updated as the first and third rows (of each partition) are only used to provide data for the neighborhood averaging (these first and third rows are often referred to as *halo* rows).

Referring to equation (1), M’ and M and R and R’ are **NOT** the same array and that the actual algorithm is:

1. Master initializes M
2. Master spawns slave processes
3. Slaves each initialize their own K
4. Master extracts an overlapping partition R
5. Master sends a slave the partition R
6. Repeat from step 4 until all slaves have a partition
7. Slave performs equation (2) on its R
8. Slaves update R by copying R’ to R (do not copy halo elements)
9. Slaves return R & master receives R
10. Repeat from step 4 until all partitions have been processed
11. Update M by coping M’ to M (do not copy halo elements)
12. Master checks for convergence
13. Repeat from step 4 until convergence occurs

I will describe the design of the example in the remainder of this chapter.

1. **Changes to .bash_profile:**

   The command language interpreter is the interface between the user and the operating system. The command language interpreter executes commands read from standard input or from a file and sets paths to commonly used utilities and libraries. Since most accounts on Shale have Bash set as their default command language interpreter, I will
use Bash in the example. As with most command language interpreters, Bash gets user configurable parameters from a `.profile` (pronounced “dot profile”) file. However, with Bash it’s a `.bash_profile` (pronounced “dot bash profile”) file.

Note that the `.bash_profile` file has a period (the “dot”) for its first character which makes it invisible to “ls” (use “ls –la” to see it). The intent of the leading “.” is to hide the file from normal use.

Since this example uses PVM, you must change your `.bash_profile` file to specify paths required by PVM. To view or edit your `.bash_profile` file:

1. Login (remain in your “home” directory).
2. Use an editor to edit `.bash_profile` (ie vi `.bash_profile`).

Below is an example `.bash_profile` file (yours is probably very similar) modified to include the paths required by PVM (note the text in bold). You should add the same text to your `.bash_profile` file.

```
# .bash_profile
# Get the aliases and functions
if [ -f ~/.bashrc ]; then
  . ~/.bashrc
fi

# User specific environment and startup programs
PATH=$PATH:$HOME/bin
export PATH

# PVM specifics
PVM_ROOT=/usr/share/pvm3
PVM_DPATH=/usr/share/pvm3/lib/pvmd
PVM_RSH=/usr/bin/ssh
export PVM_ROOT PVM_DPATH PVM_RSH

unset USERNAME
```

Once you have made the changes, logout and log back in to have the changes take effect.

2. **The mst.cpp file:**

The `mst.cpp` file is the example for the master program. Typically, this program is rather long as it usually is tasked with initializing the data, partitioning the data, load balancing the processing, and accumulating the results.

Obviously, the processing done in this example is trivial, but the intent is to show how to pass different data types, the structure of the master program (the load
balancing in particular), and to stress the need to **terminate** the slave processes when they are no longer required.

The master creates a 11x11 2D array of double values, initializes PVM, and calls PVM_LOAD_BALANCE to initialize the data (see the INITIALIZE_DATA routine), to partition the data row-by-row (see the PARTITION_DATA_BY_ROW routine), and to process the data. Processing is done by sending each row to a slave, acquiring the result (row) from each slave, and inserting that result (row) back into its proper location in the 2D array. This example also sends “bookkeeping” data back and forth between the master and slaves to aid in troubleshooting and management of the load balancing.

The heart of the algorithm is in the PVM_LOAD_BALANCE routine. The routine starts by spawning a single slave process on each node. Once each node has had a slave process spawned, if 2 jobs per node have been specified, the process repeats itself by spawning an additional slave process on each node. If more than 2 jobs per node have been specified, the process continually repeats itself until the number of slave processes per node matches the number of jobs per node requested. This interlaced spawning is done to reduce IO contention during slave spawning. Finally, if a slave process fails to spawn, that slave process is not used for any data processing.

The scheme I have employed for load balancing is simple but effective for many types of data processing tasks. The first stage of load balancing extracts a partition of data and sends the partition to a slave process on one of the nodes. This process repeats until all of the slave processes have been sent data. Like the spawning process, sending data to slave processes is done in an interlaced manner to reduce IO contention during slave execution.

At the second stage of load balancing, once all of the slave processes have been spawned and are working on the data sent, the routine enters an infinite while loop and performs a non-blocking wait (a poll) to check for returned results. As soon as any slave process returns its result, that slave process is immediately sent another data segment to process. This process repeats until all of the data has been sent for processing. **It is the user’s responsibility to define when all of the data has been sent for processing and to properly specify that point such that the infinite loop terminates!**

At the third stage of load balancing, all of the data has been sent for processing and the algorithm only has to wait until the remaining active slave processes return results (the routine compares the number of “sends” versus the number of “returns”). Again, I use a non-blocking wait (a poll) inside a while loop to check for returned data.

At the fourth stage, all of the slave processes are sent a termination command. I must stress the need to **terminate the slave processes** when they are no longer required to reduce resource consumption on the node.
The only constraint on the data processing is that all of the rows be processed before moving on to the next iteration. The specific ordering of row processing is not important. Thus, this problem is considered loosely coupled parallel.

3. The mst.h file:

The `mst.h` file is used to define include paths required by C/C++, to define global variables that would be common to any program using this example, to define PBS/PVM utility routines that would also be common to any program using this example, and to define user definable parameters that govern the problem size (see USER DEFINES and USER GLOBAL VARIABLES).

4. The mstSlave.cpp file:

The `mstSlave.cpp` file is the example for the slave program. Typically, this program is rather short as all it is usually intended to do is process the blocks of data sent by the master and return results back to the master.

In the example, the slave program starts and waits (a blocking wait) for a message to be received from the master. The message contains either data to be processed or a termination message. If it is data to be processed it will include some “bookkeeping” data (data I found useful for troubleshooting and monitoring the progress of the processing) and the data to be processed. The data is then processed and sent back to the master. If the message contains the termination string (“stop” in this example), the slave program terminates. Obviously, the processing done in this example is trivial, but the intent is to show how to pass different data types, the structure of the slave program, and to stress the need to terminate the slave process when it is no longer required.

5. Compiling code based on this example:

Like many UNIX systems, Shale has a “make” utility for compiling programs. The `Makefile` for the examples is shown below.

```bash
C   =gcc
CC  =g++
OFLAG=-O2
LIBP=-L/usr/lib64/ -L/usr/local/lib/
    -L/usr/share/pvm3/lib/LINUXX86_64
LIBS=-lpvm3 -lm
INCP=-I/usr/share/pvm3/include

mst:
    $(CC) $@.cpp $(OFLAG) $(INCP) $(LIBP) $(LIBS) -o $@

mstSlave:
```

36
$(CC) $@.cpp $(OFLAG) $(INCP) $(LIBP) $(LIBS) -o $@

However, “make” is a bit fussy and will not compile a program if a valid executable version currently exists in the current directory (talk about lazy!). Therefore, you have to remove the old executable each time before running “make”. However, if you are typing efficient (ie lazy) and don’t want to type all those commands to get something compiled, try this script (requires the above *Makefile*):

```
rm mst
make mst
```

```
rm mstSlave
make mstSlave
```

Use an editor to type the above text into a file (I like to call mine “build”) and use chmod to make the file executable (*chmod 755 build*). You can then compile the *mst* and *mstSlave* programs simply by entering “./build” and the UNIX prompt.

6. **The batch submission script:**

I will assume that you will use the batch scheduler on Shale to submit your job and have provided a simple script to configure the scheduler to use the example:

```
#!/bin/bash
#PBS -S /bin/bash
#PBS -N mst
#PBS -q long
#PBS -d /home/rmarsh/MST/BIN/
#PBS -o mst.o
#PBS -e mst.e
#PBS –m abe
#PBS -l nodes=4:ppn=1
./mst
```

In order, the items this script configures are:

10. “#PBS –S” Sets the execution shell to bash (can leave as is).
11. “#PBS –N” Sets the name of the job to “mst” (you can change or leave as is).
12. “#PBS –q” Sends the job to the “long” queue (can leave as is).
13. “#PBS –d” Sets the working directory to “/home/rmarsh/MST/BIN” (**you must change this to reflect your directory organization**).
14. “#PBS –o” Writes the output to a file (in the working directory) called “mst.o” (you can change or leave as is).
15. “#PBS –e” Writes errors to a file (in the working directory) called “mst.e” (you can change or leave as is).
16. “#PBS –m” Instructs the scheduler to send email when the job begins execution (b), terminates (e), or is aborted by the scheduler (a). You can use any combination of “abe.” If you do not want email sent, remove this line (change as desired).
17. “#PBS –l” Sets limits on the job execution. This example specifies 4 nodes with 1 process per node. Note that there are many options. Do a “Man pbs_resources” for more info (see text box below).
18. “./mst” Instructs the batch scheduler to run the example program. **You must change this argument to specify your process.**

In general, parallelizing a job across a number of nodes and processes per node is not as obvious as it may first seem. Fortunately, if you have a task that this example was designed to manage, you can probably specify as many nodes as you have data files (however, the heuristic is to specify, at most, half as many nodes as data files). Specifying the processes per node is not as obvious. Here’s another heuristic: If your processing requires mainly computations and little IO, specify 2 processes per node and reduce the number of nodes requested. If your processing involves a lot of IO, you may want to only specify 1 process per node.

Finally, remember that this example implements a master-slave paradigm. Therefore, 1 entire node (regardless of the number of processes per node specified) will be allocated to the master. For example, in this example only 3 **slave** processes will be created even though 4 processes have been requested via “#PBS –l”

7. **Running the program:**

Once you have PVM configured and a batch submission script configured you can execute the derived program using the command:

```bash
qsub ./name_of_batch_script
```

Note that when using the batch scheduler no output is displayed on the computer screen, but is written to an output file (see “#PBS –o” and “#PBS –e” above).

To summarize, modify your `.bash_profile` file to include PVM paths, edit `mstSlave.cpp` and `mst.cpp` according to your task, compile `mstSlave.cpp` and `mst.cpp` using the `Makefile` (and optional “build” script) shown above, and then run the example derived program using the batch scheduler.

Since the program gets the list of nodes to execute on from the batch scheduler each time it is executed, you can change the number of nodes or processes per node in the batch submission script and any example derived program will adjust the load balancing accordingly.

Finally, the load balancing mechanism is master-slave and naïve. Since it does not have any information regarding the processing time of any partition, it may very well end up waiting for a single long-to-process partition to be completed while all of the other partitions have long finished. An informed scheduler would process long-to-process partitions first.
7. Source code follows:

```c
/****************************************************/
/* mst.h                                               */
/*                                                   */
/* Template for a generic PVM master/slave header file. */
/*                                                   */
/****************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <unistd.h>
#include "pvm3.h"

/****************************************************/
/* GLOBAL PARAMETERS & VARIABLES                   */
/****************************************************/
/** SYSTEM DEFINES                                 */
/* !No changes should be made to these!             */
/*                                                   */
/**                                                 */
#define version 1.5
#define yes 1
#define no 0
#define maxNodes 128
#define maxCPUs 256

/****************************************************/
/* USER DEFINES                                    */
/****************************************************/
/* Define the dimensions of the data to be processed. Note that the actual data to be processed in this template is 9x9. However, it will make the algorithm much simpler if I add an outer layer of elements (halo nodes) to the data, thus I will increase the problem size to 11x11. */
/* I would expect the user would want to modify this code to fit their application. */
#define Height 11 // Height of problem (with halo rows).
#define Width 11 // Width of problem (with halo columns).
#define cellHeight 3 // Height of partition (with halo rows).
#define cellWidth 11 // Width of partition (with halo columns).

/****************************************************/
/* Define the location of the user's slave          */
```

39
/* program. If this path is incorrect PVM will */
/* fail to start. */
/* */
/* I would expect the user would want to */
/* modify this code to fit their application. */
/***************************************
#define slavePath "/home/rmarsh/MST/BIN/mstSlave"

/***************************************
/* USER GLOBAL VARIABLES */
/* */
/* Verbose is used to display log messages */
/* Turn on display with (verbose = yes) or */
/* turn off display with verbose = no). */
/* */
/* I would expect the user would want to */
/* modify this code to fit their application. */
/***************************************
int verbose = yes;

/***************************************
/* SYSTEM GLOBAL VARIABLES */
/* */
/* !No changes should be made to these! */
/* */
/***************************************
int nodesWanted;
int jobsPerNode;
char nodeNames[maxNodes][256];
char nodeMaster[256];
int TID[maxCPUs];
char pvmCommand[256];

/***************************************
/* CLUSTER/PVM UTILITY ROUTINES */
/***************************************
/v
/* READ_MASTERNODEFILE */
/* This routine does the following: */
/* 1. Prompts the system for the host on which */
/* the master resides. */
/* */
/* !No changes should be made to this routine! */
/* */
/***************************************
void READ_MASTERNODEFILE(void) {
FILE *masterNodeFile;
if (verbose) {
    printf("*************************************************
");
    printf("* MESSAGE - Identifying master node. *\n");
    printf("*************************************************
");
}
    system("uname -n > masterNodeFile");
masterNodeFile = fopen("./masterNodeFile", "r");
if (masterNodeFile == NULL) {
    printf("ERROR - cannot identify master node.\n");
    exit(0);
}

fscanf(masterNodeFile, "%s", nodeMaster);
fclose(masterNodeFile);

/***********************************************/
/* READ_PBS_HOSTFILE                         */
/* This routine does the following:          */
/* 1. Copies the contents of the $PBS_NODEFILE */
/*    environment variable to a local file    */
/*    called "pbshostfile."                 */
/* 2. Parses the pbshostfile to extract the   */
/*    number of nodes and jobs per node wanted.*/
/* 3. Creates a PVM compatible hostfile.      */
/*                                             */
/* !No changes should be made to this routine!*/
/*                                             */
/* R. Marsh. July 2006. UND Computer Science.*/
/***********************************************/

void READ_PBS_HOSTFILE(void) {
    int i, cnt;
    int flag;
    char tempName[maxCPUs][256];
    FILE *pbshosts;
    FILE *pvmhosts;

    if (verbose == yes) {
        printf("*************************************************
");
        printf("* MESSAGE - Creating PVM hostfile.            *
");
        printf("*************************************************
");
    } // Copy PBS hostfile to PVM hostfile.
    // system("cat $PBS_NODEFILE > ./pbshostfile");

    // Read PBShostfile.
    pbshosts = fopen("./pbshostfile", "r");
    if (pbshosts == NULL) {
        printf("ERROR - Cannot read PBS nodefile [./pbshostfile].\n");
        exit(0);
    }
    cnt = 0;
    while (1) {
        fscanf(pbshosts, "%s", tempName[cnt]);
        if (feof(pbshosts)) break;
        cnt++;
    }
    fclose(pbshosts);

    // Extract number of nodes and processes per node.
    flag = 1;
    jobsPerNode = 1;
    nodesWanted = 0;
    strcpy(nodeNames[nodesWanted], tempName[0]);
    for (i = 1; i < cnt; i++) {
        if (strcmp(nodeNames[nodesWanted], tempName[i])) {

```
nodesWanted++;
    strcpy(nodeNames[nodesWanted], tempName[i]);
    flag = 0;
}  
if (flag == 1) jobsPerNode++;
}
nodesWanted++;

// Write PVMhostfile.
pvmhosts = fopen("./pvmhostfile", "w");
if (pvmhosts == NULL) {
    printf("ERROR - Cannot write PVM hostfile \[./pvmhostfile\].\n");
    exit(0);
}
for (i = 0; i < nodesWanted; i++) {
    fprintf(pvmhosts, "%s\n", nodeNames[i]);
}
fclose(pvmhosts);

/**********************************************
/* READ_PVM_HOSTFILE                        */
/* This routine does the following:          */
/* 1. Reads the node names from the PVM      */
/* compatible hostfile.                      */
/* 2. Checks to ensure that the master is NOT */
/* included in the list of slave nodes.      */
/*                                             */
/* !No changes should be made to this routine! */
/*                                             */
/***********************************************/
void READ_PVM_HOSTFILE(void) {
    int  j, k;
    char tempName[256];
    FILE *hosts;
    if (verbose == yes) {
        printf("********************************************\n");
        printf("* MESSAGE - Reading PVM hostfile.            *");
        printf("********************************************\n");
    }
    hosts = fopen("./pvmhostfile", "r");
    if (hosts == NULL) {
        printf("ERROR - cannot read './pvmhostfile'.\n");
        exit(0);
    }
    k = 0;
    for (j = 0; j < nodesWanted; j++) {
        fscanf(hosts, "%s\n", tempName);
        if (strcmp(nodeMaster, tempName)) {
            strcpy(nodeNames[k], tempName);
            k++;
        }
    }
    // Adjust the number of nodes to account for master.
    if (k < nodesWanted) nodesWanted = k;
    fclose(hosts);
}
void PVM_START(void) {
  int  block = yes;
  int  argc  = 1;
  char *argv[] = {"./pvmhostfile"};
  if (verbose == yes) {
    printf("*************************************************
");
    printf("* MESSAGE - Starting PVM deamons.               *
");
    printf("*************************************************
");
  }
  //   system("xterm -iconic -e pvm ./pvmhostfile &");
  //   system("pvm ./pvmhostfile &");
  //   sleep(5);
  pvm_start_pvmd(argc, argv, block);
}

void INITIALIZE_PVM(void) {
  READ_MASTERNODEFILE();
  READ_PBS_HOSTFILE();
  READ_PVM_HOSTFILE();
  PVM_START();
}

void SPAWN_PVM_NODES(void) {
  int nJobs = 0;
  for (int node = 0; node < nNodes; node++) {
    // Initialize array to hold slave process IDs
    // Initialize array to hold array of process IDs
    // Spawn slave process via PVM
    // Check for 2 (or more) jobs per node
    // If true, spawn second process
    // Reduce IO contention during spawning
  }
}
/** process. If slave fails to spawn, it is */
/** not used. */
/***************************/
void SPAWN_PVM_NODES(void) {
  int cc, j, n;
  int tid;

  // Initialize array to hold node IDs.
  for (j = 0; j < jobsPerNode; j++) {
    for (n = 0; n < nodesWanted; n++) {
      TID[j*nodesWanted+n] = -1;
    }
  }

  // Spawn slave processes.
  if (verbose == yes) {
    printf("*************************************************
    * MESSAGE - Spawning slave processes.               *
    *************************************************
    ");
    printf("MESSAGE - Slave spawned on node: %s\n",
      nodeNames[n]);
  } else {
    if (verbose == yes) {
      printf("ERROR - Slave failed on node: %s\n",
        nodeNames[n]);
    }
  }
}

/***************************/
/** SHUT_DOWN_PVM        */
/** This routine does the following:            */
/** 1. Shuts down PVM.                          */
/**                                             */
/** !No changes should be made to this routine! */
/**                                             */
/***************************/
void SHUT_DOWN_PVM(void) {
  if (verbose == yes) {
    printf("*************************************************
    * MESSAGE - Shutting down PVM deamons.       *
    *************************************************
    ");
    pvm_halt();
  }
}
/* Template for a generic PVM slave program. */

/**************************************************************************
/* Include the header file which includes the */
/* required global variables definitions and */
/* the required function definitions. */
/* See the mst.h file for details. */
/**************************************************************************
#include "mst.h"

/**************************************************************************
/* GLOBAL VARIABLES */
/**************************************************************************
/* Define global variables. To be consistent */
/* across templates, I will define a global */
/* array to hold the problem and the matrix */
/* (or kernel) that will be used to weight the */
/* elements during the neighborhood averaging. */
/* */
/* Since we have very defined roles as master */
/* or slave, some of the global declarations */
/* should only be here or only in the master. */
/* */
/* I would expect the user would want to */
/* modify this code to fit their application. */
double scale;
double K[3][3];
double R[cellHeight][cellWidth];

/**************************************************************************
/* DATA PARTITION/GENERATION ROUTINES */
/**************************************************************************
/* The INITIALIZE_K routine initializes the */
/* 3x3 kernel matrix which is used to adjust */
/* the weight of any array element during the */
/* neighborhood average operation. */
/* */
/* I would expect the user would want to */
/* modify this code to fit their application. */
/**************************************************************************
void INITIALIZE_K(void) {
    int i, j;

    // Set kernel weights.
K[0][0] = 0.5; K[0][1] = 1.0; K[0][2] = 0.5;
K[1][0] = 1.0; K[1][1] = 1.0; K[1][2] = 1.0;
K[2][0] = 0.5; K[2][1] = 1.0; K[2][2] = 0.5;

// Calculate scale parameter.
scale = 0.0;
for (j = 0; j < 3; j++) {
  for (i = 0; i < 3; i++) {
    scale += K[j][i];
  }
}

void PROCESS_R(void) {
  int    i, j;
  int    x, y;
  double sum;
  double Rprime[cellHeight][cellWidth];
  for (j = 1; j < (cellHeight-1); j++) {
    for (i = 1; i < (cellWidth-1); i++) {
      sum = 0.0;
      for (y = 0; y < 3; y++) {
        for (x = 0; x < 3; x++) {
          sum += R[j+y-1][i+x-1]*K[y][x];
        }
      }
      Rprime[j][i] = sum / scale;
    }
  }

  // Update R with results.
  for (j = 1; j < (cellHeight-1); j++) {
    for (i = 1; i < (cellWidth-1); i++) {
      R[j][i] = Rprime[j][i];
    }
  }
}

int main(void) {
  // DATA PROCESSING ROUTINES
  // The PROCESS_R routine performs the
  // neighborhood averaging on the data using the
  // kernel to weight the individual data element
  // contributions. Note that this particular
  // algorithm is a simple convolution.
  // I would expect the user would want to
  // modify this code to fit their application.

The following 5 parameters are either required (ptid) or that have proven useful for "bookkeeping" purposes. These are the same "bookkeeping" parameters as described in the mst.cpp Master Slave Template document.

```c
int ptid;  // Required.
int tid;   // For bookkeeping.
int nodeNumber; // For bookkeeping.
int dataNumber; // For bookkeeping.
char command[256]; // For control.
```

It is expected that the user will replace this call with their own.

```c
INITIALIZE_K();
```

This line returns the ID of the parent or master process (ptid). This information is required as several PVM jobs may be running (each with a unique master) and the slave needs to know which master to communicate with.

```c
ptid = pvm_parent();
```

Begin an infinite loop to acquire and process data as it arrives.

```c
while (1) {
```

Perform a blocking wait forcing the program to stop and wait for data from the master.

```c
pvm_recv(ptid, 1);
```

Unpack the "bookkeeping" data that was sent by the master. Note that this data is NOT used in any processing and is simply returned to the master with the processed data.

```c
pvm_upkint(&tid, 1, 1);
```
pvm_upkint(&nodeNumber, 1, 1);
pvm_upkint(&dataNumber, 1, 1);
pvm_upkstr(command);

/***********************************************/
/* Unpack the user's data that was sent by the */
/* master.                                     */
/*                                           */
/* The order of the data unpacking MUST match */
/* the order that it was packed by the master! */
/*                                           */
/* I would expect the user would want to      */
/* modify this code to fit their application. */
/***********************************************/
for (j = 0; j < cellHeight; j++) {
    pvm_upkdouble(R[j], cellWidth, 1);
}

/***********************************************/
/* These lines terminate the program if the   */
/* passed string "command" contains the text   */
/* "stop". This is a required section!        */
/* Otherwise, the slave process will never    */
/* terminate!                                 */
/***********************************************/
if (!strcmp(command, "stop")) {
    pvm_exit();
    exit(0);
}

/***********************************************/
/* Call the user's routine to process the data */
/* that was sent by the master.                */
/* I would expect the user would want to       */
/* modify this code to fit their application.  */
/***********************************************/
PROCESS_R();

/***********************************************/
/* Initiate communication to the master.      */
/***********************************************/
pvm_initsend(PvmDataRaw);

/***********************************************/
/* Pack the "bookkeeping" data into the       */
/* communication buffer.                      */
/***********************************************/
pvm_pkint(&tid, 1, 1);
pvm_pkint(&nodeNumber, 1, 1);
pvm_pkint(&dataNumber, 1, 1);

/***********************************************/
/* Pack the processed data into the           */
/* communication buffer.                      */
/***********************************************/
/* I would expect the user would want to      */
for (j = 1; j < (cellHeight-1); j++) {
    pvm_pkdouble(R[j], cellWidth, 1);
}

/* Return the data in the communication buffer */
/* to the master. */
pvm_send(ptid, 1);
}
return 1;
#include "mst.h"

double M[Height][Width];
double Mprime[Height][Width];
double R[cellHeight][cellWidth];

void INITIALIZE_M(void) {
    int i, j;
    for (j = 0; j < Height; j++) {
        for (i = 0; i < Width; i++) {
            // Initialize M[j][i]
        }
    }
}
M[j][i] = 0.0;
}
}

// Set location of constant heat source.
for (j = 1; j < 4; j++) {
    M[j][0] = 100.0;
}

// Set location of constant cold source.
for (j = Height-2; j > Height-5; j--) {
    M[j][Width-1] = -10.0;
}

/*******************************************************************************/
/* The PARTITION_M_BY_ROW routine partitions                                  */
/* user's data row-by-row.                                                   */
/*                                                                           */
/* I would expect the user would want to                                   */
/* modify this code to fit their application.                              */
/*******************************************************************************/
void PARTITION_M_BY_ROW(int segment) {
    int i, j;
    for (j = 0; j < cellHeight; j++) {
        for (i = 0; i < cellWidth; i++) {
            R[j][i] = M[segment+j][i];
        }
    }
}

/*******************************************************************************/
/* The ACCUMULATE_M_BY_ROW routine inserts the                             */
/* processed results into the array holding                                */
/* the intermediate results. The row ordering is maintained.              */
/* I would expect the user would want to                                   */
/* modify this code to fit their application.                              */
/*******************************************************************************/
void ACCUMULATE_M_BY_ROW(int segment) {
    int i, j;
    for (j = 1; j < (cellHeight-1); j++) {
        for (i = 0; i < cellWidth; i++) {
            Mprime[segment+j][i] = R[j][i];
        }
    }
}

/*******************************************************************************/
/* The UPDATE_M routine copies the iteration's                             */
/* result (held in Mprime) back into M.                                    */
/* I would expect the user would want to                                   */
/* modify this code to fit their application.                              */
/*******************************************************************************/
void UPDATE_M(void) {

int i, j;
for (j = 1; j < (Height-1); j++) {
    for (i = 1; i < (Width-1); i++) {
        M[j][i] = Mprime[j][i];
    }
}

/***********************************************/
/* The SUM_M routine calculates the sum of the */
/* data.                                       */
/*                                             */
/* I would expect the user would want to       */
/* modify this code to fit their application. */
/***********************************************/
double SUM_M(void) {
    int i, j;
    double sum;
    sum = 0.0;
    for (j = 1; j < (Height-1); j++) {
        for (i = 1; i < (Width-1); i++) {
            sum += M[j][i];
        }
    }
    return sum;
}

/***********************************************************/
/* PVM GENERATION ROUTINES                               */
/***********************************************************/
/***********************************************/
/* The PVM_SEND routine is used to send data _*/
/* to the slaves.                                     */
/*                                               */
/* We could embed this function into the load _*/
/* balancing routine below, but we would have _*/
/* to do so in several places. Thus, it is _*/
/* better software engineering practice to _*/
/* encapsulate its functionality into a single _*/
/* function.                                       */
/*                                               */
/* I would expect the user would want to       */
/* modify this code to fit their application. */
/***********************************************************/
void PVM_SEND(int tid, int nodeNumber, int dataNumber, char command[])
{
    int j;

    /***************************************************************************/
    /* Initiates communication to the slave. */
    /***************************************************************************/
pvm_initsend(PvmDataRaw);

    /***************************************************************************/
    /* Pack the "bookkeeping" parameters into the */
    /* communication buffer. */
    /***************************************************************************/
pvm_pkint(&tid, 1, 1);
pvm_pkint(&nodeNumber, 1, 1);
pvm_pkint(&dataNumber, 1, 1);
pvm_pkstr(command);

for (j = 0; j < cellHeight; j++) {
    pvm_pkdouble(R[j], cellWidth, 1);
}

pvm_send(tid, 1);

void PVM_RECEIVE(int *tid, int *nodeNumber, int *dataNumber) {
    int j;

    pvm_upkint(tid, 1, 1);
pvm_upkint(nodeNumber, 1, 1);
pvm_upkint(dataNumber, 1, 1);

    for (j = 1; j < (cellHeight-1); j++) {

pvm_upkdouble(R[j], cellWidth, 1);

} } 
/* PVM LOAD BALANCE */ /* This routine does the following: */ /* 1. Inializes the user's data. */ /* 2. Spawns the slave jobs on the nodes and */ /* sends each slave data to process. */ /* 3. Iteratively processes the data via: */ /* a) Nonblockingly waits for (polls) each */ /* slave to complete the processing of */ /* its current data and sends that slave */ /* another block of data to process. */ /* This continues untill all data has */ /* been sent. */ /* b) Nonblockingly waits for (polls) each */ /* slave to complete the processing of */ /* its current data. This continues */ /* until all sent data has been */ /* processed. */ /* c) Checks for convergence and stops the */ /* data processing if we have */ /* convergence. Otherwise, it repeats */ /* from "a"). */ /* 4. Sends each slave a termination signal. */ /* 5. Prints the results to a log file. */ /* I would expect the user would want to */ /* modify this code to fit their application. */ 
/* PVM LOAD BALANCE */

void PVM_LOAD_BALANCE(void) {

    // Define required parameters.
    int i, j, n;
    int tid;
    int pvmSent, pvmReceived;
    int startTime, stopTime;
    int dataCurrent, dataReturned;
    int iteration;
    double newSum, oldSum;
    double convergence;

    // Define user specific parameters.
    FILE *log;

    // Records the start time of the program.
    startTime = time(NULL);

    // Spawn slave nodes.
    strcpy(pvmCommand, "go");
SPAWN_PVM_NODES();

/******************************************************************************
/* Initializes the user's data. */
/* */
/* It is expected that the user will replace */
/* this call with their own. */
/******************************************************************************
INITIALIZE_M();
oldSum = 0.0;

/******************************************************************************
/* Here we start the data processing. */
/* */
/* The only limitation to the data processing */
/* is that all partitions must be processed */
/* BEFORE another iteration can begin and the */
/* only communication requirement is for the */
/* slaves to exchange data with the master. */
/* Thus, this example is considered "loosly */
/* coupled". */
/******************************************************************************
if (verbose == yes) {
    printf("*************************************************
");
    printf("* MESSAGE - Processing data.                    *
");
    printf("*************************************************
");
}

// Initialize iteration counter.
iteration = 0;

// Start infinite loop for iterative data processing.
while (1) {

    /******************************************************************************
    /* Initialize "bookkeeping" parameters. */
    /******************************************************************************
    pvmSent = 0;
pvmReceived = 0;
dataCurrent = 0;
dataReturned = 0;
strcpy(pvmCommand, "NA");

    /******************************************************************************
    /* Here we partition the user's data, and send */
    /* each process data to work on. If there are */
    /* 2 (or more) jobs per node specified, we */
    /* send data to one slave process on each node */
    /* first and then go back and send data to the */
    /* second process. We do this to reduce IO */
    /* contention during slave process executions. */
    /* */
    /* It is expected that the user will need to */
    /* change the PARTITION_DATA_BY_ROW call, and */
    /* the PVM_SEND call. It is also expected that */
    /* the user will need to specify their own */
    /* method of determining when to "break" (which*/

55
for (j = 0; j < jobsPerNode; j++) {
    for (n = 0; n < nodesWanted; n++) {
        // Partition user data.
        PARTITION_M_BY_ROW(dataCurrent);

        // Break if no more data to process.
        // P.S. This should not occur if you set the job up correctly.
        if (dataCurrent == (Height-2)) break;

        // Send data to first available slave.
        if (TID[j*nodesWanted+n] != -1) {
            PVM_SEND(TID[j*nodesWanted+n], n, dataCurrent, pvmCommand);
            pvmSent++;
            if (verbose == yes) {
                printf("Iteration [%d]: Package %4d > %s\n",
                        iteration, dataCurrent, nodeNames[n]);
            }
            dataCurrent+=(cellHeight-2);
        }
    }
}

/* At this point, all of the nodes have had */
/* data sent to them for processing and we */
/* wait for datum to be returned by a (any) */

/* slave. As soon as the master receives data */
/* from a/any slave, the master immediately */
/* sends another data partition to that slave. */
/* This process repeats until all of the data */
/* has been sent for processing (it is the */
/* user's task to define when this occurs - see*/
/* the if / break statement). Finally, I use a */
/* non-blocking wait (ie a poll) inside an */
/* infinite while loop to check for returned */
/* data. */
/* */
/* It is expected that the user will need to */
/* change the parameters in the PVM_RECEIVE */
/* call, the ACCUMULATE_DATA_BY_ROW call, the */
/* PARTITION_DATA_BY_ROW call, and the */
/* PVM_SEND call. It is also expected that the */
/* user will need to specify their own method */
/* of determining when to "break". */

while (1) {
    if (pvm_nrecv(-1,-1) > 0) {
        // Get data from slave.
        PVM_RECEIVE(&tid, &n, &dataReturned);
// Accumulate user data.
ACCUMULATE_M_BY_ROW(dataReturned);
pvmReceived++;
if (verbose == yes) {
    printf("Iteration [%d]: Package %4d < %s\n", iteration,
    dataReturned, nodeNames[n]);
}

// Partition user data.
PARTITION_M_BY_ROW(dataCurrent);

// Break if no more data to process.
// Very Important line !!!
// I Repeat. This is a very Important line !!!
// Don't forget this very Important line !!!
// Get the hint?
if (dataCurrent == (Height-2)) break;

// Send data to slave.
PVM_SEND(tid, n, dataCurrent, pvmCommand);
pvmSent++;
if (verbose == yes) {
    printf("Iteration [%d]: Package %4d > %s\n", iteration,
    dataCurrent, nodeNames[n]);
}
dataCurrent+=(cellHeight-2);
}

/***********************************************/
/* At this point, since all of the data has    */
/* been sent for processing, all we have left  */
/* to do is loop until the remaining data gets */
/* returned by the slaves. I use a non-blocking*/
/* wait (ie a poll) to check for returned      */
/* data.                                     */
/* It is expected that the user will need to */
/* change the PVM_RECEIVE call and the       */
/* ACCUMULATE_DATA_BY_ROW call.              */
/***********************************************/
do {
    if (pvm_nrecv(-1,-1) > 0) {

        // Get data from slave.
PVM_RECEIVE(&tid, &n, &dataReturned);

        // Accumulate user data.
        ACCUMULATE_M_BY_ROW(dataReturned);
pvmReceived++;
        if (verbose == yes) {
            printf("Iteration [%d]: Package %4d < %s\n", iteration,
            dataReturned, nodeNames[n]);
        }
    }
} while (pvmReceived < pvmSent);

57
Call the routine to update M with the results contained in Mprime.

It is expected that the user will replace this call with their own.

We will assume convergence occurs when no more changes occur and we will determine this by monitoring the total sum (energy) of the environment.

We have an infinite loop controlling the processing, therefore we MUST have some mechanism to terminate the loop!!!

It is expected that the user will replace this code with their own.

newSum = SUM M();
convergence = fabs(newSum - oldSum);
if (verbose == yes) {
    printf("Iteration [%d] convergence: %lf.\n", iteration, convergence);
}
if (convergence < 0.00001 || iteration == 10) break;
oldSum = newSum;

// Update iteration counter.
iteration++;

// Log results (with halo data) to file.
log = fopen("./log", "w");
for (j = 0; j < Height; j++) {
    for (i = 0; i < Width; i++) {
        fprintf(log, "%6.2lf ", M[j][i]);
    }
    fprintf(log, "\n");
}
fclose(log);

strcpy(pvmCommand, "stop");
for (j = 0; j < jobsPerNode; j++) {
    for (n = 0; n < nodesWanted; n++) {
        if (TID[j*nodesWanted+n] != -1) {
            // Send termination command to slave node.
            PVM_SEND(pvmCommand, n);
        }
    }
}
PVM_SEND(TID[j*nodesWanted+n], n, n, pvmCommand);
}
}
}

// Records the stopping time of the program.
stopTime = time(NULL);

// Display the program's running time.
if (verbose == yes) {
    printf("Processing time: %d\n", (stopTime - startTime));
}

/***********************************************************/
/* MAIN.                                                   */
/***********************************************************/
/***********************************************/
/* The MAIN routine initializes the system and */
/* starts the processing. */
/* */
/* There is nothing for the user to to change */
/* in this routine. */
/***********************************************/
int main(int argc, char **argv) {
    // Initialize PVM.
    INITIALIZE_PVM();

    // Process data.
    PVM_LOAD_BALANCE();

    // Shut down.
    SHUT_DOWN_PVM();
    return 1;
}
Chapter 5: Coordinating Peers Template #1 (PTP)

This program was designed to illustrate a tightly coupled parallel situation where you have to write your own code. The program is a simple C/C++ program that performs a 2D heat transfer simulation and is based on MPI. You must modify the ptp.cpp program to initialize, partition, and accumulate the data specific to your task, and you must modify the ptp.h file to specify your problem size (otherwise, I do not think ptp.h will require any changes).

As with the previous examples, for this program, we will model the environment by a 9x9 element 2D array “M”. On the upper left side there is a source of constant heat, while at the lower right side there is a source of constant cold. This is shown in figure 6.

![Figure 6.](image)

The algorithm we will use to simulate the heat transfer is a simple 3 element weighted neighborhood averaging, where each array element becomes the average of itself and its 2 neighbors depending on a weighting factor. Equation (1) is shown again below.

\[
M'_{j,i} = \frac{1}{scale} \sum_{y=-1}^{1} \sum_{x=-1}^{1} M_{(j+y,j+x)} \times K_{y,x}
\]  

Equation 1 is a common convolution, where K is a 3x3 matrix (a kernel) that defines the weighting values and scale is a normalization term derived from K.

To facilitate the simulation of the constant heat and constant cold sources we will extend array “M” to 11x11 elements with the outer elements having fixed values. This allows a simple implementation of equation 1 as the edges (of the original 9x9 region of interest) are no longer a special case that must be dealt with separately. The constant heat source is colored red and the constant cold source is colored blue in figure 7.

![Figure 7.](image)

Note that the algorithm is iterative. A single pass of the neighborhood averaging procedure will not produce correct results. We must apply the neighborhood averaging procedure repeatedly until convergence occurs. To keep things simple, we will define convergence as being when no significant changes occur from iteration to iteration in the total value (e.g. sum) of the 9x9 region of interest.
This problem is nearly embarrassingly parallel as there are many parallelizations possible. However, since we are using a coordinating peers approach, we do need to consider the number of processors that will be available and any increase in future problem sizes. Regardless, we will adopt the simplest partitioning scheme possible and divide the problem space into 3 equal sized overlapping partitions as shown in figure 8.

The height and width of the partitions are defined in the file ptp.h via the 2 lines:

```
#define cellHeight  5     // Height of partition (with halo rows).
#define cellWidth   11   // Width of partition (with halo nodes).
```

Note that the example allows you to define partitions of any size, detects the number of processors assigned (see the batch submission script section below), and creates the same number of partitions as processors assigned. Therefore, the user can create a custom partitioning scheme. However, the partitioning scheme you use MUST uniformly fit the problem space.

Since we have partitioned the problem, we must modify equation (1) to fit the partitioning scheme. The algorithm is still the same basic 3 element weighted neighborhood averaging, where each array element becomes the average of itself and its
2 neighbors depending on a weighting factor, but we are performing the processing on each partition “R”, as shown again in equation (2).

\[
R'_{j,i} = \frac{1}{scale} \sum_{y=1}^{1} \sum_{x=1}^{1} R_{(j+y,i+x)} \times K_{j,x}
\]

(2)

Note that only the center 3 rows (of each partition) are processed/updated as the first and fifth rows (of each partition) are only used to provide data for the neighborhood averaging (again, these first and fifth rows are often referred to as halo rows). Note also, that since we do not have a master to coordinate the problem initialization and partitioning, each process must be designed with its own partition and must also initialize its own partition. Therefore, in effect, we have 3 separate problems. A more descriptive view of the problem is shown in figure 9 below (note that the halo rows are colored gray).

![Figure 9](image_url)

Referring to equation (1), M’ and M and R and R’ are NOT the same array and that the actual algorithm is:

1. Each node initializes its own K
2. Each node initializes its own R
3. Nodes exchange halo data (see figure 10 below)
4. Each node performs equation (2) on its R
5. Each node updates R by copying R’ to R (do not copy halo elements)
6. Each node calculates its local sum
7. Node 0 requests the sum of all sums be obtained (an MPI call)
8. Node 0 for checks for overall convergence
9. Repeat from step 3 until convergence occurs
Note that it is common (required?) by a program using the coordinating peers paradigm to still have a controlling node (e.g. a “master” or “root” node). Typically, it is the “root” node’s responsibility to handle terminal IO, to determine when to stop, and possibly to accumulate the final results. In our example, node 0 is the “root” node.

I will describe the design of the example in the remainder of this chapter.

![Diagram of data partitioning](image)

### Figure 10.

1. **The ptp.cpp file:**

   The `ptp.cpp` file is the only program in the example as all nodes perform the same task(s).

   Obviously, the processing done in this example is trivial, but the intent is to show how to pass different data types, the structure of a coordinating peer program (the load balancing in particular), and to stress the need to **terminate** each peer process when they are no longer required.

   The program creates a 5x11 2D array of double values, initializes MPI, and calls `MPI_LOAD_BALANCE` to initialize the data (see the `INITIALIZE_DATA_PARTITION_BY_ROW` routine), and to process the data.

   The processing starts by having each node exchange halo data with each other node. Note that the data exchange using `MPI_Send()` and `MPI_Recv()` also acts to synchronize the peers as each must wait for the sent data to be received. The routine then calls the `PROCESS_DATA` routine which processes the local partition. `MPI_Barrier()` (following the `PROCESS_DATA` call) is used to synchronize the peers again. Forcing each peer to reach this point in the algorithm and wait until all other peers have also reached this point. Once synchronization occurs, each node calls...
SUM_DATA to obtain the sum of their local partition. The following MPI_Reduce() is used to accumulate/calculate a sum of the local sums and to place that value in node 0 (our “root” node). The root node then determines if convergence has been reached. If convergence has been reached, the root node copies the string “stop” into the string variable “mpiCommand” (which was initialized with the string “go”) and calls MPI_Bcast() to broadcast the string variable “mpiCommand” to all nodes. As with the master-slave example, the broadcasting of the string variable “mpiCommand” to all nodes is akin to sending all slave processes a termination command. I must stress the need to terminate the peer processes when they are no longer required to reduce resource consumption on the node.

One of the unique characteristics of a coordinating-peers system is that all of the peers are just that, peers. All equal and identical. Fortunately, MPI assign IDs (stored in the variable myID in the example) to each peer and we can use that information to force a uniqueness on the peers. If you examine the file ptp.cpp you will see that I use myID in several places to assign particular functionality to specific peers. I also use myID to create a spatial relationship between the peers in order to reduce the communication requirements during the exchange of halo data.

The only constraint on the data processing is that all of the partitions are processed before exchanging halo data and moving on to the next iteration. This constraint is only slightly more restrictive than the constraint present in the master-slave example. However, due to the increase in communication required by the halo data exchanges, this problem is considered tightly coupled parallel.

2. The ptp.h file:

The ptp.h file is used to define include paths required by C/C++, to define global variables that would be common to any program using this example, to define MPI utility routines that would also be common to any program using this example, and to define user definable parameters that govern the problem size (see USER DEFINES and USER GLOBAL VARIABLES).

3. Compiling code based on this example:

Like many UNIX systems, Shale has a “make” utility for compiling programs. The Makefile for the example is shown below.

```bash
CC = /usr/local/mpich/1.2.5.2/pgi/x86_64/bin/mpiCC
OFLAG=-O3
LIBP=-L/usr/lib64/ -L/usr/local/lib/ -L/usr/local/mpich/1.2.5.2/pgi/x86_64/lib
LIBS=-lmpipe
INCP=-I/usr/local/mpich/1.2.5.2/pgi/x86_64/include

ptp:
```
$(CC) $@.cpp $(OFLAG) $(INCP) $(LIBP) $(LIBS) -o $@

However, “make” is a bit fussy and will not compile a program if a valid executable version currently exists in the current directory (talk about lazy!). Therefore, you have to remove the old executable each time before running “make”. However, if you are typing efficient (i.e. lazy) and don’t want to type all those commands to get something compiled, try this script (requires the above Makefile):

```
rm ptp
make ptp
```

Use an editor to type the above text into a file (I like to call mine “build”) and use chmod to make the file executable (`chmod 755 build`). You can then compile the example program simply by entering “./build” and the UNIX prompt.

4. The batch submission script:

I will assume that you will use the batch scheduler on Shale to submit your job and have provided a simple script to configure the scheduler to use the example:

```
#!/bin/bash
#PBS -S /bin/bash
#PBS -N ptp
#PBS -q long
#PBS -d /home/rmarsh/PTP/BIN/
#PBS -o ptp.o
#PBS -e ptp.e
#PBS -m abe
#PBS -l nodes=3:ppn=1

/usr/local/mpich/1.2.5.2/pgi/x86_64/bin/mpirun -np 3 ./ptp
```

In order, the items this script configures are:
19. “#PBS –S” Sets the execution shell to bash (can leave as is).
20. “#PBS –N” Sets the name of the job to “ptp” (you can change or leave as is).
21. “#PBS –q” Sends the job to the “long” queue (can leave as is).
22. “#PBS –d” Sets the working directory to “/home/rmarsh/PTP/BIN” (you must change this to reflect your directory organization).
23. “#PBS –o” Writes the output to a file (in the working directory) called “ptp.o” (you can change or leave as is).
24. “#PBS –e” Writes errors to a file (in the working directory) called “ptp.e” (you can change or leave as is).
25. “#PBS –m” Instructs the scheduler to send email when the job begins execution (b), terminates (e), or is aborted by the scheduler (a). You can use any combination of “abe.” If you do not want email sent, remove this line (change as desired).
26. “#PBS –l” Sets limits on the job execution. This example specifies 3 nodes with 1 process per node. Note that there are many options. Do a “Man pbs_resources” for more info (see text box below).
27. “/usr/local/mpich/1.2.5.2/pgi/x86_64/bin/mpirun -np 3 ./ptp” instructs the batch scheduler to run the example program via mpi using 3 processes (-np 3). **You must change this argument to specify your process and you must ensure that the number of processes specified here (-np xx) matches the total number (nodes * ppn) set with “#PBS –l”**

In general, parallelizing a job across a number of nodes and processes per node is not as obvious as it may first seem. If you have a task that this example was designed to manage you would specify as many nodes as you have data partitions. However, specifying the processes per node is still not obvious. Here’s another heuristic: If your processing requires mainly computations and little IO, specify 2 processes per node (and reduce the number of nodes accordingly). If your processing involves a lot of IO, you may want to only specify 1 process per node.

5. **Running the program:**

Once you have a batch submission script configured you can execute the derived program using the command:

```
qsub ./name_of_batch_script
```

Note that when using the batch scheduler no output is displayed on the computer screen, but is written to an output file (see “#PBS –o” and “#PBS –e” above).

To summarize, edit `ptp.cpp` according to your task, compile `ptp.cpp` using the `Makefile` (and optional “build” script) shown above, and then run the program using the batch scheduler.

Even though this example gets the list of nodes to execute on from the batch scheduler each time it is executed, **you cannot change the total number of nodes** as the example was specifically designed for 3 nodes regardless of the problem size. You can, however, change the number of processes per node as long as the total number of nodes remains the same. Obviously, this is very different from the master-slave example that had no restrictions as to changing the problem size or number of slave processes. It’s not that a coordinating peers program could not be designed to be as flexible as a master-slave program (with regards to run-time changes). It can. But the example would become much more complex.

Another issue is the partition scheme. The row-wise scheme used in this example is somewhat intuitive and easy to implement. Another intuitive partitioning scheme is to partition the 9x9 region into 9 3x3 subregions as shown in figure 11.
If we add halo rows (and columns) we have the situation shown in figure 12 (halo nodes colored gray).

We now have a partitioning scheme that allows the use of 9 nodes for processing. However, the peer-to-peer communication required to exchange the halo information is horrendous! For example, as figure 13 shows, the center partition requires halo information (2 rows, 2 columns, and 4 corners) from all of its 8 neighbors. Even the corner partitions, partitions that require the least halo information, require halo information (1 row, 1 column, and 1 corner) from 3 of their neighbors.
Sometimes it is best to start over.

Finally, the load balancing mechanism is coordinating peers and naïve. Since it does not have any information regarding the processing time of any partition, it may very well end up waiting for a single long-to-process partition to be completed while all of the other partitions have long finished. An informed scheduler (or researcher) would adjust the partition sizes to account for this difference.

8. **Source code follows:**
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <unistd.h>
#include "mpi.h"

/* USER DEFINES */
#define Height     11   // Height of problem (with halo nodes).
#define Width      11   // Width of problem (with halo nodes).
#define cellHeight 5    // Height of partition (with halo rows).
#define cellWidth  11   // Width of partition (with halo nodes).
I also need to declare a second array to hold intermediate results. Finally, I turn on (verbose = yes) or off (verbose = no) the display of log messages. Verbose is used to display log messages. Turn on display with (verbose = yes) or turn off display with verbose = no. I would expect the user would want to modify this code to fit their application.

```c
double scale;
double K[3][3];
double R[cellHeight][cellWidth];
int verbose = yes;
```

SYSTEM GLOBAL VARIABLES


```c
int nodes;
int myID;
```

CLUSTER/MPI UTILITY ROUTINES

INITIALIZE_MPI

This routine does the following:
1. Calls the routines in proper order to start MPI.
2. Returns the number of MPI nodes started.
3. Returns the rank (ID) of each MPI node.

!No changes should be made to this routine!


```c
void INITIALIZE_MPI(int argc, char **argv) {
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &nodes);
  MPI_Comm_rank(MPI_COMM_WORLD, &myID);
  if (myID == 0 && verbose == yes) {
    printf("*************************************************
    * MESSAGE - Starting MPI deamons.               *
    *************************************************
    ");
    printf("* MESSAGE - Starting MPI deamons.           *
    ");
    printf("*************************************************
    ");
  }
}
```

SHUT_DOWN_MPI

This routine does the following:

```c
```
/* 1. Shuts down MPI. */
/* */
/* !No changes should be made to this routine! */
/* */
/***************************************************************************/
void SHUT_DOWN_MPI(void) {
    if (myID == 0 && verbose == yes) {
        printf("*************************************************
");
        printf("* MESSAGE - Shutting down MPI daemons. *
");
        printf("*************************************************
");
    }
    MPI_Finalize();
}
/* DATA PARTITION/GENERATION ROUTINES */
/**********************************************************/
/* The INITIALIZE_K routine initializes the */
/* 3x3 kernel matrix which is used to adjust */
/* the weight of any array element during the */
/* neighborhood average operation. */
/* I would expect the user would want to */
/* modify this code to fit their application. */
/**********************************************************/
void INITIALIZE_K(void) {
    int i, j;
    // Set kernel weights.
    K[0][0] = 0.5; K[0][1] = 1.0; K[0][2] = 0.5;
    K[1][0] = 1.0; K[1][1] = 1.0; K[1][2] = 1.0;
    K[2][0] = 0.5; K[2][1] = 1.0; K[2][2] = 0.5;
    // Calculate scale parameter.
    scale = 0.0;
    for (j = 0; j < 3; j++) {
        for (i = 0; i < 3; i++) {
            scale += K[j][i];
        }
    }
}

/**********************************************************/
/* The INITIALIZE_R routine zeros the array */
/* containing each partition and assigns the */
/* location and values for the "hot" and "cold"*/
/* spots. Note that the "hot" and "cold" spots */
/* are positioned along the outer edges (eg. */
/* in the halo region) where they will not get*/
/* changed. */
/* I would expect the user would want to */
/* modify this code to fit their application. */
/*******************************************************/
void INITIALIZE_R(int row) {
  int i, j;
  for (j = 0; j < cellHeight; j++) {
    for (i = 0; i < cellWidth; i++) {
      R[j][i] = 0.0;
    }
  }

  // Set location of constant heat source.
  if (myID == 0) {
    for (j = 1; j < 4; j++) {
      R[j][0] = 100.0;
    }
  }

  // Set location of constant cold source.
  if (myID == (nodes-1)) {
    for (j = cellHeight-2; j > cellHeight-5; j--) {
      R[j][cellWidth-1] = -10.0;
    }
  }
}

/***********************************************************/
/* DATA PROCESSING ROUTINES                                */
/***********************************************************/
/*******************************************************/
/* The PROCESS_R routine performs the          */
/* neighborhood averaging on the partition     */
/* using the kernel to weight the individual   */
/* data element contributions. Note that this */
/* particular algorithm is a simple          */
/* convolution.                               */
/*                                                */
/* I would expect the user would want to       */
/* modify this code to fit their application. */
/*******************************************************/
void PROCESS_R(void) {
  int    i, j;
  int    x, y;
  double sum;
  double Rprime[cellHeight][cellWidth];

  // Convolve kernel with data to get neighborhood average.
  for (j = 1; j < (cellHeight-1); j++) {
    for (i = 1; i < (cellWidth-1); i++) {
      sum = 0.0;
      for (y = 0; y < 3; y++) {
        for (x = 0; x < 3; x++) {
          sum += R[j+y-1][i+x-1]*K[y][x];
        }
      }
      Rprime[j][i] = sum / scale;
    }
  }
}
// Update R with results.
for (j = 1; j < (cellHeight-1); j++) {
    for (i = 1; i < (cellWidth-1); i++) {
        R[j][i] = Rprime[j][i];
    }
}

/***********************************************/
/* The SUM_R routine calculates the sum of the */
/* partition.                                    */
/* I would expect the user would want to       */
/* modify this code to fit their application. */
/***********************************************/
double SUM_R(void) {
    int i, j;
    double sum;
    sum = 0.0;
    for (j = 1; j < (cellHeight-1); j++) {
        for (i = 1; i < (cellWidth-1); i++) {
            sum += R[j][i];
        }
    }
    return sum;
}

/***********************************************/
/* The GET_xxx_HALO_DATA routines extract halo */
/* node data from the local partition.        */
/* I would expect the user would want to       */
/* modify this code to fit their application. */
/***********************************************/
void GET_TOP_HALO_DATA(double top[cellWidth]) {
    int i;
    for (i = 0; i < cellWidth; i++) {
        top[i] = R[1][i];
    }
}

void GET_BOTTOM_HALO_DATA(double bottom[cellWidth]) {
    int i;
    for (i = 0; i < cellWidth; i++) {
        bottom[i] = R[cellHeight-2][i];
    }
}

/***********************************************/
/* The SET_xxx_HALO_DATA routines insert halo */
/* node data into the local partition.        */
/* I would expect the user would want to       */
/* modify this code to fit their application. */
/***********************************************/
void SET_TOP_HALO_DATA(double top[cellWidth]) {

```c
int i;
for (i = 0; i < cellWidth; i++) {
    R[0][i] = top[i];
}

void SET_BOTTOM_HALO_DATA(double bottom[cellWidth]) {
    int i;
    for (i = 0; i < cellWidth; i++) {
        R[cellHeight-1][i] = bottom[i];
    }
}

/**********************************************************/
/* MPI GENERATION ROUTINES                                 */
/**********************************************************/
/***************************************************************************/
/* MPI_LOAD_BALANCE                                                   */
/* This routine does the following:                                  */
/* 1. Initializes the user's data.                                  */
/* 2. We expect the user would want to modify this code to fit their */
/*    application.                                                   */
/***************************************************************************/
void MPI_LOAD_BALANCE(void) {
    // Define required parameters.
    char   mpiCommand[256];
    int    startTime, stopTime;
    int    iteration;
    double localSum, oldSum, newSum;
    double convergence;
    MPI_Status status;

    /***************************************************************************/
    /* Define user specific parameters.                                */
    /* It is expected that the user will replace these with their own    */
    /* variable declarations.                                          */
    /***************************************************************************/
    int    i, j, n;
    int    start, stop;
    double haloTop[cellWidth];
    double haloBottom[cellWidth];
    FILE   *log;

    // Records the start time of the program.
    startTime = time(NULL);

    /***************************************************************************/
    /* Initialize the kernel and local partitions.                     */
    /* It is expected that the user will replace this code with their   */
    /* own.                                                             */
    /***************************************************************************/
    INITIALIZE_K();
```
INITIALIZE_R(myID);
oldSum = 0.0;
strcpy(mpiCommand, "go");

/***********************************************************/
/* Here we start the data processing. Note in */
/* this template, we use an infinite loop to */
/* allow the processing to continue until we */
/* we obtain convergence. */
/***********************************************************/
if (myID == 0 && verbose == yes) {
    printf("************************************************");
    printf("* MESSAGE - Processing data. */
    printf("************************************************");
}

// Initialize iteration counter;
iteration = 0;

// Start infinite loop for iterative data processing.
while (1) {
    /*******************************************************************/
    /* At this point we need to exchange halo node */
    /* data but, due to the partitioning, this can */
    /* get messy: */
    /* Node 0 sends halo data to node 1 */
    /* Node 1 sends halo data to nodes 0 and 2 */
    /* Node 2 sends halo data to node 1 */
    /* */
    /* Note that nodes 0 and 2 are end points and */
    /* special cases. All other inbetween nodes */
    /* would be treated as node 1. */
    /* */
    /* Note that the send call (MPI_Send) and */
    /* receive call (MPI_Recv) forces each node to */
    /* stop and wait for the call completion - a */
    /* form of synchronization. */
    /* */
    /* It is expected that the user will replace */
    /* this call with their own. */
    /*******************************************************************/
    if (myID == 0) {
        GET_BOTTOM_HALO_DATA(haloBottom);
        MPI_Send(haloBottom, cellWidth, MPI_DOUBLE, (myID+1), myID,
                MPI_COMM_WORLD);
    } else if (myID == (nodes-1)) {
        GET_TOP_HALO_DATA(haloTop);
        MPI_Send(haloTop, cellWidth, MPI_DOUBLE, (myID-1), myID,
                MPI_COMM_WORLD);
    } else {
        GET_TOP_HALO_DATA(haloTop);
        MPI_Send(haloTop, cellWidth, MPI_DOUBLE, (myID-1), myID,
                MPI_COMM_WORLD);
        GET_BOTTOM_HALO_DATA(haloBottom);
        MPI_Send(haloBottom, cellWidth, MPI_DOUBLE, (myID+1), myID,
                MPI_COMM_WORLD);
    }
}
if (myID == 0) {
    MPI_Recv(haloTop, cellWidth, MPI_DOUBLE, (myID+1),
            MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    SET_BOTTOM_HALO_DATA(haloTop);
} else if (myID == (nodes-1)) {
    MPI_Recv(haloBottom, cellWidth, MPI_DOUBLE, (myID-1),
            MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    SET_TOP_HALO_DATA(haloBottom);
} else {
    MPI_Recv(haloBottom, cellWidth, MPI_DOUBLE, (myID-1),
            MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    SET_TOP_HALO_DATA(haloBottom);
    MPI_Recv(haloTop, cellWidth, MPI_DOUBLE, (myID+1),
            MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    SET_BOTTOM_HALO_DATA(haloTop);
}

/**********************************************
/* Here each node processes its own local data.*/
/* We use MPI_Barrier to force synchronization */
/* all nodes are forced to stop and wait until */
/* all others nodes reach this point also. */
/* */
/* It is expected that the user will replace */
/* this call with their own. */
/***********************************************/
PROCESS_R();
MPI_Barrier(MPI_COMM_WORLD);

/**********************************************
/* We will assume convergence occurs when no */
/* more changes occur and we will determine */
/* this by monitoring the total sum (energy) */
/* of the environment. */
/* */
/* First we calculate the sums local to each */
/* node. */
/* */
/* It is expected that the user will replace */
/* this call with their own. */
/***********************************************/
localSum = SUM_R();

/**********************************************
/* We now use MPI to acquire the sums local to */
/* the nodes and produce a single overall sum. */
/* */
/* Note that the reduce call (MPI_Reduce) */
/* forces ALL nodes to stop and send the data */
/* - a form of synchronization. */
/* */
/* It is expected that the user will replace */
/* this call with their own. */
/***********************************************/
MPI_Reduce(&localSum, &newSum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

/***********************************************/
/* Here we check for convergence to see if we */
/* are done processing. */
/* */
/* We have an infinite loop controlling the */
/* processing, therefore we MUST have some */
/* mechanism to terminate the programs. */
/* */
/* It is expected that the user will replace */
/* this code with their own. */
/***********************************************/
if (myID == 0) {
    convergence = fabs(newSum - oldSum);
    if (verbose == yes) {
        printf("Iteration [%d] convergence: %lf.\n", iteration, convergence);
    }
    if (convergence < 0.00001 || iteration == 10)
        strcpy(mpiCommand, "stop");
    oldSum = newSum;
}

// Send all nodes a command to break or continue the infinite loop.
MPI_Bcast(mpiCommand, 5, MPI_CHARACTER, 0, MPI_COMM_WORLD);
if (!strcmp(mpiCommand, "stop")) break;

// Update iteration counter.
iteration++;

/***********************************************/
/* At this point we will log the results to a */
/* file by having each node send its partition */
/* to node 0, which will then log the results. */
/* The MPI_Barrier is used to ensure that all */
/* nodes have sent their partition before node */
/* 0 writes the log file. */
/***********************************************/
MPI_Send(R, cellHeight*cellWidth, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);

// Log results (with halo data) to file.
if (myID == 0) {
    log = fopen("./log", "w");
    for (n = 0; n < nodes; n++) {
        MPI_Recv(R, cellHeight*cellWidth, MPI_DOUBLE, n, 0, MPI_COMM_WORLD, &status);
        if (n == 0) start = 0;
        else start = 1;
        if (n == nodes-1) stop = cellHeight;
        else stop = (cellHeight-1);
        for (j = start; j < stop; j++) {
            // Log data
        }
    }
    fclose(log);
}
for (i = 0; i < cellWidth; i++) {
    fprintf(log, "%6.2lf ", R[j][i]);
}
fprintf(log, "\n");
}
fclose(log);

// Records the stopping time of the program.
stopTime = time(NULL);

// Display the program's running time.
if (myID == 0 && verbose == yes) {
    printf("Processing time: %d.\n", (stopTime - startTime));
}
}
Chapter 6: Serial Template #2 (serial_2)

This program is another example of a serial situation where you have to write your own code and where we will read the problem’s initial values from a file. The program is a simple C/C++ program that performs an N-body simulation as shown in figure 14.

For this program, we will assume that we have 10 bodies, each having a 2D position coordinate and a 2D velocity vector. We will model the environment by a 10x4 element 2D array “DATA” where each row contains the data <x, y, v_x, v_y>. This is shown in figure 15.

The algorithm we will use to simulate the system is simple. Each body’s trajectory is deflected towards the center of mass of the system by an amount determined by the body’s distance from the center of mass of the system. Equation (1), (2), and (3) depict the algorithm.

\[
X_i' = X_i + Vx_i * t \quad \text{and} \quad Y_i' = Y_i + Vy_i * t
\]  

(1)

Where:
\[ Vx_i' = \left( Vx_i + \frac{1}{(X_i - \bar{X})^3} * t \right) \quad \text{and} \quad Vy_i' = \left( Vy_i + \frac{1}{(Y_i - \bar{Y})^3} * t \right) \] (2)

and

\[ \bar{X} = \frac{1}{10} \sum_{j=0}^{9} X_j \quad \text{and} \quad \bar{Y} = \frac{1}{10} \sum_{j=0}^{9} Y_j \] (3)

No halo elements are required for this problem, but the problem is still iterative. A single pass will not produce correct results. We must apply the procedure repeatedly until convergence occurs. To keep things simple, we will define convergence as being when no significant changes occur in the average position of the bodies from iteration to iteration. Finally, note that X’ and X, Y’ and Y, Vx’ and Vx, and Vy’ and Vy are NOT the same variables and that the actual algorithm is:

1. Read the values for DATA from the file “particles.data”
2. Find center of mass via equation (3)
3. Adjust body velocities via equation (2)
4. Adjust body positions via equation (1)
5. Update the array DATA by coping X’ to X, Y’ to Y, Vx’ to Vx, and Vy’ to Vy
6. Check for convergence
7. Repeat from step 2 until convergence occurs

The major difference between the “serial” program and this program is that this problem is not driven by any neighborhood relationships. ALL of the data must be used in all iterations and this characteristic will impact our parallelization.

9. The serial_2.cpp file:

The serial.cpp file is the program.

Obviously, the processing done in this example is trivial, but the intent is to show how the basic algorithm was implemented such that the reader has a starting point for parallelizing the algorithm.

10. The serial_2.h file:

The serial_2.h file is used to define include paths required by C/C++, to define global variables that would be common to any program using this program as a starting point, and to define User definable parameters that govern the problem size (see USER DEFINES and USER GLOBAL VARIABLES).
11. Compiling code based on this example:

Like many UNIX systems, Shale has a “make” utility for compiling programs. The *Makefile* for the program is shown below.

```
C   =gcc
CC  =g++
OFLAG=-O2
LIBP=-L/usr/lib64/ -L/usr/local/lib/
LIBS=-lm
INCP=

Serial_2:
  $(CC) $@.cpp $(OFLAG) $(INCP) $(LIBP) $(LIBS) -o $@
```

However, “make” is a bit fussy and will not compile a program if a valid executable version currently exists in the current directory (talk about lazy!). Therefore, you have to remove the old executable each time before running “make”. However, if you are typing efficient (i.e. lazy) and don’t want to type all those commands to get something compiled, try this script (requires the above *Makefile*):

```
rm serial_2
make serial_2
```

Use an editor to type the above text into a file (I like to call mine “build”) and use chmod to make the file executable (*chmod 755 build*). You can then compile the *serial* program simply by entering “./build” and the UNIX prompt.

12. The batch submission script:

Not required.

13. Running the program:

You can execute the program using the command:

```
./serial_2
```

14. Source code follows:
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <unistd.h>

#define version 1.0
#define yes 1
#define no 0

#define Height 10    // Height of problem (no halo nodes required).
#define Width 4      // Width of problem (no halo nodes required).
/* modify this code to fit their application. */
double DATA[Height][Width];
double DATAprime[Height][Width];
double timeStep = 10.0;
int    verbose = yes;
#include "serial_2.h"

void INITIALIZE_DATA(void) {
    int j, i;
    FILE *data;
    data = fopen("./particle.data", "r");
    for (j = 0; j < Height; j++) {
        for (i = 0; i < Width; i++) {
            fscanf(data, "%lf", &DATA[j][i]);
        }
    }
    fclose(data);
}

void PROCESS_DATA(void) {
    int j;
    double xAverage, yAverage;
    double xDistance, yDistance;
}
double xVelocity, yVelocity;
double xForce, yForce;

    // Find center of gravity for system.
    xAverage = 0.0;
yAverage = 0.0;
    for (j = 0; j < Height; j++) {
        xAverage += DATA[j][0];
yAverage += DATA[j][1];
    }

    // Find new position of each particle.
    for (j = 0; j < Height; j++) {
        xDistance = DATA[j][0] - xAverage;
yDistance = DATA[j][1] - yAverage;

        xForce = 1.0 / pow(xDistance, 3.0);
yForce = 1.0 / pow(yDistance, 3.0);

        xVelocity = DATA[j][2] + xForce * timeStep;
yVelocity = DATA[j][3] + yForce * timeStep;

        DATAprime[j][0] = DATA[j][0] + xVelocity * timeStep;
        DATAprime[j][1] = DATA[j][1] + yVelocity * timeStep;
        DATAprime[j][2] = xVelocity;
        DATAprime[j][3] = yVelocity;
    }
}

/***********************************************************/
/* The UPDATE_DATA routine copies the */
/* iteration's result (held in DATAprime) back */
/* into DATA. */
/* */
/* I would expect the user would want to */
/* modify this code to fit their application. */
/***********************************************************/
void UPDATE_DATA(void) {
    int  i, j;
    for (j = 0; j < Height; j++) {
        for (i = 0; i < Width; i++) {
            DATA[j][i] = DATAprime[j][i];
        }
    }
}

/***********************************************************/
/* The DIFFERENCE_DATA routine calculates the */
/* difference of the previous positions with */
/* current positions. */
/* */
/* I would expect the user would want to */
/* modify this code to fit their application. */
/***********************************************************/
double DIFFERENCE_DATA(void) {
    int  j;
    double difference;
difference = 0.0;
for (j = 0; j < Height; j++) {
    difference += fabs(DATA[j][0] - DATAprime[j][0]);
    difference += fabs(DATA[j][1] - DATAprime[j][1]);
}
return difference;

/**************************************************************************/
/* GENERATION ROUTINES */
/**************************************************************************/
/***************************************************************************/
/* LOAD_BALANCE */
/* This routine does the following: */
/* 1. Call functions to inialize the data. */
/* 2. Call the function to process each */
/*    iteration of the data. */
/* I would expect the user would want to */
/* modify this code to fit their application. */
/**************************************************************************/
void LOAD_BALANCE(void) {

    // Define required parameters.
    int    j;
    int    startTime, stopTime;
    int    iteration;
    double newDifference, oldDifference;
    double convergence;

    /**************************************************************************/
    /* Define user specific parameters. */
    /* It is expected that the user will replace these with their own variable declarations. */
    /**************************************************************************/
    FILE   *log;

    // Records the start time of the program.
    startTime = time(NULL);

    /**************************************************************************/
    /* Initializes the user's data. */
    /* It is expected that the user will replace this call with their own. */
    /**************************************************************************/
    INITIALIZE_DATA();
    oldDifference = 0.0;

    /**************************************************************************/
    /* Here we start the data processing. */
    /**************************************************************************/
    if (verbose == yes) {
        printf("**********************************************************************\n");
        printf("* MESSAGE - Processing data. *\n");

printf("*************************************************
");

// Initialize iteration counter.
iteration = 0;

// Start infinite loop for iterative data processing.
while (1) {

    /**************************************************************************/
    /* Call the routine to perform 1 iteration of data processing. */
    /* It is expected that the user will replace this call with their own. */
    /**************************************************************************/
    PROCESS_DATA();

    /**************************************************************************/
    /* We will assume convergence occurs when no more changes in position occur. */
    /* We have an infinite loop controlling the processing, therefore we MUST have some mechanism to terminate the loop!!! */
    /* It is expected that the user will replace this code with their own. */
    /**************************************************************************/
    newDifference = DIFFERENCE_DATA();
    convergence = fabs(newDifference - oldDifference);
    if (verbose == yes) {
        printf("Iteration [%d] convergence: %lf.\n", iteration, convergence);
    }
    if (convergence < 0.00001 || iteration == 10) break;
    oldDifference = newDifference;

    // Update iteration counter.
    iteration++;

    /**************************************************************************/
    /* Call the routine to update DATA with the results contained in DATAprime. */
    /* It is expected that the user will replace this call with their own. */
    /**************************************************************************/
    UPDATE_DATA();
}

// Write results to log file.
log = fopen("./log", "w");
for (j = 0; j < Height; j++) {
    fprintf(log, "%6.2lf %6.2lf\n", DATA[j][0], DATA[j][1]);
}
fclose(log);
// Records the stopping time of the program.
stopTime = time(NULL);

// Display the program's running time.
if (verbose == yes) {
    printf("Processing time: %d.\n", (stopTime - startTime));
}

/***********************************************************/
/* MAIN.                                                   */
/***********************************************************/
/***********************************************/
/* The MAIN routine initializes the system and */
/* starts the processing.                        */
/* There is nothing for the user to to change   */
/* in this routine.                            */
/***********************************************/
int main(int argc, char **argv) {

    // Process data.
    LOAD_BALANCE();

    // Shut down.
    return 1;
}
Chapter 7: Master-Slave Template #2 (MST_2)

This program was designed to illustrate a loosely coupled parallel situation where you have to write your own code and where we will read the problem’s initial values from a file. The program is a simple C/C++ program that performs an N-body simulation (shown in figure 16) and is based on PVM and thus, you must modify your .bash_profile file to set the paths required by PVM. You must also modify the mst_2.cpp program to initialize, partition, and accumulate the data specific to your task, you must modify the mstSlave_2.cpp program to process your data, and you must modify the mst_2.h file to specify your problem size (otherwise, I do not think mst_2.h will require any changes).

Figure 16.

For this example, we will assume that we have 10 bodies, each having a 2D position coordinate and a 2D velocity vector. We will model the environment by a 10x4 element 2D array “DATA” where each row contains the data <x, y, vx, vy>. This is shown in figure 17.

Figure 17.

The height and width of array “DATA” are defined in the file mst_2.h via the 2 lines:

```
#define Height       10   // Height of problem (no halo nodes required).
```
The algorithm we will use to simulate the system is simple. Each body’s trajectory is deflected towards the center of mass of the system by an amount determined by the body’s distance from the center of mass of the system. Equation (1), (2), and (3) depict the algorithm.

\[ X_i' = X_i + Vx_i' * t \quad \text{and} \quad Y_i' = Y_i + Vy_i' * t \]  \hspace{1cm} (1)

Where:

\[ Vx_i' = (Vx_i + \frac{1}{(X_i - \bar{X})^3} * t) \quad \text{and} \quad Vy_i' = (Vy_i + \frac{1}{(Y_i - \bar{Y})^3} * t) \]  \hspace{1cm} (2)

and

\[ \bar{X} = \frac{1}{10} \sum_{j=0}^{9} X_j \quad \text{and} \quad \bar{Y} = \frac{1}{10} \sum_{j=0}^{9} Y_j \]  \hspace{1cm} (3)

No halo elements are required for this problem, but the problem is still iterative. A single pass will not produce correct results. We must apply the procedure repeatedly until convergence occurs. To keep things simple, we will define convergence as being when no significant changes occur in the average position of the bodies from iteration to iteration.

This problem is nearly embarrassingly parallel as there are many parallelizations possible. Furthermore, since we are using a master-slave approach, we do not need to consider the number of processors that may be available nor any increase in future problem sizes (this is a real strength of the master-slave approach). Thus, we will adopt the simplest partitioning scheme possible: a body-by-body approach.

The height and width of the partitions are defined in the file \textit{mst\_2.h} via the 2 lines:

\begin{verbatim}
#define cellHeight 1 // Height of partition (no halo nodes required).
#define cellWidth 4 // Width of partition (no halo nodes required).
\end{verbatim}

You can define partitions of multiple bodies as well. However, the partitioning scheme you use MUST uniformly fit the problem space.

Even though we have partitioned the problem, the basic algorithm will not change. As this problem is not driven by any neighborhood relationships, \textbf{all} of the data must be used in \textbf{all} iterations. Finally, note that \(X'\) and \(X\), \(Y'\) and \(Y\), \(Vx'\) and \(Vx\), and \(Vy'\) and \(Vy\) are \textbf{NOT} the same variables and that the actual algorithm is:
1. Master initializes M
2. Master spawns slave processes
3. Slaves each initialize their own K
4. Master extracts an overlapping partition R
5. Master sends a slave the partition R
6. Repeat from step 4 until all slaves have a partition
7. Slave performs equation (2) on its R
8. Slaves update R by copying R’ to R (do not copy halo elements)
9. Slaves return R & master receives R
10. Repeat from step 4 until all partitions have been processed
11. Update M by coping M’ to M (do not copy halo elements)
12. Master checks for convergence
13. Repeat from step 4 until convergence occurs

I will describe the design of the example in the remainder of this chapter.

1. **Changes to .bash_profile:**

   The command language interpreter is the interface between the user and the operating system. The command language interpreter executes commands read from standard input or from a file and sets paths to commonly used utilities and libraries. Since most accounts on Shale have Bash set as their default command language interpreter, I will use Bash in the example. As with most command language interpreters, Bash gets user configurable parameters from a `.profile` (pronounced “dot profile”) file. However, with Bash it's a `.bash_profile` (pronounced “dot bash profile”) file.

   ![Note that the .bash_profile file has a period (the “dot”) for its first character which makes it invisible to “ls” (use “ls –la” to see it). The intent of the leading “.” is to hide the file from normal use.]

Since this example uses PVM, you must change your `.bash_profile` file to specify paths required by PVM. To view or edit your `.bash_profile` file:

1. Login (remain in your “home” directory).
2. Use an editor to edit `.bash_profile` (ie vi `.bash_profile`).

   Below is an example `.bash_profile` file (yours is probably very similar) modified to include the paths required by PVM (note the text in bold). You should add the same text to your `.bash_profile` file.

```bash
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs
```
PATH=$PATH:$HOME/bin
export PATH

# PVM specifics
PVM_ROOT=/usr/share/pvm3
PVM_DPATH=/usr/share/pvm3/lib/pvmd
PVM_RSH=/usr/bin/ssh
export PVM_ROOT PVM_DPATH PVM_RSH
unset USERNAME

Once you have made the changes, logout and log back in to have the changes take effect.

2. **The mst_2.cpp file:**

The *mst.cpp* file is the example for the master program. Typically, this program is rather long as it usually is tasked with initializing the data, partitioning the data, load balancing the processing, and accumulating the results.

Obviously, the processing done in this example is trivial, but the intent is to show how to pass different data types, the structure of the master program (the load balancing in particular), and to stress the need to **terminate** the slave processes when they are no longer required.

The master creates a 10x4 2D array of double values, initializes PVM, and calls `PVM_LOAD_BALANCE` to initialize the data (see the `INITIALIZE_DATA` routine) with values read from the “particle.data” file, to calculate the center of mass of the bodies, to partition the data body-by-body (see the `PARTITION_DATA` routine), and to process the data.

Processing is done by sending each body to a slave, acquiring the result (body) from each slave, and inserting that result (body) back into its proper location in the 2D array. This example also sends “bookkeeping” data back and forth between the master and slaves to aid in troubleshooting and management of the load balancing.

The heart of the algorithm is in the `PVM_LOAD_BALANCE` routine. The routine starts by spawning a single slave process on each node. Once each node has had a slave process spawned, if 2 jobs per node have been specified, the process repeats itself by spawning an additional slave process on each node. If more than 2 jobs per node have been specified, the process continually repeats itself until the number of slave processes per node matches the number of jobs per node requested. This interlaced spawning is done to reduce IO contention during slave spawning. Finally, if a slave process fails to spawn, that slave process is not used for any data processing.

The scheme I have employed for load balancing is simple but effective for many types of data processing tasks. The first stage of load balancing extracts a partition of
data and sends the partition to a slave process on one of the nodes. This process repeats until all of the slave processes have been sent data. Like the spawning process, sending data to slave processes is done in an interlaced manner to reduce IO contention during slave execution.

At the second stage of load balancing, once all of the slave processes have been spawned and are working on the data sent, the routine enters an infinite while loop and performs a non-blocking wait (a poll) to check for returned results. As soon as any slave process returns its result, that slave process is immediately sent another data segment to process. This process repeats until all of the data has been sent for processing. It is the user’s responsibility to define when all of the data has been sent for processing and to properly specify that point such that the infinite loop terminates!

At the third stage of load balancing, all of the data has been sent for processing and the algorithm only has to wait until the remaining active slave processes return results (the routine compares the number of “sends” versus the number of “returns”). Again, I use a non-blocking wait (a poll) inside a while loop to check for returned data.

At the fourth stage, all of the slave processes are sent a termination command. I must stress the need to terminate the slave processes when they are no longer required to reduce resource consumption on the node.

The only constraint on the data processing is that all of the bodies be processed before moving on to the next iteration. The specific ordering of body processing is not important. Thus, this problem is considered loosely coupled parallel.

Finally, other than the specific data sent by the master and the actual processing done on the data, this master program and the first example’s master program (mst) are nearly identical in their structure. Another item to note is that the master calculates the mass center and sends the XY coordinates of the mass center to each slave. A more traditional master-slave environment would have sent the entire array to each slave and had each slave calculate its own version of the mass center. Our approach is a slight departure from normal master-slave environments (where the master does none of the processing). However, the traditional approach would result in redundancy and would greatly increase the communication requirements.

3. The mst_2.h file:

The mst_2.h file is used to define include paths required by C/C++, to define global variables that would be common to any program using this example, to define PBS/PVM utility routines that would also be common to any program using this example, and to define user definable parameters that govern the problem size (see USER DEFINES and USER GLOBAL VARIABLES).
4. The mstSlave_2.cpp file:

The mstSlave.cpp file is the example for the slave program. Typically, this program is rather short as all it is usually intended to do is process the blocks of data sent by the master and return results back to the master.

In the example, the slave program starts and waits (a blocking wait) for a message to be received from the master. The message contains either data to be processed or a termination message. If it is data to be processed it will include some “bookkeeping” data (data I found useful for troubleshooting and monitoring the progress of the processing) and the data to be processed. The data is then processed and sent back to the master. If the message contains the termination string (“stop” in this example), the slave program terminates. Obviously, the processing done in this example is trivial, but the intent is to show how to pass different data types, the structure of the slave program, and to stress the need to terminate the slave process when it is no longer required.

Finally, other than the specific data sent by the master and the actual processing done on the data, this slave program and the first example’s slave program (mstSlave) are nearly identical in their structure.

5. Compiling code based on this example:

Like many UNIX systems, Shale has a “make” utility for compiling programs. The Makefile for the programs is shown below.

```make
C     =gcc
CC    =g++

OFLAG=-O2
LIBP=-L/usr/lib64/ -L/usr/local/lib/
    -L/usr/share/pvm3/lib/LINUXX86_64
LIBS=-lpvm3 -lm
INCP=-I/usr/share/pvm3/include

mst_2:
   $(CC) $0.cpp $(OFLAG) $(INCP) $(LIBP) $(LIBS) -o $@

mstSlave_2:
   $(CC) $0.cpp $(OFLAG) $(INCP) $(LIBP) $(LIBS) -o $@
```

However, “make” is a bit fussy and will not compile a program if a valid executable version currently exists in the current directory (talk about lazy!). Therefore, you have to remove the old executable each time before running “make”. However, if you are typing efficient (i.e. lazy) and don’t want to type all those commands to get something compiled, try this script (requires the above Makefile):
6. The batch submission script:

I will assume that you will use the batch scheduler on Shale to submit your job and have provided a simple script to configure the scheduler to use the example:

```
#!/bin/bash
#PBS -S /bin/bash
#PBS -N mst_2
#PBS -q long
#PBS -d /home/rmarsh/MST/BIN/
#PBS -o mst_2.o
#PBS -e mst_2.e
#PBS –m abe
#PBS -l nodes=4:ppn=1
./mst_2
```

In order, the items this script configures are:
28. “#PBS –S” Sets the execution shell to bash (can leave as is).
29. “#PBS –N” Sets the name of the job to “mst_2” (you can change or leave as is).
30. “#PBS –q” Sends the job to the “long” queue (can leave as is).
31. “#PBS –d” Sets the working directory to “/home/rmarsh/MST/BIN” (you must change this to reflect your directory organization).
32. “#PBS –o” Writes the output to a file (in the working directory) called “mst_2.o” (you can change or leave as is).
33. “#PBS –e” Writes errors to a file (in the working directory) called “mst_2.e” (you can change or leave as is).
34. “#PBS –m” Instructs the scheduler to send email when the job begins execution (b), terminates (e), or is aborted by the scheduler (a). You can use any combination of “abe.” If you do not want email sent, remove this line (change as desired).
35. “#PBS –l” Sets limits on the job execution. This example specifies 4 nodes with 1 process per node. Note that there are many options. Do a “Man pbs_resources” for more info (see text box below).
36. “./mst_2” Instructs the batch scheduler to run the program. You must change this argument to specify your process.

In general, parallelizing a job across a number of nodes and processes per node is not as obvious as it may first seem. Fortunately, if you have a task that this
template was designed to manage, you can probably specify as many nodes as you have data files (however, the heuristic is to specify, at most, half as many nodes as data files). Specifying the processes per node is not as obvious. Here’s another heuristic: If your processing requires mainly computations and little IO, specify 2 processes per node and reduce the number of nodes requested. If your processing involves a lot of IO, you may want to only specify 1 process per node.

Finally, remember that this template implements a master-slave paradigm. Therefore, 1 entire node (regardless of the number of processes per node specified) will be allocated to the master. For example, in this example only 3 slave processes will be created even though 4 processes have been requested via “#PBS –l”

7. Running the program:

Once you have PVM configured and a batch submission script configured you can execute the program using the command:

qsub ./name_of_batch_script

Note that when using the batch scheduler no output is displayed on the computer screen, but is written to an output file (see “#PBS –o” and “#PBS –e” above).

To summarize, modify your .bash_profile file to include PVM paths, edit mstSlave_2.cpp and mst_2.cpp according to your task, compile mstSlave_2.cpp and mst_2.cpp using the Makefile (and optional “build” script) shown above, and then run the program using the batch scheduler.

Since the program gets the list of nodes to execute on from the batch scheduler each time it is executed, you can change the number of nodes or processes per node in the batch submission script and any derived program will adjust the load balancing accordingly. Remember, however, that the number of processes and the partitioning scheme must match.

Finally, the load balancing mechanism is master-slave and naïve. Since it does not have any information regarding the processing time of any partition, it may very well end up waiting for a single long-to-process partition to be completed while all of the other partitions have long finished. An informed scheduler (or researcher) would adjust the partition sizes to account for this difference.

8. Source code follows:
# include <stdio.h>
# include <string.h>
# include <stdlib.h>
# include <math.h>
# include <time.h>
# include <unistd.h>
# include "pvm3.h"

/* GLOBAL PARAMETERS & VARIABLES */

/* SYSTEM DEFINES */
/* !No changes should be made to these! */

#define version 1.0
#define yes 1
#define no 0
#define maxNodes 128
#define maxCPUs 256

/* USER DEFINES */
/* Define the dimensions of the data to be processed. Note that the actual data to be processed in this template is a list of 10 with 4 datum per element. */
/* with 4 datum per element. */
/* I would expect the user would want to modify this code to fit their application. */

#define Height 10    // Height of problem (no halo nodes required).
#define Width 4      // Width of problem (no halo nodes required).
#define cellHeight 1  // Height of partition (no halo nodes required).
#define cellWidth 4   // Width of partition (no halo nodes required).
```c
#define slavePath "/home/rmarsh/MST/BIN/mstSlave_2"

/**********************************************
/* USER GLOBAL VARIABLES */
/* */
/* Verbose is used to display log messages */
/* Turn on display with (verbose = yes) or */
/* turn off display with verbose = no). */
/* */
/* I would expect the user would want to */
/* modify this code to fit their application. */
/***********************************************/
double timeStep = 10.0;
int verbose = yes;

/**********************************************
/* SYSTEM GLOBAL VARIABLES */
/* */
/* !No changes should be made to these! */
/* */
/***********************************************/
int nodesWanted;
int jobsPerNode;
char nodeNames[maxNodes][256];
char nodeMaster[256];
int TID[maxCPUs];
char pvmCommand[256];

/**********************************************
/* CLUSTER/PVM UTILITY ROUTINES */
/***********************************************/

/**********************************************
/* READ_MASTERNODEFILE */
/* This routine does the following: */
/* 1. Prompts the system for the host on which */
/* the master resides. */
/* */
/* !No changes should be made to this routine! */
/* */
/***********************************************/
void READ_MASTERNODEFILE(void) {
  FILE *masterNodeFile;
  if (verbose) {
    printf("*************************************************
");
    printf("* MESSAGE - Identifying master node. *
");
    printf("*************************************************
");
  }
  system("uname -n > masterNodeFile");
  masterNodeFile = fopen("./masterNodeFile", "r");
  if (masterNodeFile == NULL) {
    printf("ERROR - cannot identify master node.\n");
    exit(0);
  }
  fscanf(masterNodeFile, "%s", nodeMaster);

```
fclose(masterNodeFile);

/***********************************************/
/* READ_PBS_HOSTFILE */
/* This routine does the following: */
/* 1. Copies the contents of the $PBS_NODEFILE */
/*    environment variable to a local file */
/*    called "pbshostfile." */
/* 2. Parses the pbshostfile to extract the */
/*    number of nodes and jobs per node wanted.*
/* 3. Creates a PVM compatible hostfile. */
/* */
/* !No changes should be made to this routine! */
/* */
/***********************************************/
void READ_PBS_HOSTFILE(void) {
    int  i, cnt;
    int  flag;
    char tempName[maxCPUs][256];
    FILE *pbshosts;
    FILE *pvmhosts;
    if (verbose == yes) {
        printf("*************************************************
        * MESSAGE - Creating PVM hostfile.              *
        *************************************************
        ");
    }
    // Copy PBS hostfile to PVM hostfile.
    // system("cat $PBS_NODEFILE > ./pbshostfile");
    pbshosts = fopen("./pbshostfile", "r");
    if (pbshosts == NULL) {
        printf("ERROR - Cannot read PBS nodefile ./pbshostfile.
        ");
        exit(0);
    }
    cnt = 0;
    while (1) {
        fscanf(pbshosts, "%s", tempName[cnt]);
        if (feof(pbshosts)) break;
        cnt++;
    }
    fclose(pbshosts);
    // Extract number of nodes and processes per node.
    flag = 1;
    jobsPerNode = 1;
    nodesWanted = 0;
    strcpy(nodeNames[nodesWanted], tempName[0]);
    for (i = 1; i < cnt; i++) {
        if (strcmp(nodeNames[nodesWanted], tempName[i])) {
            nodesWanted++;
            strcpy(nodeNames[nodesWanted], tempName[i]);
            flag = 0;
        }
        if (flag == 1) jobsPerNode++;
nodesWanted++;

// Write PVMhostfile.
pvmhosts = fopen("./pvmhostfile", "w");
if (pvmhosts == NULL) {
    printf("ERROR - Cannot write PVM hostfile ./pvmhostfile.\n");
    exit(0);
}
for (i = 0; i < nodesWanted; i++) {
    fprintf(pvmhosts, "%s\n", nodeNames[i]);
}
fclose(pvmhosts);
/* This routine does the following: */
/* 1. Starts PVM using the nodes listed in the */
/* PVM compatible hostfile. */
/* */
/* !No changes should be made to this routine! */
/* */
/***************************************************************************/
void PVM_START(void) {
    int block = yes;
    int argc = 1;
    char *argv[] = {"./pvmhostfile"};
    if (verbose == yes) {
        printf("*************************************************
");
        printf("* MESSAGE - Starting PVM deamons.               *
");
        printf("*************************************************
");
    }
    // system("xterm -iconic -e pvm ./pvmhostfile &");
    // system("pvm ./pvmhostfile &");
    // sleep(5);
    pvm_start_pvmd(argc, argv, block);
}
***************************************************************************/
/* INITIALIZE_PVM */
/* This routine does the following: */
/* 1. Calls the above 3 routines in proper */
/* order to start PVM. */
/* */
/* !No changes should be made to this routine! */
/* */
/***************************************************************************/
void INITIALIZE_PVM(void) {
    READ_MASTERNODEFILE();
    READ_PBS_HOSTFILE();
    READ_PVM_HOSTFILE();
    PVM_START();
}
***************************************************************************/
/* SPAWN_PVM_NODES */
/* This routine does the following: */
/* 1. Initializes the array used to record */
/* slave process IDs involved with the */
/* processing. */
/* 2. Spawns the slave process via PVM. Note */
/* if there are 2 (or more) jobs per node */
/* specified, we spawn one slave process on */
/* each node first and then go back and */
/* spawn a second process. We do this to */
/* reduce IO contention during the spawning */
/* process. If slave fails to spawn, it is */
/* not used. */
/***************************************************************************/
void SPAWN_PVM_NODES(void) {
int cc, j, n;
int tid;

    // Initialize array to hold node IDs.
    for (j = 0; j < jobsPerNode; j++) {
        for (n = 0; n < nodesWanted; n++) {
            TID[j*nodesWanted+n] = -1;
        }
    }

    // Spawn slave processes.
    if (verbose == yes) {
        printf("*************************************************
");
        printf("* MESSAGE - Spawning slave processes.          *
");
        printf("*************************************************
");
    }
    for (j = 0; j < jobsPerNode; j++) {
        for (n = 0; n < nodesWanted; n++) {
            cc = pvm_spawn(slavePath, NULL, 1, nodeNames[n], 1, &tid);
            if (cc == 1) {
                TID[j*nodesWanted+n] = tid;
                if (verbose == yes) {
                    printf("MESSAGE - Slave spawned on node: %s
",
                        nodeNames[n]);
                }
            } else {
                if (verbose == yes) {
                    printf("WARNING - Slave failed on node: %s
",
                        nodeNames[n]);
                }
            }
        }
    }

    /*******************************************************************************
    /* SHUT_DOWN_PVM */
    /* This routine does the following: */
    /* 1. Shuts down PVM. */
    /* */
    /* !No changes should be made to this routine! */
    /* */
    /*******************************************************************************
void SHUT_DOWN_PVM(void) {
    if (verbose == yes) {
        printf("*************************************************
");
        printf("* MESSAGE - Shutting down PVM deamons.     *
");
        printf("*************************************************
");
    }
    pvm_halt();
}
/***********************************************************/
/* mstSlave_2.cpp                                          */
/*                                                          */
/* Template for a generic PVM slave program.                */
/*                                                          */
/************************************************************/
#include "mst_2.h"

/***********************************************************/
/* GLOBAL VARIABLES                                        */
/***********************************************************/
	
_paid
#include "mst_2.h"

/***********************************************************/
/* DATA PARTITION/GENERATION ROUTINES                      */
/***********************************************************/
	
_paid
#include "mst_2.h"

/***********************************************************/
/* DATA PROCESSING ROUTINES                                */
/***********************************************************/
	
_paid
#include "mst_2.h"

/***********************************************************/
/* PROCESS_PARTITION routine performs the calculations to */
/* adjust each particle's position based on the center of */
/* gravity of the entire system of particles.             */
/***********************************************************/
void PROCESS_PARTITION(void) {
    int j, i;
    //double xAverage, yAverage;
    double xDistance, yDistance;
    double xVelocity, yVelocity;

double xForce, yForce;
double PARTITIONprime[cellHeight][cellWidth];

// Find new position of each particle.
for (j = 0; j < cellHeight; j++) {
    xDistance = PARTITION[j][0] - xAverage;
yDistance = PARTITION[j][1] - yAverage;

    xForce = 1.0 / pow(xDistance, 3.0);
yForce = 1.0 / pow(yDistance, 3.0);

    xVelocity = PARTITION[j][2] + xForce * timeStep;
yVelocity = PARTITION[j][3] + yForce * timeStep;

    PARTITIONprime[j][0] = PARTITION[j][0] + xVelocity * timeStep;
    PARTITIONprime[j][1] = PARTITION[j][1] + yVelocity * timeStep;
    PARTITIONprime[j][2] = xVelocity;
    PARTITIONprime[j][3] = yVelocity;
}

// Update PARTITION with results.
for (j = 0; j < cellHeight; j++) {
    for (i = 0; i < cellWidth; i++) {
        PARTITION[j][i] = PARTITIONprime[j][i];
    }
}

/*****************************/
/* MAIN.                     */
/*****************************/
int main(void) {
    /**************************/
    /* The following 5 parameters are either */
    /* required (ptid) or that have proven useful */
    /* for "bookkeeping" purposes. These are the */
    /* same "bookkeeping" parameters as described */
    /* in the mst.cpp Master Slave Template */
    /* document. */
    /**************************/
    int ptid;       // Required.
    int tid;        // For bookkeeping.
    int nodeNumber; // For bookkeeping.
    int dataNumber; // For bookkeeping.
    char command[256]; // For control.
    int j;

    /**************************/
    /* Initializes the user's data. */
    /* */
    /* It is expected that the user will replace */
    /* this call with their own. */
    /**************************/

    105
/* This line returns the ID of the parent or */
/* master process (ptid). This information is */
/* required as several PVM jobs may be running */
/* (each with a unique master) and the slave */
/* needs to know which master to communicate */
/* with. */
/* ********************************************/ 
ptid = pvm_parent();

/* ********************************************/ 
/* Begin an infinite loop to acquire and */
/* process data as it arrives. */
/* ********************************************/ 
while (1) {

/* ********************************************/ 
/* Perform a blocking wait forcing the program */
/* to stop and wait for data from the master. */
/* ********************************************/ 
pvm_recv(ptid, 1);

/* ********************************************/ 
/* Unpack the "bookkeeping" data that was sent */
/* by the master. Note that this data is NOT */
/* used in any processing and is simply */
/* returned to the master with the processed */
/* data. */
/* */
/* */
/* The order of the data unpacking MUST match */
/* the order that it was packed by the master! */
/* ********************************************/ 
pvm_upkint(&tid, 1, 1);
pvm_upkint(&nodeNumber, 1, 1);
pvm_upkint(&dataNumber, 1, 1);
pvm_upkstr(command);

/* ********************************************/ 
/* Unpack the user's data that was sent by the */
/* master. */
/* */
/* The order of the data unpacking MUST match */
/* the order that it was packed by the master! */
/* */
/* I would expect the user would want to */
/* modify this code to fit their application. */
/* ********************************************/ 
pvm_upkdoublle(&xAverage, 1, 1);
pvm_upkdoublle(&yAverage, 1, 1);
for (j = 0; j < cellHeight; j++) {
    pvm_upkdoublle(PARTITION[j], cellWidth, 1);
}

/* ********************************************/ 
/* These lines terminate the program if the */
/* passed string "command" contains the text */
/* "stop". This is a required section! */
/* Otherwise, the slave process will never */
/* terminate! */
 /**************************************************************************/
 if (!strcmp(command, "stop")) {
   pvm_exit();
   exit(0);
 }
 /**************************************************************************/
/* Call the user’s routine to process the data */
/* that was sent by the master. */
/* */
/* I would expect the user would want to */
/* modify this code to fit their application. */
/**************************************************************************/
PROCESS_PARTITION();
/**************************************************************************/
/* Initiate communication to the master. */
/**************************************************************************/
pvm_initsend(PvmDataRaw);
/**************************************************************************/
/* Pack the "bookkeeping" data into the */
/* communication buffer. */
/**************************************************************************/
pvm_pkint(&tid, 1, 1);
pvm_pkint(&nodeNumber, 1, 1);
pvm_pkint(&dataNumber, 1, 1);
/**************************************************************************/
/* Pack the processed data into the */
/* communication buffer. */
/* */
/* I would expect the user would want to */
/* modify this code to fit their application. */
/**************************************************************************/
for (j = 0; j < cellHeight; j++) {
  pvm_pkdouble(PARTITION[j], cellWidth, 1);
}
/**************************************************************************/
/* Return the data in the communication buffer */
/* to the master. */
/**************************************************************************/
pvm_send(ptid, 1);
return 1;
/* Include the header file which includes any */
/* required global variable definitions, any */
/* user global variable definitions, and any */
/* required functions. */
/* See the mst_2.h file for details. */
#include "mst_2.h"

double xAverage, yAverage;
double DATA[Height][Width];
double DATAprime[Height][Width];
double PARTITION[cellHeight][cellWidth];

/* The INITIALIZE_DATA routine reads the values*/
/* for each particle's original location and */
/* velocity <x, y, vx, vy> from a data file. */
/* I would expect the user would want to */
/* modify this code to fit their application. */
void INITIALIZE_DATA(void) {
  int j, i;
  FILE *data;
  data = fopen("./particle.data", "r");
  for (j = 0; j < Height; j++) {
    for (i = 0; i < Width; i++) {
```c
fscanf(data, "%lf", &DATA[j][i]);
}
fclose(data);

/***********************************************/
/* DATA PROCESSING ROUTINES */
/*************************************************************/
/*/ The PROCESS_GLOBAL_DATA routine performs the*/
/*/ calculations to find the center of gravity */
/*/ of the particles. */
/*/ */
/*/ I would expect the user would want to */
/*/ modify this code to fit their application. */
/*************************************************************/
void PROCESS_GLOBAL_DATA(void) {
    int    j;
    xAverage = 0.0;
    yAverage = 0.0;
    for (j = 0; j < Height; j++) {
        xAverage += DATA[j][0];
        yAverage += DATA[j][1];
    }
}
/**/ The PARTITION_DATA routine partitions */
/**/ user's data. */
/**/ */
/**/ I would expect the user would want to */
/**/ modify this code to fit their application. */
/*************************************************************/
void PARTITION_DATA(int segment) {
    int i, j;
    for (j = 0; j < cellHeight; j++) {
        for (i = 0; i < cellWidth; i++) {
            PARTITION[j][i] = DATA[segment+j][i];
        }
    }
}
/**/ The ACCUMULATE_DATA routine inserts the */
/**/ processed results into the array holding */
/**/ the intermediate results. Data ordering is */
/**/ maintained. */
/**/ */
/**/ I would expect the user would want to */
/**/ modify this code to fit their application. */
/*************************************************************/
void ACCUMULATE_DATA(int segment) {
    int i, j;
    for (j = 0; j < cellHeight; j++) {
        for (i = 0; i < cellWidth; i++) {
            // Code continues here...
        }
    }
}
```

DATAprime[segment+j][i] = PARTITION[j][i];
}
}

/******************************************************************************
/* The UPDATE DATA routine copies the */
/* iteration's result (held in DATAprime) back */
/* into DATA. */
/* */
/* I would expect the user would want to */
/* modify this code to fit their application. */
/*******************************************************************************/
void UPDATE_DATA(void) {
    int i, j;
    for (j = 0; j < Height; j++) {
        for (i = 0; i < Width; i++) {
            DATA[j][i] = DATAprime[j][i];
        }
    }
}

/******************************************************************************
/* The DIFFERENCE DATA routine calculates the */
/* difference of the previous positions with */
/* current positions. */
/* */
/* I would expect the user would want to */
/* modify this code to fit their application. */
/*******************************************************************************/
double DIFFERENCE_DATA(void) {
    int j;
    double difference;
    difference = 0.0;
    for (j = 0; j < Height; j++) {
        difference += fabs(DATA[j][0] - DATAprime[j][0]);
        difference += fabs(DATA[j][1] - DATAprime[j][1]);
    }
    return difference;
}

/******************************************************************************
/* PVM GENERATION ROUTINES */
/*******************************************************************************/
/* The PVM_SEND routine is used to send data */
/* to the slaves. */
/* */
/* We could embed this function into the load */
/* balancing routine below, but we would have */
/* to do so in several places. Thus, it is */
/* better software engineering practice to */
/* encapsulate its functionality into a single */
/* function. */
/* */
/* I would expect the user would want to */
void PVM_SEND(int tid, int nodeNumber, int dataNumber, char command[])
{
    int j;

    /* Initiates communication to the slave. */
    pvm_initsend(PvmDataRaw);

    /* Pack the "bookkeeping" parameters into the communication buffer. */
    pvm_pkint(&tid, 1, 1);
    pvm_pkint(&nodeNumber, 1, 1);
    pvm_pkint(&dataNumber, 1, 1);
    pvm_pkstr(command);

    /* Pack the user's data to be processed into the communication buffer. */
    pvm_pkdouble(&xAverage, 1, 1);
    pvm_pkdouble(&yAverage, 1, 1);
    for (j = 0; j < cellHeight; j++) {
        pvm_pkdouble(PARTITION[j], cellWidth, 1);
    }

    /* Send the data in the communication buffer to the slave identified by the tid parameter. */
    pvm_send(tid, 1);
}

void PVM_RECEIVE(int *tid, int *nodeNumber, int *dataNumber) {
    int j;

/* Unpack the "bookkeeping" data from the communication buffer. */
pvm_upkint(tid, 1, 1);
pvm_upkint(nodeNumber, 1, 1);
pvm_upkint(dataNumber, 1, 1);

/* Unpacks the processed user's data from the communication buffer. */
/* I would expect the user would want to modify this code to fit their application. */
for (j = 0; j < cellHeight; j++) {
    pvm_upkdouble(PARTITION[j], cellWidth, 1);
}

/* PVM_LOAD_BALANCE */
/* This routine does the following: */
/* 1. Inializes the user's data. */
/* 2. Spawns the slave jobs on the nodes and sends each slave data to process. */
/* 3. Iteratively processes the data via: */
/*   a) Nonblockingly waits for (polls) each slave to complete the processing of its current data and sends that slave another block of data to process. */
/*   This continues untill all data has been sent. */
/*   b) Nonblockingly waits for (polls) each slave to complete the processing of its current data. This continues until all sent data has been processed. */
/*   c) Checks for convergence and stops the data processing if we have convergence. Otherwise, it repeats from "a"). */
/* 4. Sends each slave a termination signal. */
/* 5. Prints the results to a log file. */
/* I would expect the user would want to modify this code to fit their application. */
void PVM_LOAD_BALANCE(void) {
    // Define required parameters.
    int    j, n;
    int    tid;
    int    pvmSent, pvmReceived;
    int    startTime, stopTime;
    int    dataCurrent, dataReturned;
    ...
int iteration;
double newDifference, oldDifference;
double convergence;

/**************************************************************/
/* Define user specific parameters. */
/* */
/* It is expected that the user will replace */
/* these with their own variable declarations. */
/**************************************************************/
FILE *log;

// Records the start time of the program.
startTime = time(NULL);

// Spawn slave nodes.
strcpy(pvmCommand, "go");
SPAWN_PVM_NODES();

/**************************************************************/
/* Initializes the user's data. */
/* */
/* It is expected that the user will replace */
/* this call with their own. */
/**************************************************************/
INITIALIZE_DATA();
oldDifference = 0.0;

/**************************************************************/
/* Here we start the data processing. */
/* */
/* The only limitation to the data processing */
/* is that all partitions must be processed */
/* BEFORE another iteration can begin and the */
/* only communication requirement is for the */
/* slaves to exchange data with the master. */
/* Thus, this example is considered "loosly */
/* coupled". */
/**************************************************************/
if (verbose == yes) {
    printf("*********************************************************
");
    printf("MESSAGE - Processing data. \n");
    printf("*********************************************************
");
}

// Initialize iteration counter.
iteration = 0;

// Start infinite loop for iterative data processing.
while (1) {

/**************************************************************/
/* Initialize "bookkeeping" parameters. */
/**************************************************************/
pvmSent = 0;
pvmReceived = 0;
dataCurrent = 0;
dataReturned = 0;
strcpy(pvmCommand, "NA");

/***********************************************/
/* To reduce communication, we will calculate */
/* the center of mass for the system before */
/* pass this info on to the slaves. Our other */
/* choice would be to send the entire data */
/* array to each slave and have each slave do */
/* this calculation - redundant! */
/***********************************************/
PROCESS_GLOBAL_DATA();

/***********************************************/
/* Here we partition the user's data, and send */
/* each process data to work on. If there are */
/* 2 (or more) jobs per node specified, we */
/* send data to one slave process on each node */
/* first and then go back and send data to the */
/* second process. We do this to reduce IO */
/* contention during slave process executions. */
/* */
/* It is expected that the user will need to */
/* change the PARTITION_DATA call, and */
/* the PVM_SEND call. It is also expected that */
/* the user will need to specify their own */
/* method of determining when to "break" (which*/
/* should not be required here unless the */
/* problem was not setup correctly). */
/***********************************************/
for (j = 0; j < jobsPerNode; j++) {
    for (n = 0; n < nodesWanted; n++) {
        // Partition user data.
        PARTITION_DATA(dataCurrent);

        // Break if no more data to process.
        // P.S. This should not occur if you set the job up
        // correctly.
        if (dataCurrent == Height) break;

        // Send data to first available slave.
        if (TID[j*nodesWanted+n] != -1) {
            PVM_SEND(TID[j*nodesWanted+n], n, dataCurrent,
            pvmCommand);
            pvmSent++;
            if (verbose == yes) {
            printf("Iteration [%d]: Package %d > %s\n",
            iteration, dataCurrent, nodeNames[n]);
            }
        }
        dataCurrent+=cellHeight;
    }
}

/***********************************************/
/* At this point, all of the nodes have had */
/*
/* data sent to them for processing and we 
* wait for datum to be returned by a (any) 
* slave. As soon as the master receives data 
* from a/any slave, the master immediately 
* sends another data partition to that slave. */ 
/* This process repeats until all of the data 
* has been sent for processing (it is the 
* user's task to define when this occurs - see* 
* the if / break statement). Finally, I use a */ 
/* non-blocking wait (ie a poll) inside an */ 
/* infinite while loop to check for returned */ 
/* data. */ 
/* */ 
/* It is expected that the user will need to */ 
/* change the parameters in the PVM_RECEIVE 
* call, the ACCUMULATE_DATA call, the */ 
/* PARTITION_DATA call, and the PVM_SEND call. */ 
/* It is also expected that the user will need */ 
/* to specify their own method of determining */ 
/* when to "break". */ 
*******************************************************************************/
while (1) {
    if (pvm_nrecv(-1,-1) > 0) {
        // Get data from slave.
PVM_RECEIVE(&tid, &n, &dataReturned);
        // Accumulate user data.
        ACCUMULATE_DATA(dataReturned);
pvmReceived++;
        if (verbose == yes) {
            printf("Iteration [%d]: Package %4d < %s\n", iteration,
dataReturned, nodeNames[n]);
        }
        // Partition user data.
PARTITION_DATA(dataCurrent);
        // Break if no more data to process.
        // Very Important line !!!
        // I Repeat. This is a very Important line !!!
        // Don't forget this very Important line !!!
        // Get the hint?
        if (dataCurrent == Height) break;
        // Send data to slave.
PVM_SEND(tid, n, dataCurrent, pvmCommand);
pvmSent++;
        if (verbose == yes) {
            printf("Iteration [%d]: Package %4d > %s\n", iteration,
dataCurrent, nodeNames[n]);
        }
dataCurrent+=cellHeight;
    }
}
*******************************************************************************/
/* At this point, since all of the data has been sent for processing, all we have left to do is loop until the remaining data gets returned by the slaves. I use a non-blocking wait (ie a poll) to check for returned data.*/
/* It is expected that the user will need to change the PVM_RECEIVE call and the ACCUMULATE_DATA_BY_ROW call.*/
/***********************************************/
do {
    if (pvm_nrecv(-1,-1) > 0) {
        // Get data from slave.
PVM_RECEIVE(&tid, &n, &dataReturned);

        // Accumulate user data.
        ACCUMULATE_DATA(dataReturned);
pvmReceived++;
        if (verbose == yes) {
            printf("Iteration [%d]: Package %4d < %s\n", iteration, dataReturned, nodeNames[n]);
        }
    }
} while (pvmReceived < pvmSent);

/***********************************************/
/* We will assume convergence occurs when no more changes in position occur.*/
/* We have an infinite loop controlling the processing, therefore we MUST have some mechanism to terminate the loop!!!*/
/* It is expected that the user will replace this code with their own.*/
/***********************************************/
newDifference = DIFFERENCE_DATA();
convergence = fabs(newDifference - oldDifference);
if (verbose == yes) {
    printf("Iteration [%d] convergence: %lf.\n", iteration, convergence);
}
if (convergence < 0.00001 || iteration == 10) break;
oldDifference = newDifference;

// Update iteration counter.
iteration++;

/***********************************************/
/* Call the routine to update DATA with the results contained in DATAnode.*/
/* It is expected that the user will replace this call with their own.*/
}
UPDATE_DATA();

// Write results to log file.
log = fopen("./log", "w");
for (j = 0; j < Height; j++) {
    fprintf(log, "<%6.2lf %6.2lf>\n", DATA[j][0], DATA[j][1]);
}
fclose(log);

/* Sends all slaves involved in the processing */
/* a termination command. */
/* It is expected that the user will need to */
/* change the parameters in the PVM_SEND call. */
strcpy(pvmCommand, "stop");
for (j = 0; j < jobsPerNode; j++) {
    for (n = 0; n < nodesWanted; n++) {
        if (TID[j*nodesWanted+n] != -1) {
            PVM_SEND(TID[j*nodesWanted+n], n, n, pvmCommand);
        }
    }
}

// Records the stopping time of the program.
stopTime = time(NULL);

// Display the program's running time.
if (verbose == yes) {
    printf("Processing time: %d.\n", (stopTime - startTime));
}

int main(int argc, char **argv) {

    // Initialize PVM.
    INITIALIZE_PVM();

    // Process data.
    PVM_LOAD_BALANCE();

    // Shut down.
SHUT_DOWN_PVM();
return 1;
}
Chapter 8: Coordinating Peers Template #2 (PTP_2)

This program was designed to illustrate a tightly coupled parallel situation where you have to write your own code and where we will read the problem’s initial values from a file. The program is a simple C/C++ program that performs an N-body simulation (shown in figure 18) and is based on MPI. You must modify the ptp_2.cpp program to initialize, partition, and accumulate the data specific to your task, and you must modify the ptp_2.h file to specify your problem size (otherwise, I do not think ptp_2.h will require any changes).

Figure 18.

For this template, we will assume that we have 10 bodies, each having a 2D position coordinate and a 2D velocity vector. We will model the environment by a 10x4 element 2D array “DATA” where each row contains the data <x, y, vx, vy>. This is shown in figure 19.

Figure 19.

The height and width of array “DATA” are defined in the file ptp_2.h via the 2 lines:
The algorithm we will use to simulate the system is simple. Each body’s trajectory is deflected towards the center of mass of the system by an amount determined by the body’s distance from the center of mass of the system. Equation (1), (2), and (3) depict the algorithm.

\[ X'_i = X_i + Vx_i^* t \quad \text{and} \quad Y'_i = Y_i + Vy_i^* t \]  

(1)

Where:

\[ Vx_i^* = (Vx_i + \frac{1}{(X_i - \bar{X})^3})^* t \quad \text{and} \quad Vy_i^* = (Vy_i + \frac{1}{(Y_i - \bar{Y})^3})^* t \]  

(2)

and

\[ \bar{X} = \frac{1}{10} \sum_{j=0}^{9} X_j \quad \text{and} \quad \bar{Y} = \frac{1}{10} \sum_{j=0}^{9} Y_j \]  

(3)

No halo elements are required for this problem, but the problem is still iterative. A single pass will not produce correct results. We must apply the procedure repeatedly until convergence occurs. To keep things simple, we will define convergence as being when no significant changes occur in the average position of the bodies from iteration to iteration.

This problem is nearly embarrassingly parallel as there are many parallelizations possible. However, since we are using a coordinating peers approach, we do need to consider the number of processors that will be available and any increase in future problem sizes. Regardless, we will adopt the simplest partitioning scheme possible: a body-by-body approach.

The height and width of the partitions are defined in the file `ptp_2.h` via the 2 lines:

```
#define cellHeight  1    // Height of partition (no halo nodes required).
#define cellWidth   4    // Width of partition (no halo nodes required).
```

Note that the example allows you to define partitions of any size, detects the number of processors assigned (see the batch submission script section below), and creates the same number of partitions as processors assigned. Therefore, the user can create a custom partitioning scheme. However, the partitioning scheme you use MUST uniformly fit the problem space.
Even though we have partitioned the problem, the basic algorithm will not change. As this problem is not driven by any neighborhood relationships, all of the data must be used in all iterations. Finally, note that $X'$ and $X$, $Y'$ and $Y$, $Vx'$ and $Vx$, and $Vy'$ and $Vy$ are NOT the same variables and that the actual algorithm is:

1. Node 0 reads the values for DATA from the file “particles.data”
2. Node 0 broadcasts DATA to all other nodes.
3. Each node extracts its own partition from DATA
4. Each node finds the center of mass via equation (3)
5. Each node adjusts its partition’s body velocities via equation (2)
6. Each node adjusts its partition’s body positions via equation (1)
7. Each node calculates its local change in position
8. Node 0 requests the sum of all position changes be obtained (an MPI call)
9. Node 0 for checks for overall convergence
10. Each node updates its partition by coping $X'$ to $X$, $Y'$ to $Y$, $Vx'$ to $Vx$, and $Vy'$ to $Vy$
11. Each node broadcasts its partition to node 0
12. Node 0 updates DATA
13. Node 0 broadcasts DATA to all other nodes.
14. Repeat from step 3 until convergence occurs

Note that it is common (required?) by a program using the coordinating peers paradigm to still have a controlling node (e.g. a “master” or “root” node). Typically, it is the “root” node’s responsibility to handle terminal IO, to determine when to stop, and possibly to accumulate the final results. In our example, node 0 is the “root” node.

Another item of interest is that fact that our algorithm is not a true equal and identical coordinating peers algorithm as we are tasking node 0 with significant additional work (by collecting the individual partitions, updating DATA, and broadcasting the new DATA – an all-gather operation). However, if want the nodes to be equal and identical, then all of the nodes would have to send their partitions to all other nodes and receive all other node’s partitions. Since we have 10 nodes in our example and since each node would send 9 partitions (and receive 9 partitions), we would have 90 messages per iteration. Using the all-gather approach, we reduce the number of messages to 18 per iteration. Remember that it is always better to send fewer large messages than many small messages.

1. The ptp_2.cpp file:

The ptp.cpp file is the only program in the example as all nodes perform the same task(s).

Obviously, the processing done in this example is trivial, but the intent is to show how to pass different data types, the structure of a coordinating peer program (the load balancing in particular), and to stress the need to terminate each peer process when they are no longer required.
The master creates a 10x4 2D array of double values and calls MPI_LOAD_BALANCE to initialize the data (see the INITIALIZE_DATA routine) with values read from the “particle.data” file, to partition the data body-by-body (see the EXTRACT_PARTITION routine), and to process the data.

Processing is done by first broadcasting the entire data set to each node using MPI_Bcast(). Note that the data exchange using MPI_Bcast() also acts to synchronize the peers as each must wait for the sent data to be received. The routine then calls the EXTRACT_PARTITION routine and the PROCESS_PARTITION routine to extract and process the local partition. MPI_Barrier() (following the PROCESS_DATA call) is used to synchronize the peers again. Forcing each peer to reach this point in the algorithm and wait until all other peers have also reached this point. Once synchronization occurs, each node calls DIFFERENCE_DATA to obtain the sum of the position differences for their local partition. The following MPI_Reduce() is used to accumulate/calculate a sum of the local difference sum and to place that value in node 0 (our “root” node). The root node then determines if convergence has been reached. If convergence has been reached, the root node copies the string “stop” into the string variable “mpiCommand” (which was initialized with the string “go”) and calls MPI_Bcast() to broadcast the string variable “mpiCommand” to all nodes. As with the master-slave example, the broadcasting of the string variable “mpiCommand” to all nodes is akin to sending all slave processes a termination command. I must stress the need to terminate the peer processes when they are no longer required to reduce resource consumption on the node. If convergence has not been reached, the local partition is updated with the just calculated position and velocity data and the all-gather operation is performed preparing the system for the next iteration.

One of the unique characteristics of a coordinating-peers system is that all of the peers are just that, peers. All equal and identical. Fortunately, MPI assign IDs (stored in the variable myID in the example) to each peer and we can use that information to force a uniqueness on the peers. If you examine the file ptp_2.cpp you will see that I use myID in several places to assign particular functionality to specific peers.

The only constraint on the data processing is that all of the bodies be processed before moving on to the next iteration. The specific ordering of body processing is not important. This constraint is only slightly more restrictive than the constraint present in the master-slave example. However, due to the tremendous increase in communication required by the partition exchanges, this problem is considered tightly coupled parallel.

2. The ptp_2.h file:

The ptp_2.h file is used to define include paths required by C/C++, to define global variables that would be common to any program using this example, to define MPI utility routines that would also be common to any program using this example, and to
3. **Compiling code based on these templates:**

Like many UNIX systems, Shale has a “make” utility for compiling programs. The *Makefile* for the templates is shown below.

```bash
CC = /usr/local/mpich/1.2.5.2/pgi/x86_64/bin/mpiCC
OFLAG=-O3
LIBP=-L/usr/lib64/ -L/usr/local/lib/ -
L/usr/local/mpich/1.2.5.2/pgi/x86_64/lib
LIBS=-lmpe
INCP=-I/usr/local/mpich/1.2.5.2/pgi/x86_64/include

ptp_2:
  $(CC) $@.cpp $(OFLAG) $(INCP) $(LIBP) $(LIBS) -o $@
```

However, “make” is a bit fussy and will not compile a program if a valid executable version currently exists in the current directory (talk about lazy!). Therefore, you have to remove the old executable each time before running “make”. However, if you are typing efficient (i.e. lazy) and don’t want to type all those commands to get something compiled, try this script (requires the above *Makefile*):

```bash
rm ptp_2
make ptp_2
```

Use an editor to type the above text into a file (I like to call mine “build”) and use `chmod` to make the file executable (`chmod 755 build`). You can then compile the *mst* and *mstSlave* programs simply by entering “./build” and the UNIX prompt.

4. **The batch submission script:**

I will assume that you will use the batch scheduler on Shale to submit your job and have provided a simple script to configure the scheduler to use the template:

```bash
#!/bin/bash
#PBS -S /bin/bash
#PBS -N ptp_2
#PBS -q long
#PBS -d /home/rmarsh/PTP/BIN/
#PBS -o ptp_2.o
#PBS -e ptp_2.e
#PBS -m abe
#PBS -l nodes=5:ppn=2

/usr/local/mpich/1.2.5.2/pgi/x86_64/bin/mpirun -np 10 ./ptp_2
```

In order, the items this script configures are:
37. “#PBS –S” Sets the execution shell to bash (can leave as is).
38. “#PBS –N” Sets the name of the job to “ptp_2” (you can change or leave as is).
39. “#PBS –q” Sends the job to the “long” queue (can leave as is).
40. “#PBS –d” Sets the working directory to “/home/rmarsh/PTP/BIN” (you must change this to reflect your directory organization).
41. “#PBS –o” Writes the output to a file (in the working directory) called “ptp_2.o” (you can change or leave as is).
42. “#PBS –e” Writes errors to a file (in the working directory) called “ptp_2.e” (you can change or leave as is).
43. “#PBS –m” Instructs the scheduler to send email when the job begins execution (b), terminates (e), or is aborted by the scheduler (a). You can use any combination of “abe.” If you do not want email sent, remove this line (change as desired).
44. “#PBS –l” Sets limits on the job execution. This example specifies 5 nodes with 2 processes per node. Note that there are many options. Do a “Man pbs_resources” for more info (see text box below).
45. “./ptp_2” Instructs the batch scheduler to run the example program. You must change this argument to specify your process.

In general, parallelizing a job across a number of nodes and processes per node is not as obvious as it may first seem. If you have a task that this example was designed to manage you would specify as many nodes as you have data partitions. However, specifying the processes per node is still not obvious. Here’s another heuristic: If your processing requires mainly computations and little IO, specify 2 processes per node (and reduce the number of nodes accordingly). If your processing involves a lot of IO, you may want to only specify 1 process per node.

5. Running a template derived program:

Once you have a batch submission script configured you can execute the derived program using the command:

```
qsub ./name_of_batch_script
```

Note that when using the batch scheduler no output is displayed on the computer screen, but is written to an output file (see “#PBS –o” and “#PBS –e” above).

To summarize, edit ptp_2.cpp according to your task, compile ptp_2.cpp using the Makefile (and optional “build” script) shown above, and then run the program using the batch scheduler.

Since this example gets the list of nodes to execute on from the batch scheduler each time it is executed, you can change the number of nodes or processes per node in the batch submission script and any derived program will adjust the load balancing accordingly. Remember, however, that the number of processes and the partitioning scheme must match.
6. Source code follows:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <unistd.h>
#include "mpi.h"

#define version    1.0
#define yes        1
#define no         0

#define Height     10    // Height of problem (no halo nodes required).
#define Width      4     // Width of problem (no halo nodes required).
#define cellHeight 1     // Height of partition (no halo nodes required).
#define cellWidth  4     // Width of partition (no halo nodes required).

#define Height     10    // Height of problem (no halo nodes required).
#define Width      4     // Width of problem (no halo nodes required).
#define cellHeight 1     // Height of partition (no halo nodes required).
#define cellWidth  4     // Width of partition (no halo nodes required).
```
/* array to hold the problem. */
/* */
/* Verbose is used to display log messages */
/* Turn on display with (verbose = yes) or */
/* turn off display with verbose = no). */
/* */
/* I would expect the user would want to */
/* modify this code to fit their application. */
*******************************************************************************/
double timeStep = 10.0;
double DATA[Height][Width];
double PARTITION[cellHeight][cellWidth];
double PARTITIONprime[cellHeight][cellWidth];
int verbose = yes;
*******************************************************************************/
/* SYSTEM GLOBAL VARIABLES */
/* */
/* !No changes should be made to these! */
/* */
*******************************************************************************/
int nodes;
int myID;
*******************************************************************************/
/* CLUSTER/MPI UTILITY ROUTINES */
*******************************************************************************/
/* INITIALIZE_MPI */
/* This routine does the following: */
/* 1. Calls the routines in proper order to */
/* to start MPI. */
/* 2. Returns the number of MPI nodes started. */
/* 3. Returns the rank (ID) of each MPI node. */
/* */
/* !No changes should be made to this routine! */
/* */
*******************************************************************************/
void INITIALIZE_MPI(int argc, char **argv) {
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nodes);
MPI_Comm_rank(MPI_COMM_WORLD, &myID);
if (myID == 0 && verbose == yes) {
printf("*************************************************
");
printf("* MESSAGE - Starting MPI deamons.               *
");
printf("*************************************************
");
}
}
*******************************************************************************/
/* SHUT_DOWN_MPI */
/* This routine does the following: */
/* 1. Shuts down MPI. */
/* */
void SHUT_DOWN_MPI(void) {
    if (myID == 0 && verbose == yes) {
        printf("*************************************************
        ");
        printf("* MESSAGE - Shutting down MPI deamons.          *
        ");
        printf("*************************************************\n") ;
    }
    MPI_Finalize();
}
#include "ptp_2.h"

void INITIALIZE_DATA(void) {
  int j, i;
  FILE *data;
  data = fopen("./particle.data", "r");
  for (j = 0; j < Height; j++) {
    for (i = 0; i < Width; i++) {
      fscanf(data, "%lf", &DATA[j][i]);
    }
  }
  fclose(data);
}

void EXTRACT_PARTITION(int myNode) {
  int i, j;
  for (j = 0; j < cellHeight; j++) {
    for (i = 0; i < cellWidth; i++) {
      PARTITION[j][i] = DATA[j+myNode*cellHeight][i];
    }
  }
}
void ACCUMULATE_PARTITION(int myNode) {
  int i, j;
  for (j = 0; j < cellHeight; j++) {
    for (i = 0; i < cellWidth; i++) {
      DATA[j+myNode*cellHeight][i] = PARTITION[j][i];
    }
  }
}

void PROCESS_PARTITION(void) {
  int j;
  double xAverage, yAverage;
  double xDistance, yDistance;
  double xVelocity, yVelocity;
  double xForce, yForce;
  // Find center of system.
  xAverage = 0.0;
  yAverage = 0.0;
  for (j = 0; j < Height; j++) {
    xAverage += DATA[j][0];
    yAverage += DATA[j][1];
  }
  // Adjust particle positions.
  for (j = 0; j < cellHeight; j++) {
    xDistance = PARTITION[j][0] - xAverage;
    yDistance = PARTITION[j][1] - yAverage;
    
    xForce = 1.0 / pow(xDistance, 3.0);
    yForce = 1.0 / pow(yDistance, 3.0);
    
    xVelocity = PARTITION[j][2] + xForce * timeStep;
    yVelocity = PARTITION[j][3] + yForce * timeStep;
    
    PARTITIONprime[j][0] = PARTITION[j][0] + xVelocity * timeStep;
    PARTITIONprime[j][1] = PARTITION[j][1] + yVelocity * timeStep;
    PARTITIONprime[j][2] = xVelocity;
    PARTITIONprime[j][3] = yVelocity;
  }
}
void UPDATE_PARTITION(void) {
    int    i, j;
    for (j = 0; j < cellHeight; j++) {
        for (i = 0; i < cellWidth; i++) {
            PARTITION[j][i] = PARTITIONprime[j][i];
        }
    }
}

double DIFFERENCE_DATA(void) {
    int j;
    double difference;
    difference = 0.0;
    for (j = 0; j < cellHeight; j++) {
        difference += fabs(PARTITION[j][0] - PARTITIONprime[j][0]);
        difference += fabs(PARTITION[j][1] - PARTITIONprime[j][1]);
    }
    return difference;
}

void MPI_LOAD_BALANCE(void) {
    // Define required parameters.
    char mpiCommand[256];
    int   startTime, stopTime;
    int   iteration;
}
double localDifference, newDifference, oldDifference;
double convergence;
MPI_Status status;

/***********************************************/
/* Define user specific parameters. */
/* */
/* It is expected that the user will replace */
/* these with their own variable declarations. */
/****************************************************/
int    j, n;
FILE   *log;

// Records the start time of the program.
startTime = time(NULL);

/***********************************************/
/* Initializes the user's data. */
/* */
/* It is expected that the user will replace */
/* this call with their own. */
/****************************************************/
if (myID == 0) INITIALIZE_DATA();
oldDifference = 0.0;
strcpy(mpiCommand, "go");

/***********************************************/
/* At this point we need to broadcast the */
/* original data to all nodes. */
/* */
/* Note that the broadcast call (MPI_Bcast) */
/* forces each node to stop and wait for the */
/* call completion - a form of synchronization.*/
/* */
/* It is expected that the user will replace */
/* this call with their own. */
/****************************************************/
MPI_Bcast(DATA, Height*Width, MPI_DOUBLE, 0, MPI_COMM_WORLD);

/***********************************************/
/* Here we start the data processing. Note in */
/* this template, we use an infinite loop to */
/* allow the processing to continue until we */
/* we obtain convergence. */
/****************************************************/
if (myID == 0 & verbose == yes) {
    printf("*************************************************
");
    printf("* MESSAGE - Processing data.                      *
");
    printf("*************************************************
");
}

// Initialize iteration counter;
iteration = 0;

// Start infinite loop for iterative data processing.
while (1) {


/***********************************************
/* At this point we need to have each node     */
/* extract their own partition from DATA and   */
/* processes their own local data.            */
/* We use MPI_Barrier to force synchronization */
/* all nodes are forced to stop and wait until */
/* all others nodes reach this point also.     */
/* It is expected that the user will replace   */
/* this call with their own.                  */
EXTRACT_PARTITION(myID);
PROCESS_PARTITION();
MPI_Barrier(MPI_COMM_WORLD);

/***********************************************
/* We will assume convergence occurs when no */
/* more changes in position occur.            */
/* First we calculate the sums local to each */
/* node.                                       */
/* It is expected that the user will replace   */
/* this call with their own.                  */
localDifference = DIFFERENCE_DATA();

/***********************************************
/* We now use MPI to acquire the sums local to */
/* the nodes and produce a single overall sum. */
/* Note that the reduce call (MPI_Reduce)      */
/* forces ALL nodes to stop and send the data  */
/* - a form of synchronization.               */
/* It is expected that the user will replace   */
/* this call with their own.                  */
MPI_Reduce(&localDifference, &newDifference, 1, MPI_DOUBLE,
MPI_SUM, 0, MPI_COMM_WORLD);

/***********************************************
/* Here we check for convergence to see if we */
/* are done processing.                        */
/* We have an infinite loop controlling the    */
/* processing, therefore we MUST have some     */
/* mechanism to terminate the programs.        */
/* It is expected that the user will replace   */
/* this code with their own.                   */
if (myID == 0) {
  convergence = fabs(newDifference - oldDifference);
  if (verbose == yes) {
    printf("Iteration [%d] convergence: %lf.\n", iteration,
convergence);
if (convergence < 0.00001 || iteration == 10)
strcpy(mpiCommand, "stop");
oldDifference = newDifference;
}
// Send all nodes a command to break or continue the infinite
loop.
MPI_Bcast(mpiCommand, 5, MPI_CHARACTER, 0, MPI_COMM_WORLD);
if (!strcmp(mpiCommand, "stop")) break;
// Update iteration counter.
iteration++;

/***********************************************/
/* Call the routine to update PARTITION with */
/* the results contained in PARTITIONprime. */
/* */
/* It is expected that the user will replace */
/* this call with their own. */
/***********************************************/
UPDATE_PARTITION();

/***********************************************/
/* We will now have all of the nodes send their*/
/* partition to node 0. Node 0 will then */
/* recreate DATA, and broadcast the updated */
/* DATA to everyone for the next iteration. */
/* */
/* It is expected that the user will replace */
/* this call with their own. */
/***********************************************/
MPI_Send(PARTITION, cellHeight*cellWidth, MPI_DOUBLE, 0, 0,
MPI_COMM_WORLD);
if (myID == 0) {
  for (n = 0; n < nodes; n++) {
    MPI_Recv(PARTITION, cellHeight*cellWidth, MPI_DOUBLE, n,
0, MPI_COMM_WORLD, &status);
    ACCUMULATE_PARTITION(n);
  }
}
MPI_Barrier(MPI_COMM_WORLD);
MPI_Bcast(DATA, Height*Width, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}

/***********************************************/
/* At this point we will log the results to a */
/* file by having each node send its partition */
/* to node 0, which will then log the results. */
/***********************************************/
MPI_Send(PARTITION, cellHeight*cellWidth, MPI_DOUBLE, 0, 0,
MPI_COMM_WORLD);
if (myID == 0) {
  for (n = 0; n < nodes; n++) {
    MPI_Recv(PARTITION, cellHeight*cellWidth, MPI_DOUBLE, n, 0,
MPI_COMM_WORLD, &status);
    ACCUMULATE_PARTITION(n);
  }
}
// Write results to log file.
log = fopen("./log", "w");
for (j = 0; j < Height; j++) {
    fprintf(log, "<%6.2lf %6.2lf>\n", DATA[j][0], DATA[j][1]);
}
fclose(log);

// Records the stopping time of the program.
stopTime = time(NULL);

// Display the program's running time.
if (myID == 0 && verbose == yes) {
    printf("Processing time: %d.\n", (stopTime - startTime));
}

/***********************************************************/
/* MAIN.                                                   */
/***********************************************************/
/***********************************************/
/* The MAIN routine initializes the system and */
/* starts the processing. */
/* There is nothing for the user to to change */
/* in this routine. */
/***********************************************/
int main(int argc, char **argv) {

    // Initialize MPI.
    INITIALIZE_MPI(argc, argv);

    // Process data.
    MPI_LOAD_BALANCE();

    // Shut down.
    SHUT_DOWN_MPI();
    return 1;
}