While building a Beowulf cluster presents some interesting technical challenges, the only reason to build such a cluster is to gain the "horsepower" required to solve some large computational problem or perform some repetitive processing task in a practical amount of time. As such, the requirements of your software and model should always dictate your choice of hardware, node interconnects, and hardware and software configuration. Indeed, the correct answer to nearly every cluster design question is, "It depends on your application."

Having said that, it's equally important to understand the limitations of your hardware when writing parallel programs. Coders have many choices to make when designing data structures and message passing methods. Ultimately, however, these choices are constrained by the needs of your algorithms and the limits of your hardware, including communications bandwidth and latency and free memory.

Unfortunately, the most efficient implementation -- the one that minimizes memory utilization and provides optimal communication -- is often the most difficult to implement. Typically, a balance must be struck between redundant calculations and data storage and communications between distributed memory nodes. In all cases, tradeoffs must be made no matter what kind of parallel computer is used -- the balance is merely shifted somewhat by the various strengths and weaknesses of each architecture.

**Decomposition and Message Passing**

Parallel computing divides a large computational problem into smaller tasks that can be performed simultaneously by multiple computer processors. Each processor performs some portion of the calculation, or more often, performs all of the calculations on a subset of the data. In either case, communication among processors -- called message passing -- coordinates the combined effort and is required to complete the full computation.

While all computational problems can't be solved in parallel, many physical and biological science simulations can be cast into a parallel framework. Some tasks, like image processing, tend to be relatively simple to perform in parallel since each pixel can be operated on independently of the others. These sorts of "coarse grained" problems are generally the
easiest to implement on a distributed memory cluster, and are often called "embarrassingly parallel." Other types of problems are more challenging and may require communications during calculations because of dependencies between domains.

In the past, this column has covered some of these parallel programming issues and has provided a basic introduction to parallel programming using the Parallel Virtual Machine (PVM, http://www.epm.ornl.gov/pvm) and the Message Passing Interface (MPI) application programming interfaces (APIs). (See the March, April, and May 2002 columns at http://www.linuxmagazine.com/2002-{03,04,05}/extreme_01.html, respectively). This month, let's continue our look at parallel programming by developing some more concrete examples using additional MPI routines. The examples contained herein were tested using MPICH (MPI Chameleon) developed at Argonne National Laboratory (available at http://www-unix.mcs.anl.gov/mpi/mpich), but should work with any MPI implementation.

Master/Slave Algorithms: Dealing the Cards

Many coarse-grained applications can be implemented in a master/slave framework, where one process (usually the first process), the master process, doles out work to all of the other processes, called slaves. A single process or thread is usually associated with one real physical processor, but this is not a requirement. For generality, this discussion will refer to processes instead of processors or nodes.

In a master/slave application the problem is decomposed into reasonably sized chunks of data (called cards below) that can be processed independently by many processes or threads simultaneously. Image processing, remote sensing, and pattern and landscape analysis applications are strong candidates for this kind of parallelism.

In such codes, the master process usually reads in parameters and an image or map that's subsequently sent to all the remaining processes. If the image or map is decomposed, only the relevant pieces are sent to each slave process. If many operations are to be performed with the map, all processes may receive the full map up front and then perform the relevant operations across the entire map as assigned.

Next, the master generally assigns work to each slave process in turn, like a card dealer in a poker game. The master then collects and saves the results as they're returned from the slaves. As each slave process completes the work it's assigned, it requests additional work (or cards) until the master signals that no other work remains.

This rather straightforward strategy is particularly effective because it can automatically provide a kind of dynamic load balancing capability. That means the available capacity of the cluster is used more efficiently. As long as the work can be broken up into many more cards than available processes, each slave process will have the opportunity to work on multiple cards.

For example, in a heterogeneous cluster (one with nodes of varying speed
or capabilities) or a shared cluster (one with multiple jobs sharing the same node), the processes on faster or less-busy nodes automatically perform more of the work since they can finish each card more quickly. If a node suddenly becomes loaded down with another job, it requests cards less frequently. Thanks to the independence of the computing tasks, this "card dealing" strategy provides load balancing for free.

An Example Implementation

Listing One, contains a C program, named model.c, that demonstrates one possible implementation of this master/slave algorithm using MPI. This code performs some kind of categorical analysis on a map, where the number of categories is usually larger than the number of slave processes. Each slave node is given a single category to analyze in a map which is completely in memory for every process. It's presumed that that map's been read by the master process and sent to all other processes. A comment is included in the code where this would occur. Due to space limitations, error checks have been omitted.

Listing One: model.c, a sample master/slave implementation - Part 1

```c
#include <stdio.h>
#include "mpi.h"

#define HIT_ME_TAG        1
#define ASSIGNMENT_TAG    2
#define RESULTS_TAG       3
#define RESULTS_SIZE      100000
#define NUM_CATEGORIES    100

void analyze_category(unsigned long cat, double *results)
{
    /* Some very-time consuming pattern analysis for category 'cat' */
    .
    .
    return;
}

void save_results(unsigned long cat, double *results)
{
    /* Save off the results to memory or disk or screen */
    .
    .
    return;
}

int main(int argc, char** argv)
{
    int rank, num_proc, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int p, proc, dummy, goodbyes;
    unsigned long num_cat, cats_done, *cat_assign;
    double results[RESULTS_SIZE];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```
MPI_Comm_size(MPI_COMM_WORLD, &num_proc);
MPI_Get_processor_name(processor_name, &namelen);

printf("Program starting on rank %d of %d on %s\n", rank, num_proc, processor_name);

num_cat = NUM_CATEGORIES;
/* Broadcast of parameters and global data to slave processes */

/* Begin processing */
if (rank) {
  /* SLAVE PROCESS CODE */
  for (cat = 0; cat != -1;) {
    printf("%d: Sending HIT_ME request to master\n", rank);
    MPI_Send(&rank, 1, MPI_INT, 0, HIT_ME_TAG, MPI_COMM_WORLD);
    MPI_Recv(&cat, 1, MPI_UNSIGNED_LONG, 0, ASSIGNMENT_TAG, MPI_COMM_WORLD, &status);
    if (cat != -1) {
      printf("%d: Received assignment of category %ld\n", rank, cat);
      analyze_category(cat, results);
      printf("%d: Sending results from category %ld to master\n", rank, cat);
      MPI_Send(&results, RESULTS_SIZE, MPI_DOUBLE, 0, RESULTS_TAG, MPI_COMM_WORLD);
    }
    else
      printf("%d: Received die signal from master\n", rank);
  }
}
else {
  /* MASTER PROCESS CODE */
  cat_assign = (unsigned long *)malloc(num_proc * sizeof(unsigned long));
  for (cat = goodbye = 0; goodbye < num_proc - 1;) {
    MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    proc = status.MPI_SOURCE;
    if (status.MPI_TAG == HIT_ME_TAG) {
      MPI_Recv(&dummy, 1, MPI_INT, proc, HIT_ME_TAG, MPI_COMM_WORLD, &status);
      MPI_Send(&cat, 1, MPI_UNSIGNED_LONG, proc, ASSIGNMENT_TAG, MPI_COMM_WORLD);
      printf("%d: Assigning category %ld to process %d\n", rank, cat, proc);
      cat_assign[proc] = cat;
      if (cat != -1) {
        cat++;
        if (cat == num_cat) cat = -1;
      }
      else
        goodbye++;
    }
    else {
      MPI_Recv(&results, RESULTS_SIZE, MPI_DOUBLE, proc, RESULTS_TAG, MPI_COMM_WORLD, &status);
      save_results(cat_assign[proc], results);
    }
  }
}
printf("%d: Calling MPI_Finalize()\n", rank);
MPI_Finalize();
return 0;
The `main()` routine begins, as usual, by initializing MPI with `MPI_Init()`, discovering the process rank with `MPI_Comm_rank()`, obtaining the number of processes participating in the computation with `MPI_Comm_size()`, and getting the processor name on which the process is running (usually the node host name). At the end of the code, all processes call `MPI_Finalize()` before exiting.

Below the `/* Begin processing */` comment, the code branches: the slave code is executed if `rank` is greater than zero, otherwise the master code is executed (for `rank` equal to zero). The slave processes enter a loop that checks the value of `cat`. Inside this loop, the slave processes send their `rank` to the master using the `HIT_ME_TAG` using `MPI_Send()`. They then call `MPI_Recv()` with an `ASSIGNMENT_TAG` to receive a category assignment. When the master runs out of work (or cards) to be performed, it sends the slaves a category value of `-1`.

If the category value is not `-1`, the slaves analyze the category in question by calling `analyze_category()` (which does nothing in the example provided). Then the slaves send the results of the analysis (stored in a double precision floating point array called `results`) to the master using `MPI_Send()` with the `RESULTS_TAG`. The loop repeats until the category value of `-1` is received from the master.

`MPI_Send()` and `MPI_Recv()` are blocking routines. They do not return until the message has been sent or received, and relevant message buffers may be safely manipulated. Non-blocking routines are also available in MPI, but in this case we want the program not to continue until the communication is complete, so the blocking routines are best.

The master code is a little more complicated. First, the master creates an array of length `num_proc` to track the categories assigned to each slave process. Next, the master enters a loop where it deals out the categories and receives results. This loop exits only after all the slave processes have been signaled to terminate.

Inside the loop, the master calls `MPI_Probe()` to check for an incoming message without actually receiving it. The message can originate from any process (`MPI_ANY_SOURCE`) with any tag (`MPI_ANY_TAG`). Upon return, `status` contains information about the message, including the originating process (stored in `status.MPI_SOURCE`), the message tag (stored in `status.MPI_TAG`), and the size of the message (obtained by calling `MPI_Get_count()`). The rank of the message originator is stored in `proc`, and then the code branches depending on what tag the message carries.

If the message tag is `HIT_ME_TAG`, the slave is requesting more work. Here, the master receives the "hit me" message using `MPI_Recv()` (storing the
message in the dummy variable since the process rank is already known), and sends a new assignment to the slave using MPI_Send() with the ASSIGNMENT_TAG. The master then records the category assignment in the cat_assign[] array for use later.

If cat is not already set to -1 (that is, if all the categories have not yet been assigned), the master bumps the category number (cat), and checks to see if it's the last category to be processed. If it's the last one (if cat equals num_cat), then cat is set to -1. If, on the other hand, cat was already set to -1 (and it was just sent to the slave), then the goodbyes counter is bumped.

If the message tag is something other than HIT_ME_TAG, the message must contain results so the master receives results with MPI_Recv() using the RESULTS_TAG. The master then saves the results by passing them to save_results() (which does nothing in this example). The loop exits only after all the slave processes have been signaled that no more cards of work remain (goodbyes equals num_proc - 1) which can only occur after all results have been received.

This algorithm is fairly efficient, and provides automatic dynamic load balancing. However, it does require the slaves to send tiny "hit me" messages to the master to receive work assignments. Sending small messages is relatively expensive since all the time is attributable to the latency of the network or interconnect. Tiny messages take just as much time to send and receive as medium-sized messages.

As a rule, unnecessary communication should be avoided, and this same algorithm can be implemented in a slightly different way, eliminating the need for "hit me" messages. Listing Two contains an improved main() section for model.c that has no "hit me" communications at all.

Listing Two: Improved model.c main() with no "hit me" communication - Part 1

```c
int main(int argc, char** argv)
{
    int rank, num_proc, namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int p, proc;
    unsigned long num_cat, cat, cats_done, *cat_assign;
    double results[RESULTS_SIZE];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_proc);
    MPI_Get_processor_name(processor_name, &namelen);

    printf("Program starting on rank %d of %d on %s\n", rank, num_proc,
           processor_name);
    num_cat = NUM_CATEGORIES;

    /* Broadcast of parameters and global data to slave processes */
```
After initialization, the slave processes enter their work loop and post an `MPI_Recv()` immediately to wait for a category assignment. As before, they perform the analysis by calling
analyze_category() and then send the results back to the master with MPI_Send(). They exit the loop once a category of assignment of -1 has been received.

The master code now contains two loops: one loop for handing out initial assignments, and a second loop for receiving results and doling out the rest of the categories. As before, the master allocates memory for the cat_assign[] array to track assignments. Then it enters the first loop where it deals out categories in turn, ensuring that each process receives one assignment (except for process 0, the master) whether it's a real work assignment or just a -1 signal to quit. If the number of categories is less than the number of available slave processes, only the first num_cat processes are assigned real work.

After initial assignments are sent, the master enters the second loop. In this improved code, the master receives only results, so it posts an MPI_Recv() for results from any process (MPI_ANY_SOURCE) using RESULTS_TAG. Once a slave sends results, the master saves them with save_results() and bumps the cats_done counter. Next, the master sends that same slave another category to work on or a -1 if all categories have been assigned. The rest proceeds as in the first loop. This second loop exits once results have been received for each category (when cats_done equals num_cat).

While a limited number of computations fit this model of parallelism, the example code provided here serves as a good framework for implementing a card-dealing strategy for those that do. The dynamic load balancing implicit in the algorithm ensures maximum utilization of limited computing resources. These codes can also be used to test your network performance: simply alter the values of RESULTS_SIZE and NUM_CATEGORIES to see the impact of passing a varying number of messages of different sizes.

Thinking up creative parallel computing strategies can be fun and implementing them is always challenging. More tips and tricks for parallel programming will appear in future columns.

Forrest Hoffman is a computer modeling and simulation researcher at Oak Ridge National Laboratory. He can be reached at forrest@climate.ornl.gov.