Running Star-P on Beowulf

This guide is meant to help people who already have Star-P running on a cluster or shared-memory computer but are not yet familiar with the program. This guide assumes that the user is running Star-P on our Beowulf cluster or on another set-up similar to it.

1. Starting the Program

This section will cover allocating resources necessary for running Star-P as well as starting the actual program.

Allocating Resources

Before running Star-P, you must first allocate the appropriate number of processors that Star-P will be using. You do this with the following command:

/share/getresv <n>

where <n> is the number of nodes that you wish to allocate. On Beowulf, each node has two processors so allocating <n> nodes will actually allocate <2*n> processors.

After you run this command, the following should appear:

qsub: waiting for job 7703.beowulf.lcs.mit.edu to start
qsub: job 7703.beowulf.lcs.mit.edu ready
[username@compute-0-1 username]$

This means that you successfully allocated the number of nodes that you requested. **Job 7703.beowulf.lcs.mit.edu** is your job identifier. The last line is the new command prompt that should appear. Remember that if your command prompt looks like this, then you are still taking up those nodes. If you do not need them, then you should release them by typing in **exit**.

Starting Star-P

After you allocate the nodes, just type in the following command to start Star-P:

/share/startmp

That should start Star-P. After it is done, you will get a Matlab prompt. The startmp script puts you in your home directory by default; you can set the path on all nodes by using mmpath().

2. Understanding Parallel Structures

This section will help you understand how parallel structures work and what the difference is between mm mode and 'normal' parallel mode.

Parallel Structures and *p

Star-P uses the same commands as Matlab to create and manipulate matrices with a few small differences. For example in Matlab, to create a 100 x 100 matrix, you type:

A = randn(100, 100);

In Star-P, to create a 100 x 100 distributed matrix, you type:

A = randn(100*p, 100);

As you can see, the only difference is the *p variable that we add after one of the dimensions of the matrix. This tells Star-P to create a distributed matrix instead of a non-distributed one.

In Star-P, putting the *p after different dimensions creates matrices that are distributed across the processors in different ways. If we place the *p after the first dimension like so:

A = randn(100*p, 100);

Then we are creating a row-distributed matrix. Each processor contains a chunk of the matrix. Each of these chunks are adjacent rows from the matrix. For example, with 4 processors, this command would spawn a matrix with 25 rows, each with length 100 on each of the four processors. Processor 0 would hold the first 25 rows. Processor 1 would hold the next 25 rows and so on. If the number of rows does not divide evenly by the number of processor, the number of rows per processor is rounded up and the last row is short. For example, distributing 14 rows among four processors would yield four rows per processor and two rows for the last processor.

If we did the following:

A = randn(100, 100*p);

Then we are creating a column-distributed matrix. Again each processor contains a chunk of the matrix. Each of these chunks are now adjacent columns from the matrix. For example, with 4 processors, this command would spawn a matrix with 25 columns, each with length 100 on each of the four processors. Processor 0 would hold the first 25 columns. Processor 1 would hold the next 25 columns and so on. The behavior for uneven distributions is the same as for row-distributed matrices.

Lastly, there is the block-cyclic distributed matrix.

A = randn(100*p, 100*p);

This format distributes the data as a series of smaller blocks across the processors. If you are really interested in learning more about block-cyclic distributed matrices, read the Star-P manual.

Each of these three formats has their advantages and disadvantages. Take the time to try out different distributions and see if the speed of the problem increases or decreases.

Viewing the Matrix

In Matlab, typing in the name of the matrix returns the contents of the matrix. However, in Star-P, typing in the name of the matrix yields different results:

A
A = ddense object: 100-by-100

The answer is telling you that A is a distributed object of size 100 by 100. In order to view the entire matrix, type in the following:

A(:)

This brings the entire matrix to the front-end machine and then displays it as Matlab would. Remember, that means the result of **A(:)** is no longer distributed!

Like Matlab, you can also access individual parts of the matrix in Star-P. Star-P brings the requested parts of the matrix to the front-end and displays them. For example, the following code snippets would all behave identically to Matlab:

A(1,1) *returns the element at 1,1 of the matrix*
A(:,1) *returns the first column of the matrix*
A(2:6, 1) *returns all elements between 2,1 and 6,1 inclusive of the matrix*
A(3, :) *returns the third row of the matrix*

Again, it is important to remember that the result of these commands is on the front-end so they are not distributed even though A remains a distributed matrix.

If you are running Matlab*p with X support, you can use ppspy() to view distributed matrices graphically.

3. Writing and Running Code

This section covers writing code in Star-P. It is remarkable similar to writing code in Matlab.

Writing Parallel Code

At the time of this guide, many functions in Star-P are overloaded Matlab functions. This means that they will work on either distributed matrices or non-distributed matrices with the same result. For example, the following snippet of code results in the same answer on both Matlab and Star-P

```
B = sin(A);
X = A\B;
norm(A*X-B)
```

It's that simple. If you get an error message, then it might mean that the function has not yet been overloaded. Check the appendix in the Star-P manual for a list of overloaded functions.

MM Mode

If you need to use a function in parallel that hasn't been overloaded, then you can also use mm mode. MM mode runs the function that it takes as an argument independently over each processor. This is similar to how mpi code runs over multiple processors (If you don't know MPI, that's okay). For example:

```
A = randn(100, 100*p);
B = mm('fft', A);
```

The preceding code would cause each processor to run the fft function over the columns that it is holding. If there were four processors, processor 0 would run the fft function over the first 25 columns. Processor 1 would run the fft function over the next 25 columns and so on. Since the fft function is only dependent on the data in a single column, this code dramatically speeds up the fft function.

However, there are two major caveats to using mm mode. The first caveat is that the distribution of the data does matter. Since each processor is running the function independently, running the same code over two different distributions (row, column, or block-cyclic) could yield dramatically different results. For example, using something similar to the example above:

```
A = randn(100*p, 100);
B = mm('fft', A);
```

While this looks almost identical to the code from before, since the matrix is row distributed then the answer will be completely different (and in most situations, not what you are looking for). This is because each processor holds 25 rows a piece. The fft function goes down the columns and only sees columns of length 25 and calculates the fft on those.

Another caveat is that there is no communication between the processors so merely using mm mode on some function to make it parallel may not result in the correct answer. For example:

```
A = randn(100, 100*p);
```

```
B = mm('fft2', A);
```

This code calculates the two-dimensional fft on the matrix A. However, in actuality it only calculates the two dimensional fft on each of the chunks of columns that each processor contains. Since fft2 relies on the entire matrix for an answer, running mm mode on fft2 will not result in the correct answer.

There are several other potential issues with MM mode. First, it only works when the return values of all processors are the same size (this is documented in 'help mm'). This is usually only a problem when the number of rows/columns does not divide evenly by the number of processors, resulting in a smaller chunk on the last processor. The only solution to this problem is to make all the return values the same size, possibly by adding zeros onto the last chunk.

Second, MM mode only returns the real parts of complex numbers. For example, mm('eig',A) will only give you the real parts of the eigenvalues. To solve this problem, you can create a function to return the real and imaginary parts of the answer in separate return matrices. To do this, declare a function like this:

```
function [re,im] = foo(A)
```

You can return the real part of your output in re and the imaginary part in im. Running mm('foo',A) will then return both parts, and you can put the data back together with something like this:

```
B = mm('foo',A);
C = B(:,1) + B(:,2)*i;
```

It is also possible to use MM mode to execute different pieces of code on each processor. To do this, declare a row vector i as follows:

```
id = 1:np*p;
```

On each processor, the local chunk of this variable is the processor ID. Now if this variable is passed into a function, you can do something like this:

```
if id==1
        %Do something on processor 1
elseif id==2
        %Do something on processor 2
```

and so on.

Finally, note that there is also MO mode, for Multi-Octave mode. This uses multiple instances of Octave rather than Matlab, which has all the same numerical functions, although sometimes with slightly different numerical output. To use MO mode simply substitute mo() for mm() – octave can run the exact same .m files.

Running Scripts

You run scripts on Star-P the same way as you run scripts on Matlab. You write scripts on Star-P the same way as you write scripts on Matlab taking in consideration the differences from the sections above.

4. Conclusion

This concludes the quick start guide for Star-P on Beowulf. Don't be afraid to experiment with different functions and distributions. You can always compare your final answer by running the same program in serial mode on Matlab.

Lastly, one final caveat: **Do NOT use the variable p for anything except *p in creating distributed matrices.**