Star-P's Power Tools for Parallelism

Table of Contents

Executive Overview............................................................................................................................2
Section 1 – Steps to Creating Parallel Code using Star-P's Tools............................................................3
  First Draw up Blueprints: Block Up Your Code..............................................................................3
  Measure Your Code: Star-P's Profiling Tools. ................................................................................4
    What Functions are Called? “profile” and “ppprofile”.................................................................4
    How Much Time Does My Function Take? “tic/toc” and “pptic/pptoc”........................................6
    What Are My Variables? “whos” and “ppwhos”.........................................................................7
    Craftsman’s advice:........................................................................................................................8
  Is Your Problem Data-Parallel or Task-Parallel?............................................................................8
  Overcoming Parallelization Challenges..........................................................................................11
    Missing Functions or Unsupported MATLAB Toolboxes...........................................................11
    Dealing with Structs and Cell Arrays............................................................................................12
  Vectorize Your Code! .......................................................................................................................13
  The Big Step: Parallelizing your code..............................................................................................15
Section 2 – Tool Tips for Writing Parallel Code ....................................................................................17
  Matrix Types: Dense or Sparse? ........................................................................................................17
  Row or Column: What's the Best Distribution for Parallel Processing?........................................21
    Optimal Distributions for Star-P Built-ins.................................................................................21
    Optimal Distributions for Third-Party Libraries.........................................................................23
    Redistributing Matrix or Array Data...........................................................................................23
  Polishing Your Code: Tool-tips for “ppeval”...............................................................................24
    Performance Tip: Vectorize “for” Loops Inside of “ppeval” Calls..................................................24
    Performance Tip: Beware of Trailing Singleton Dimensions.......................................................25
    Performance Tip: Assignments to Structs can be Slow.................................................................26
    Performance Tip: Avoid ND Indexing!............................................................................................27
  Hardware considerations................................................................................................................28
    Network Latency..........................................................................................................................28
  Cluster vs. Shared Memory Supercomputers...................................................................................29
  Sharpen Your Tools: Star-P Settings for Maximum Performance...................................................30
Section 3 – Parallelization Patterns and Examples............................................................................31
  Data-parallel: Crunching Large Matrices......................................................................................31
  Data-parallel: Interacting Bodies and Swarms..............................................................................32
  Data-parallel: Reduction and Scan Operations...............................................................................36
  Task-parallel: Unrolling a Loop – Generating the Mandelbrot Set..............................................38
  Task-parallel: Monte Carlo and other statistical algorithms.........................................................42
The Right Tool For the Job: Parallelization Using Star-P..................................................................44
Appendix: Online Star-P Resources..................................................................................................45
About Interactive Supercomputing....................................................................................................46

Copyright (C) 2007 Interactive Supercomputing, inc.
Executive Overview

Classroom training in parallel programming is often like trying to teach somebody how to ride a bicycle by deriving Newton's equations for a two-wheeled vehicle. Good programming is really only learned by doing, and parallel programming has traditionally been the domain of specialists who have perfected their craft through practice and experience. However, with the advent of Interactive Supercomputing's Star-P parallelization system, you – the non-specialist – can now easily create MATLAB programs which will run on a parallel computer. In this light, the purpose of this document is three-fold:

1. Walk through the steps involved in parallelizing MATLAB code under Star-P (shown on the right)

2. Provide you a bag of tips for using Star-P's tools to achieve the best performance from your code.

3. Present some example domain-specific code for you to experiment with on your own. The Star-P parallelization patterns shown in these codes can serve as starting points for your own program.

This document assumes that you are already somewhat familiar with Star-P, perhaps through an introductory hands-on experience, or by having read through the “Star-P Test Drive”. It is meant to help you in one of two situations:

1. You have a pre-existing body of MATLAB code which you’d like to run on a parallel supercomputer using Interactive Supercomputing's Star-P software.

2. You are preparing to write some new MATLAB code, and you want to be able to run it under Star-P.

When you are done reading this document you should have sufficient understanding of Star-P – and parallel computing in general – that you can become an effective parallel programmer using the tools provided by Interactive Supercomputing's Star-P platform.

Figure 1: Steps for parallelization using Star-P.
Section 1 – Steps to Creating Parallel Code using Star-P's Tools

You're a researcher with some MATLAB code you want to run on a supercomputer. Or maybe you're about to write a new program, and would like to ensure that your code is “parallel-ready”. In either case, Star-P presents a chest full of sleek, shiny power tools ready for you to create a parallel program! However, you may be asking: “How do I know what to do? Is there a procedure for using these parallelization tools? And is there a best tool for my particular job?”

Life would be easier if there was one single tool which you could use to make your program run on a parallel machine. Unfortunately, no such Swiss Army Knife exists. The reason is that computational problems are all different – a method which efficiently parallelizes one kind of computation won't necessarily work for another one. After all, you can't drive nails into a board with a screwdriver! However, there are some commonly applied techniques which you can apply in a step-by-step procedure when parallelizing your code. If you're relatively new to Star-P and want to learn some craftsman's techniques, put on your safety goggles, and we'll guide you through the steps to use when building a parallel program using Star-P's tools!

First Draw up Blueprints: Block Up Your Code

An architect planning a new building must envision how it is constructed from individual pieces like bricks, boards, shingles, and stones. Similarly, before you begin parallelizing your code, you need to think about how it may be broken into individual compute steps. This is illustrated in the figure presented below. A compute step might be a single operation, or it might be an interdependent series of calculations. Once you have identified your program's individual steps, you can use the appropriate tools to parallelize them (or leave them alone). We call this step “blocking up your code”; it is shown schematically in Figure 2.

Here are some things to look for when blocking your code:

- For each proposed block, can you envision how parallel computation might speed up the computation?
- For each proposed block, what are the inputs and outputs? Do you create new matrices in this step? Or just manipulate old ones? Are the results of your calculation larger or smaller than the inputs? This will help you decide where your
data should live.

- How will the data move between client and server as your computation progresses? At Interactive Supercomputing we say, "follow the data", meaning that you should do your best to understand where your variables live (on the client or on the server), and where they are being processed. "Following the data" is important since optimizing your program's performance requires that your variables live in the best place for parallel processing.

Later, when you actually introduce parallelization syntax, understanding your program's structure as a series of logical blocks will help you identify the best technique to parallelize each block of code.

Craftsman's advice:
- Visualize your code's logical blocks in preparation for parallelizing it.

Measure Your Code: Star-P's Profiling Tools.

Just as every builder needs a tape measure to do his job, a parallel programmer needs code profiling tools. The reason is that if your code has sections or specific sub-functions which take significant time to execute, those are the best places for you to focus your parallelization efforts. Frequently, the only way to find those bottlenecks is to measure your code! For that reason, both MATLAB and Star-P offer a number of profiling tools for use in taking the measure of your code.

What Functions are Called? “profile” and “pprofile”

Prior to parallelizing an existing program using Star-P, it is important to gather a few clues about how it works, and see what functions are called frequently. Besides studying the code manually, running the code with MATLAB's profiler often provides insights into what the program is doing. Here are some things to look for when running MATLAB's “profile” on a new or unfamiliar MATLAB program:

- Look for functions with large vector or matrix arguments which take a long time to execute. These functions are easy to speed up by making sure that they operate on server variables. That is, a matrix operation which takes lots of time running on the client is a obvious candidate for execution on the parallel server.
- Look for functions which get called many times. If you find a function which is
invoked a large number of times, there two possible explanations:

1. First, multiple invocations of the same function suggest that your function might need to be vectorized. That is, rather than repeatedly invoking the same function on individual pieces of data, you may be better off feeding the function with all the data grouped together into a vector or matrix.

2. Another reason why a function might be invoked many times is that your program implements some type of "task parallelism". That is, your program invokes the same function many times with many different, independent datasets. In this case, your function can be executed using Star-P's facility for task-parallelism, "ppeval".

Later, once you have parallelized your program using Star-P, you can return to profiling to using Star-P's analog to "profile" called "ppprofile". Like MATLAB's "profile", Star-P's "ppprofile" gathers and presents statistics about all the functions invoked by your program during execution. As you might expect, "ppprofile" concentrates on functions executed on the parallel server, including calls to the various low-level parallel routines invoked by your program. Here's an example "ppprofile" run:

```matlab
>> ppprofile on
>> x = rand(100, 100*p);
>> y = zeros(100, 100*p);
>> for n = 1:100
>>   for m = 1:100
>>     y(n, m) = 1 + sin(x(n, m)*pi/6);
>>   end
>> end
>> z = inv(y);
>> w = eig(z);
>> ppprofile report
```

<table>
<thead>
<tr>
<th>function</th>
<th>calls</th>
<th>time</th>
<th>avg time</th>
<th>%calls</th>
<th>%time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ppdense_setelement</td>
<td>10000</td>
<td>136.5445</td>
<td>0.013654</td>
<td>49.97</td>
<td>49.5558</td>
</tr>
<tr>
<td>ppdense_viewelement</td>
<td>10000</td>
<td>135.9923</td>
<td>0.013599</td>
<td>49.97</td>
<td>49.3554</td>
</tr>
<tr>
<td>ppdense_rand</td>
<td>1</td>
<td>1.9965</td>
<td>1.9965</td>
<td>0.004997</td>
<td>0.7246</td>
</tr>
<tr>
<td>ppscalapack_eig</td>
<td>1</td>
<td>0.63034</td>
<td>0.63034</td>
<td>0.004997</td>
<td>0.22877</td>
</tr>
<tr>
<td>pppblas_gemm</td>
<td>1</td>
<td>0.11945</td>
<td>0.11945</td>
<td>0.004997</td>
<td>0.043351</td>
</tr>
<tr>
<td>ppdense_diag</td>
<td>1</td>
<td>0.080013</td>
<td>0.080013</td>
<td>0.004997</td>
<td>0.029039</td>
</tr>
<tr>
<td>ppdense_sumv</td>
<td>1</td>
<td>0.05649</td>
<td>0.05649</td>
<td>0.004997</td>
<td>0.020502</td>
</tr>
<tr>
<td>ppdense_binary_op</td>
<td>1</td>
<td>0.02606</td>
<td>0.02606</td>
<td>0.004997</td>
<td>0.0094579</td>
</tr>
<tr>
<td>ppdense_transpose</td>
<td>2</td>
<td>0.025164</td>
<td>0.012582</td>
<td>0.009994</td>
<td>0.0091327</td>
</tr>
<tr>
<td>ppscalapack_inv</td>
<td>1</td>
<td>0.020696</td>
<td>0.020696</td>
<td>0.004997</td>
<td>0.0075111</td>
</tr>
<tr>
<td>ppbase_changeDist</td>
<td>1</td>
<td>0.020171</td>
<td>0.020171</td>
<td>0.004997</td>
<td>0.0073206</td>
</tr>
<tr>
<td>ppbase_createMatrixCopyRe</td>
<td>1</td>
<td>0.015522</td>
<td>0.015522</td>
<td>0.004997</td>
<td>0.0056334</td>
</tr>
<tr>
<td>ppdense_zeros</td>
<td>1</td>
<td>0.009712</td>
<td>0.009712</td>
<td>0.004997</td>
<td>0.0035248</td>
</tr>
<tr>
<td>Total</td>
<td>20012</td>
<td>275.537</td>
<td>0.013769</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

Just like MATLAB's "profile on", the Star-P command "ppprofile on" turns on profiling, and "ppprofile report" returns the profiling report. ("Ppprofile clear" clears the profiling information.) In the example above, we have run a particularly sluggish program – it's sluggish because it uses a "for" loop to index through a matrix. As you can see, 99.9% of the execution time is chewed by Star-P's low-level routines "ppdense_setelement" and "ppdense_viewelement". These are the functions used to transfer scalar values between the client and the server. They are called because the individual elements of x and y are
transferred between the client and the server with each iteration of the double loop. Time spent in the other functions represents a minuscule percentage of the total execution time. When you see this kind of result when profiling your code, it signals you that a bottleneck exists. In this case, the problem is that we need to do away with the double “for” loops and vectorize the code.

Here’s the same calculation performed using vectorized constructs:

```matlab
>> ppprofile on
>> x = rand(100, 100*p);
>> y = ones(100, 100*p) + sin(x*pi/6);
>> z = inv(y);
>> w = eig(z);
>> ppprofile report
function                 calls      time  avg time    %calls     %time
ppdense_rand                 1   0.73145   0.73145      6.25   38.4985
ppbase_changeDist            1   0.57641   0.57641      6.25   30.3384
ppdense_binary_op            2   0.17032  0.085158      12.5    8.9643
ppdense_transpose            2   0.16401  0.082004      12.5    8.6322
ppscalapack_eig              1   0.13009   0.13009      6.25    6.8473
pppblas_gemm                 1  0.028697  0.028697      6.25    1.5104
ppdense_scalar_op            2  0.021018  0.010509      12.5    1.1062
ppscalapack_inv              1  0.020779  0.020779      6.25    1.0937
ppdense_diag                 1  0.015524  0.015524      6.25    0.81708
ppbase_createMatrixCopyRe    1  0.013731  0.013731      6.25    0.72271
ppdense_ones                 1  0.013469  0.013469      6.25    0.70892
ppdense_unary_op             1  0.011289  0.011289      6.25    0.59418
ppdense_sumv                 1  0.003154  0.003154      6.25    0.16601
Total                       16    1.8999   0.11875
```

In this case, the time spent in the various functions is more evenly distributed, suggesting that no major bottlenecks exist. Judicious use of both MATLAB’s “profile” and Star-P's “ppprofile” will generally reveal a large amount of important clues about how you can speed up your code.

**How Much Time Does My Function Take? “tic/toc” and “pptic/pptoc”**

To monitor execution time, the classic tools are MATLAB's "tic" and "toc", which measure the time taken to execute a block of code. Star-P offers a similar facility using functions called "pptic" and "pptoc". "Pptic/pptoc" are useful if you wish to measure the performance of code on the server alone, divorced from front-end server.

Here’s a simple example of using pptic/pptoc to monitor what happens on the server when we invert a matrix.

```matlab
>> A = rand(2000, 2000*p)
A =
>> pptic; B = inv(A); pptoc
```

- 6 -
Client/server communication report:
  Sent by server: 1 messages, 8.800e+01 bytes
  Received by server: 1 messages, 6.600e+01 bytes
  Total communication time: 1.464e-04 seconds

Server processing report:
  Duration of calculation on server (wall clock time): 4.788e+00s
  #ppchangedist calls: 2

----------------------------------------------------------------------
Total time: 4.834e+00 seconds

Three kinds of information are provided by pptic/pptoc: 1. Client/server communication
time and other statistics, 2. Execution time on the server, and 3. Total elapsed time.
Both the communication and the execution time information is useful when optimizing
parallel program performance. Communication overhead is important since if your code
mistakenly creates lots of client/server communication (for example, if you use a for loop
to index a server variable), this diagnostic tool can identify that for you. Computation
time on the server is also important, since it can identify calculations which take longer
than you think they should. Finally, “pptic/pptoc” returns the total time taken by the
calculation. This is essentially the same time as reported by “tic/toc”, as you can see here:

```
>> tic; pptic; B = inv(A); pptoc; toc;
Client/server communication report:
  Sent by server: 1 messages, 8.800e+01 bytes
  Received by server: 1 messages, 6.600e+01 bytes
  Total communication time: 2.579e-04 seconds

Server processing report:
  Duration of calculation on server (wall clock time): 4.828e+00s
  #ppchangedist calls: 2

----------------------------------------------------------------------
Total time: 4.872e+00 seconds
```

The first reported time is that reported by “pptic/pptoc”, the second is that reported by
“tic/toc”. As expected, the two times are nearly the same.

**What Are My Variables? “whos” and “ppwhos”**

An important Star-P tool to gage your program's memory utilization is "ppwhos", which
returns the name, type, and size of all variables – both those on the client as well as those
on the server. This is different from (but analogous to) the information reported by
MATLAB's “whos”, which only has complete information about variables living on the
client. Here's a session log showing the difference:

```
>> A = rand(100, 100); % lives on client
>> B = rand(1000*p, 1000); % lives on server
>> C = sin(B); % also lives on server
>> ppwhos
Your variables are:
  Name   Size         Bytes       Class
  A  100x100     80000       double array
  B  1000px1000  8000000     ddense array
```
When looking at the output of “whos”, you might ask: “Why does ‘whos’ report anything about the server variables B and C? And what do the reported values mean?” On the client side, “whos” reports the size of the **reference objects** which point to B and C on the server. The references are much smaller than B and C themselves. Only “ppwhos” shows the full size of B and C; moreover you can deduce that B & C live on the server since their sizes are 1000p x 1000 – the p is the tip that they are server variables. (Their class – ddense – also tells you they are server variables.)

**Craftsman's advice:**

- Use code profiling tools to find bottlenecks in your code.
- When faced with porting an existing MATLAB program to Star-P, it's a good idea to run MATLAB's profiler first to find code trouble spots, and obvious areas for parallelization.
- MATLAB's profiling tools generally return information about the entire calculation performed. Star-P's profiling tools measure only things happening on the server.
- Once you have identified performance bottlenecks in your code, then you can focus your primary parallelization efforts on removing them.

**Is Your Problem Data-Parallel or Task-Parallel?**

When building a house, the tools you will depend upon the nature of the materials you wish to use. If you're building a brick house, then the necessary tools include trowels and plumb lines. But if you're building a wood house, you need a hammer and a saw.

Similarly, when thinking about crafting code for a parallel computer, you need to consider the nature of your algorithm before applying one parallelization method or another. The key question is: "What model of parallel computation most closely fits your algorithm -- data-parallel or task-parallel?" To remind you of the difference:

- **Data-parallelism** (sometimes called "fine-grained parallelism") refers to spreading matrix data over all compute nodes in the machine and having each processor manipulate a portion of the matrix. Each processor performs short, identical computations in parallel lockstep, and then communicates the results to other processors before the next operation. In Star-P, you trigger data-parallel operations by creating the appropriate matrices with the “*p” tag, or by explicitly placing them on the server using “ppback” or “ppload”. 

```plaintext
C  1000px1000  8000000  ddense array
Grand total is 2010000 elements using 16080000 bytes
MATLAB has a total of 10000 elements using 80000 bytes
Star-P server has a total of 2000000 elements using 16000000 bytes
>> whos
Name      Size                    Bytes  Class
A       100x100                   80000  double array
B      1000x1000                   1148  ddense object
C      1000x1000                   1156  ddense object
Grand total is 10062 elements using 82304 bytes
```
- Task-parallelism (also called "coarse-grained parallelism") refers to performing longer, non-communicating computations on independent compute nodes. Task parallel operations could sensibly run on separate threads (or separate processes) on a serial computer. In Star-P, you use “ppeval” when you want to invoke a task-parallel computation.

Keep in mind that these are not mutually exclusive categories. Data- and task-parallelism mark two end points of a continuous spectrum of computational models, and many problems can be solved using either (or both) of the two computational styles. Nonetheless, determining which model best fits your problem can help you decide how best to proceed with your parallelization efforts. Here are some common examples:

- **Data-parallel**
  - Finite element modeling (FEM)
  - Fast Fourier transforms (FFT)
  - Interacting particle systems
  - Image processing
  - Swarms
  - Neural nets

- **Task-parallel**
  - Unrolling for loops (embarrassingly parallel problems)
  - Monte Carlo simulations
  - Genetic algorithms
  - Searching (e.g. Google)
  - Video processing (operate on frames independently)

We will present examples and code samples of several of these problem types in the following sections.

Another hint about whether your program is best cast as data- or task-parallel can be taken from the sizes of the matrices you are using. Here are some loose guidelines:

1. **Matrix sizes under 100kB.** If you have matrices whose sizes are under under 100kB, you are probably best off not using Star-P's parallelism features at all. That is, the overhead required by Star-P to communicate between the client and server machines is likely larger than the speedup you will gain by running your operation on a parallel machine. Use serial MATLAB instead.

2. **Matrix sizes between 100kB and 100MB.** In this range, your problem can be well served using either data-parallel or task-parallel methods. Which you choose depends upon the exact nature of your algorithm.

3. **Matrix sizes above 100MB.** Above this size, the communication overhead involved in distributing your data using ppeval will impact performance significantly. Consider a data-parallel method to solve your problem.
This is shown schematically in Figure 3. Note that the exact crossover points are not necessarily precisely those provided above, but rather depend upon the detailed configuration of your client and server, the number of processors use, the operation you wish to perform, and a host of other factors. For example, depending upon which function is used, we have observed the serial vs. parallel performance crossover (i.e. case 1 above) at matrix sizes ranging between 100kB and 1MB. As another example, parallelizing a “for” loop using “ppeval” (i.e. task-parallel execution) doesn’t necessarily bring a major performance improvement if your server has only a small number of parallel compute nodes, but does become advantageous for large numbers of compute nodes. If you are in doubt about your particular problem, do some timing experiments at the Star-P command line using “tic” and “toc”.

Craftsman's advice:
- Decide if each block in your program is more naturally data-parallel or task-parallel. This determination is generally based upon the nature of your problem or your algorithm.
- Also, consideration of your data's size usually provides a clue about whether your program better fits a data-parallel or a task-parallel computational model.
- Use the “*p” tag to label large matrices as data-parallel. Use “ppeval” to initiate task-parallel computations.
Overcoming Parallelization Challenges

Missing Functions or Unsupported MATLAB Toolboxes

Star-P's “ppeval” function enables task-parallel evaluation of MATLAB functions by exporting multiple copies of your function to the supercomputer server, and then executing your function using standard numerical libraries on the back-end. However, a small number of MATLAB functions are currently unimplemented on the back-end, especially those included in the various MATLAB add-on toolkits.

What can you do if your code uses unsupported MATLAB functions? We recommend you try the following:

1. Can you find an open-source implementation of your desired function on the web? Many people have contributed MATLAB, Octave, or Scilab code to various on-line code repositories; it’s worth your while to explore those sites to see if your function – or one similar to it – has already been written by somebody else. A list of useful websites is given in the Appendix. Also, a quick Internet search is usually a fruitful way to find relevant code and advice.

2. Can you write a MATLAB replacement function yourself? Since Star-P supports all base MATLAB functions, writing a replacement using functions available from the base MATLAB package may be easy.

   Once you’ve either located a replacement “.m” function, or have written your own, make sure you test it for correctness! Then, when you’re satisfied that it gives the right answers, copy your replacement file into the “compat” directory on the server. The “compat” directory is located in the top level of the Star-P distribution, i.e. <your Star-P installation directory>/compat.

   By the way, the “compat” directory is the first directory in Star-P’s search path, so if you ever want to overload or replace any existing Star-P functions, copy your replacement function into the “compat” directory, and your replacement function will override any other function of the same name.

3. If your desired function is more involved, or is time-critical, you can find or write a C/C++ replacement. Star-P offers a method to load functions from C/C++ libraries called “Star-P Connect”. This functionality is outside of the scope of this document; contact your Interactive Supercomputing representative for more information about this feature.

4. Finally, Interactive Supercomputing has partnered with other prominent firms serving the numerical computation market. These firms offer numerical libraries which plug into Star-P, providing you access to high-quality solutions to difficult numerical problems. Notable libraries available for Star-P include Visual Numerics' IMSL library, as well as a variety of libraries marketed by the Numerical Algorithms Group (NAG). Contact your Interactive Supercomputing representative for more information about these third-party solutions.
Dealing with Structs and Cell Arrays

Complex MATLAB code often makes use of structs or cell arrays because these constructs provide a convenient way to organize and maintain related data sets. Besides using structs to keep track of low-dimensional data, some programmers occasionally use them to hold large arrays of numbers. Sometimes they even create gigantic arrays of structs whose members are data which they wish to manipulate.

Star-P allows the use of structures **locally** inside of functions called by “ppeval”. Structs and cell arrays on the client side also work in Star-P. However, current versions of Star-P lack the ability to pass structures or cell arrays from client to server. This means that you cannot distribute a struct or cell array using the “*p” tag. Nor can you pass a struct or cell array as an argument to “ppeval”.

If your current MATLAB code uses data stored in structs, and you wish to move that data to the server for manipulation, then you need to replace the struct with a numeric data type. For example, consider a struct holding a vector as one of its members. In such a situation it's useful to give the new array a name which is reminiscent of the old struct, like this:

% Original code – struct includes a vector as member
foo.bar = 1
foo.baz = [1 2 3 4 5 6 ]
foo.woof = 'String'

% Under Star-P – make vector into separate variable.
foo.bar = 1  % client side -- still valid under Star-P
foo.woof = 'String' % client side -- still valid under Star-P
foo_baz = [1 2 3 4 5 6]  % vector name reminds you of struct.

Other, more complicated structs are possible using MATLAB. As a general rule, if your data is held in a struct, and you need to pass a part of that data to the server, then you must convert that portion of the struct into a numeric vector, matrix, or array before moving it to the server.

Craftsman's advice:

- Star-P supports all base MATLAB functions. However, functions from some toolboxes remain unimplemented. If you need to use a function missing from Star-P, consider writing your own, finding a similar function on the web, or using a third-party library.

- Star-P does not support passing structs or cell arrays between client and server. Replace structs and cell arrays with numeric vector or matrix types if you wish to pass them.

- Give your replacement matrices names evocative of your original struct or cell array to help you keep track of what your code is doing.
Vectorize Your Code!

Vectorizing your code is essential to obtaining good performance of your application under Star-P. But what does "vectorizing" mean? Simply put, vectorizing your code means that you avoid using "for" loops to operate on individual elements of a vector or array. Instead, you operate upon vectors and arrays as entire units using MATLAB's built-in functions.

"Vectorization" is probably a familiar concept to you if you've been writing MATLAB code for a while. As you may know, vectorizing your code remains a cornerstone of achieving the best performance from both MATLAB and Star-P code. There are at least two reasons for this:

1. For Star-P, the first reason has to do with where your variables live. In Star-P, scalar variables always live on the client machine. When you use a scalar to index into an array residing on the server, Star-P must send the index from the client to the server to find the appropriate array element. Then, Star-P must spend time communicating amongst the different process nodes while looking for your element. Once the element has been found, it is then sent back to the client for use. All this communication implies a significant time penalty.

2. For both MATLAB and for Star-P, the second reason is this: The staff at both The Mathworks and at Interactive Supercomputing are experts at numerical mathematics. They have devoted a lot of research and development time on creating highly optimized, parallel algorithms for evaluating of many of the common math functions which you are likely to use. As a general rule, you're best off relying upon the built-in, parallelized functions provided by experts for optimum performance, rather than trying to write your own.

Here are some quick examples demonstrating ways to vectorize common situations. The lines which change upon vectorization have been highlighted in blue.
<table>
<thead>
<tr>
<th>Not vectorized</th>
<th>Vectorized</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \delta_t = 0.001; ] [ \omega = 50; ] [ \text{end}_t = 10; ] [ \text{numpoints} = \frac{\text{end}_t}{\delta_t} + 1; ]</td>
<td>[ \delta_t = 0.001; ] [ \omega = 50; ] [ \text{end}_t = 10; ] [ \text{numpoints} = \frac{\text{end}_t}{\delta_t} + 1; ]</td>
</tr>
<tr>
<td>% -- Bad: create vector with for loop -- [ \text{fourier}_\text{vector3} = \text{zeros}(\text{numpoints*}p, 1)'; ] [ \text{for } j = 1: \text{numpoints} ] [ \text{time}<em>3 = (j-1)*\delta_t; ] [ \text{fourier}</em>\text{vector3}(j) = \ldots ] [ \text{exp}(i<em>2</em>pi*\omega*\text{time}_3); ] [ \text{end} ]</td>
<td>% ---- Good 1: use colon operator ----- [ \text{time}<em>\text{vector1} = 0: \delta_t:<em>\text{end}_t</em>p; ] [ \text{fourier}</em>\text{vector1} = \ldots ] [ \text{exp}(i<em>2</em>pi*\omega*\text{time}<em>\text{vector1}); ] % ---- Good 2: use linspace ----- [ \text{time}</em>\text{vector2} = \ldots ] [ \text{linspace}(0, \text{end}<em>t, \text{numpoints}*p); ] [ \text{fourier}</em>\text{vector2} = \text{exp}(i<em>2</em>pi*\omega*\text{time}_\text{vector2}); ]</td>
</tr>
<tr>
<td>% Bad: calculate matrix with for loop [ v=\text{rand}(1000*p); ] [ \text{for } i=1: \text{size}(v,1) ] [ \text{for } j=1: \text{size}(v,2) ] [ \text{vs} = \sin(v(i,j)); ] [ \text{end} ] [ \text{end} ]</td>
<td>% Good: calculate matrix with built-in [ v=\text{rand}(1000*p); ] [ \text{vs}_\text{vec} = \sin(v); ]</td>
</tr>
<tr>
<td>% Bad: conditional inside for loop [ \text{numpoints} = 10000; ] [ a = 2*\text{rand}(\text{numpoints}*p, 1)-1; ] [ \text{for } i = 1: \text{numpoints} ] [ \text{if } (a(i) &lt; 0) ] [ a(i) = 0; ] [ \text{end} ] [ \text{end} ]</td>
<td>% Good: vectorized conditional [ \text{numpoints} = 10000; ] [ a = 2*\text{rand}(\text{numpoints}*p, 1)-1; ] [ \text{idx} = (a &lt; 0); ] [ a(\text{idx}) = 0; ]</td>
</tr>
<tr>
<td>Not vectorized</td>
<td>Vectorized</td>
</tr>
<tr>
<td>----------------</td>
<td>------------</td>
</tr>
</tbody>
</table>
| % Bad: loop over vector elements  
numpoints = 1000;  
x = rand(numpoints*p, 1); % row vec  
y = rand(numpoints*p, 1);  
r = zeros(numpoints, numpoints*p);  
for i = 1:numpoints  
  for j = 1:numpoints  
    r(i, j) = sqrt(x(i)^2 + y(j)^2);  
  end  
end | % Good: vectorize using ndgrid  
numpoints = 1000;  
x = rand(numpoints*p, 1); % row vec  
y = rand(numpoints*p, 1);  
[ xmat, ymat ] = ndgrid(x, y);  
r = sqrt( xmat.^2 + ymat.^2)  
| % (Can use meshgrid too, except  
% meshgrid creates transposed matrix) |

For more complicated expressions, an obvious vectorized expression is not always readily apparent. However, this doesn't mean that an expression is not vectorizable! Several vectorization tricks exist which can come in handy when writing or porting your code to Star-P. A few tricks were presented in the data-parallel examples above. Also, a simple web search will quickly turn up more ideas. Some useful websites are listed in the Appendix. For good performance from Star-P it is critical to vectorize your code as much as possible! Therefore, taking the time to search for vectorization tricks to apply to your code is time well spent.

## The Big Step: Parallelizing your code

At this point, you are ready to parallelize your code. Star-P offers facilities for both data-parallel and task-parallel computation. Data parallel operations occur implicitly on variables living on the server. You can place variables onto the server either by creating them using the “*p” tag, or by explicitly moving them there using “ppback”, “ppload”, or a similar Star-P function. Task-parallel operations are invoked explicitly using the "ppeval" statement, which calls any desired function in parallel on given arguments.

A simple program using both of these facilities is shown below. This program is a Star-P classic since it nicely demonstrates how easy Star-P makes parallel computing. It reads in an image file stored in MATLAB's RGB format, moves it to the server, and performs a singular value decomposition based compression operation on it. Then, it decompresses the image and displays it. This program shows both data-parallel and task-parallel styles of parallel computation at work. Specifically:

- **Data-parallel**: After read-in, the image itself is moved to the server as a distributed three-dimensional array, with the first two dimensions labeling the pixel row and column, and the third dimension demarking the color plane. Since the array is distributed, mathematical operations performed on it are performed by Star-P in data-parallel mode.

- **Task-parallel**: The compression and decompression functions are called using “ppeval”, and the image is split so that individual color planes are processed in task-parallel mode.

Here's the code:
% SVD_top_pp.m – Singular value decomposition image compression.
% This is top-level fcn which reads an image (as an NxMx3 dimensional
% array), and does an SVD compression on the image. Then it decompresses
% the image and displays it on the client.

% Set compression rank
rank = 16;

% Read image from disk on client. Name of image array is "a".
load('trilliumIII.mat','a'); % trilliumIII.mat is name of image file.

% Move image to server for data parallel processing.
a = ppback(a);

% compress image using SVD factorization
[uout sout vout] = ppeval('SVD_compress', split(a, 3), rank);

% reconstitute compressed image.
compressed_image = ppeval('SVD_decompress', split(uout, 3), ...
        split(sout, 2), split(vout, 3));

% Move compressed image to client
compressed_image = ppfront(compressed_image);

% plot the compressed image
image(compressed_image);

function [uout sout vout] = SVD_compress(a, rank)
% Apply Matlab's SVD function
[u,s,v] = svd(a, 0);

% Chop matrices to size specified by rank.
vout = v(:,1:rank)';
uout = u(:,1:rank);
fulls = diag(s);
sout = fulls(1:rank);

function o = SVD_decompress(uout, sout, vout)
% Reconstruct the image from uout, sout, vout
o = uout * diag(sout) * vout;

% Now clip out of range image data caused by compression
ind = (o > 1);
o(ind) = 1;
ind = (o < 0);
o(ind) = 0;

The resulting image, compressed to rank 16 is shown in Figure 4.

More code examples showing several different types of algorithms parallelized using Star-
P are presented in Section 3 of this document.
Craftsman's advice:

- Data-parallel computation occurs implicitly on variables living on the server.
- You can create server variables either by creating them with the "*p" tag, or by explicitly moving them there using "ppback", "ppload", or a similar function.
- Task-parallel computation is invoked explicitly using "ppeval".

Section 2 – Tool Tips for Writing Parallel Code

Matrix Types: Dense or Sparse?

StarP, like MATLAB, supports two storage methods to hold matrix data: dense and sparse. These two storage methods are optimized to hold your matrix differently, depending upon how many non-zero elements it has:
A dense matrix is one in which almost every element is non-zero. It is stored in computer memory as a large array in which every element has a reserved storage location, regardless of its value. Such matrices are called “ddense” in Star-P.

A sparse matrix is one which contains a large number of zeros, and only a few non-zero elements. In this case, only the non-zero elements are held in memory; since the rest are known to be zero, they aren’t stored. Such matrices are called “dsparse” in Star-P.

Graphical representations of a dense and a sparse matrix are presented in Figure 5.

Figure 5: Dense vs. sparse matrices. Both matrices are 60x60. Each non-zero element is represented by a blue dot. The dense matrix on the left has a non-zero element in every position. The sparse matrix on the right has non-zero elements near the main diagonal, and scattered throughout the rest of the matrix. This particular sparse matrix has manifest structure in the non-zero elements, but an arbitrary sparse matrix may have no structure at all.

Situations where dense matrices are applied to real-world problems include image processing, physical simulations of interacting particles when the forcing function is long range (e.g. gravity), and economics (e.g. linear programming, input-output models). Sparse matrices are used in electrical circuit simulation, physical simulations of particles interacting via a short-range force (e.g. screened potentials), and finite element modeling (heat flow or elasticity simulations).

Whether your matrix is dense or sparse depends upon how you create it. Most matrix creation methods in Star-P create dense matrices. To create sparse matrices, you usually need to do so explicitly. Here are some examples of matrix creation:

```matlab
>> a = zeros(1000, 1000*p)
a =
    ddense object: 1000-by-1000p
>> b = ones(500, 500*p)
```
As you can see, “zeros”, “ones”, and “eye” all return dense matrices. Making a sparse matrix from a dense one can be accomplished using the “sparse” function.

There are several advantages to using sparse matrices if your problem allows it:

- Obviously, a sparse matrix uses up less memory than a dense one. This can be seen in the output of “ppwhos” above, which shows that while matrix “c” chews up 32MB of memory, matrix “d” – which is mathematically equal to “c” – requires only 48kB! Since a sparse matrix uses less memory than a dense one of equivalent rank, you can squeeze much larger sparse matrices into your computer than dense ones.

- Besides the memory advantages offered by using sparse matrices, they can help your program’s performance too. If you have super large matrices, and their size is comparable to than the amount of RAM in your HPC, then any operation on the matrix can incur a tremendous speed penalty because your computer starts to page blocks of RAM onto and off of disk, which is very, very slow. Worse, your program can simply crash if it runs out of memory. The threshold size at which this occurs is much higher for a sparse matrix than for a dense one.

- Many “for” loops – both explicit and implicit – iterating over an NxM sparse matrix can finish much more quickly than loops running over the same NxM dense matrix. The reason is that you need only iterate over – and operate on – the non-zero elements of the sparse matrix, whereas you must iterate over all elements in a dense matrix, whether they are zero or not.

Here’s an example program showing the performance speed-up achievable using sparse matrices. We consider a one dimensional Ising Model simulation. As a reminder, the Ising model is studied in the physics of magnetism and phase transitions. It models a
magnetic material as a lattice of spins which can point either up or down. Each spin will interact with the magnetic field produced by its nearest neighbor, as well as with an external applied magnetic field. Depending upon the dimensionality of the problem, different types of physically interesting behavior can be demonstrated by the Ising Model.

In this code we examine the one-dimensional Ising Model, in which the spins are imagined to live evenly spaced on a chain. Since the spins interact only with their nearest neighbor, the spin-spin interaction matrix for the entire spin chain is very sparse, with non-zero interaction coefficients lying only on the super and sub diagonals (i.e. right above and right below the main diagonal) of the matrix. Additionally, the externally applied field interacts with the spins only via the main diagonal.

```matlab
% Ising -- calculates energy of Ising chain model
%          for random chain configuration.

n = 100000*p;    % Length of linear Ising chain.
J = 1;           % Spin-spin interaction
h = .03;         % Ext magnetic field
e = ones(n, 1);  % Helper vector
Beta = 1/40;     % Temperature parameter

% Hamiltonian matrix -- create as sparse
H_sparse = -J * spdiags([e 0*e e], -1:1, n, n) + ...
         -h * spdiags([e], 0, n, n);

% Create spin chain -- random vec of +/- 1
Is = 2*(rand(1, n) < .5) - 1;

% Calc energy of chain using sparse matrix
tic
E_sparse = Is*H_sparse*Is';
toc

% Calc energy using dense matrix
H_dense = full(H_sparse);
tic
E_dense = Is*H_dense*Is';
toc
```

In this program, we compute the total energy of a randomly generated chain of spins by forming the product

$$ E = I_s H I_s' $$

where \( I_s \) is a vector whose elements are randomly +1 or -1 representing the spin chain's randomly chosen configuration, and \( H \) is a square matrix which represents the energy contribution of each spin's interaction with its neighbors and with the external field. The energy is computed twice, once using a sparse representation of \( H \), and then again using a dense representation of \( H \).

Implicit in the energy product is a set of loops, summing the spin interactions over each element in the \( I_s \) vector. Both MATLAB and Star-P implement efficient looping algorithms which are aware of whether the matrix involved is sparse or dense. Therefore, when
creating the energy product, using a sparse matrix requires fewer loop iterations than a dense one, and the difference grows as the size of the matrix grows. Timing results showing the computation time vs. spin chain length are presented in the table below. These results were obtained using Star-P on an 8-way shared memory machine.

<table>
<thead>
<tr>
<th>Chain length n</th>
<th>Time (sparse)</th>
<th>Time (dense)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.101</td>
<td>0.106</td>
</tr>
<tr>
<td>300</td>
<td>0.097</td>
<td>0.104</td>
</tr>
<tr>
<td>1000</td>
<td>0.097</td>
<td>0.115</td>
</tr>
<tr>
<td>3000</td>
<td>0.115</td>
<td>0.304</td>
</tr>
<tr>
<td>10000</td>
<td>0.096</td>
<td>2.193</td>
</tr>
<tr>
<td>30000</td>
<td>0.099</td>
<td>19.343</td>
</tr>
<tr>
<td>100000</td>
<td>0.148</td>
<td>Crash (out of memory)</td>
</tr>
</tbody>
</table>

For chain lengths less than 1000 elements (corresponding to size(Hdense) = 1 million elements), both sparse and dense matrices offer the same computational time. Above this threshold, however, using the sparse representation of H offers a clear performance advantage over the dense representation. This occurs because far fewer loop iterations must be performed for matrix multiplication in the sparse case.

**Craftsmen's advice:**
- Use sparse matrices if your problem permits it, particularly if you are handling extremely large matrices.

**Row or Column: What's the Best Distribution for Parallel Processing?**

When first becoming acquainted with Star-P, you undoubtedly learned to row or column distribute your matrices by placing the *p tag appropriately, like this:

```matlab
A_row = rand(1000*p, 1000);    % Row distributed
B_col = rand(1000, 1000*p);    % Column distributed
```

What you may have wondered, however, is: “Why does it matter?” It turns out that the answer to this question is: “It might not matter! But if it does, then the optimal distribution depends upon the details of each function you wish to compute.” And in general, the greatest determining factor is whether your function is a Star-P built-in or if it comes from a third-party library.

**Optimal Distributions for Star-P Built-ins**

Naively, you might expect that choosing row or column distribution would be critically important for matrix multiplication. After all, when multiplying two matrices, the rows of the left-hand matrix are multiplied by the columns of the right-hand matrix! However, for
most built-in functions, Star-P performs a matrix redistribution before invoking the function. Therefore, no distribution choice will offer an overwhelming performance advantage over any other! Here's an example:

```matlab
>> A_row = rand(1000*p, 1000);
>> A_col = rand(1000, 1000*p);
>> B_row = rand(1000*p, 1000);
>> B_col = rand(1000, 1000*p);
>
>> tic; A_row * B_row; toc
Elapsed time is 0.158763 seconds.
>> tic; A_row * B_col; toc
Elapsed time is 0.186611 seconds.
>> tic; A_col * B_row; toc
Elapsed time is 0.132834 seconds.
>> tic; A_col * B_col; toc
Elapsed time is 0.112696 seconds.
```

The differences between the various distribution combinations is largely just random fluctuation due to network latency and variable server loading. Similarly, most Star-P built-in functions – whether unary or binary – evince the same insensitivity to distribution.

However, there are a few Star-P functions which greatly prefer one distribution over the other. Perhaps the most frequently used is the fast Fourier transform function “fft”. Here's an example:

```matlab
>> x = rand(4000, 4000*p);
>> tic; y=fft(x); toc
Elapsed time is 0.232781 seconds.
>> x = rand(4000*p, 4000);
>> tic; y=fft(x); toc
Elapsed time is 1.924117 seconds.
```

Clearly, “fft” prefers column distributed data.

In light of this, the natural question is: How does one know if a given function prefers a particular distribution, or is insensitive to distribution? In general, if a function has a preference, it prefers column distributed matrices. Here is a table of common unary functions in which one of the distributions executes more than twice as fast as the other when operating on a dense 2500x2500 matrix.

<table>
<thead>
<tr>
<th>Function</th>
<th>Preferred distribution</th>
<th>Speedup ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>cumprod</td>
<td>Column</td>
<td>3.5</td>
</tr>
<tr>
<td>fft</td>
<td>Column</td>
<td>6.9</td>
</tr>
<tr>
<td>ifft</td>
<td>Column</td>
<td>6.9</td>
</tr>
<tr>
<td>median</td>
<td>Column</td>
<td>10</td>
</tr>
<tr>
<td>sign</td>
<td>Column</td>
<td>9.7</td>
</tr>
<tr>
<td>Function</td>
<td>Preferred distribution</td>
<td>Speedup ratio</td>
</tr>
<tr>
<td>----------</td>
<td>------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>sort</td>
<td>Column</td>
<td>18</td>
</tr>
</tbody>
</table>

This list is not meant to be authoritative. If you have questions about a particular built-in function, please contact your Interactive Supercomputing support engineer. In general, however, we recommend you to distribute your matrices on the column (or on the last dimension if you are using ND arrays).

**Optimal Distributions for Third-Party Libraries**

Although most StarP functions are insensitive to matrix distribution, many (or most) third party libraries are not. Therefore if your program interfaces to external programs (i.e. through “Star-P Connect”) or libraries which are sensitive to distribution, then you must carefully consider how you distribute your matrices. In this situation you may ask “how do I know whether to call it with a row distributed or a column distributed matrix?” The answer is: Experiment! Surround the function with “tic/toc” or "pptic/pptoc" and send it random matrices distributed in all ways possible. Then scale the matrix size up and see which distribution (if any) offers faster execution time, or which breaks first when the matrix size becomes gigantic.

**Redistributing Matrix or Array Data**

One final piece of advice should be given when it comes to distributing your data. Redistributing your server data via “reshape”, transpose, or other such operations requires a lot of communication amongst the process nodes on your server. Therefore, it's best to keep the amount of redistribution to a minimum for best Star-P performance. In particular, if you have a matrix which must be transposed repeatedly for a large number of computations, consider creating it in transposed form, rather than transposing it with every computational step.

**Craftsman's advice:**

- Most Star-P built-in functions – unary and binary – don't care about how the data is distributed.
- Some built-in functions, however, do care. Therefore, if you are in doubt about how to distribute your data, choose column distribution as your default. (Or the last dimension, if your data is 3D or above.)
- Third-party function may require row or column distribution based upon the details of the particular library and function invoked.
- Using “tic/toc” or “pptic/pptoc” to experimentally determine the best distribution for your data is the safest way to ensure optimal performance.
- Minimize the number of times you redistribute matrix or array data.
Polishing Your Code: Tool-tips for “ppeval”

Performance Tip: Vectorize “for” Loops Inside of “ppeval” Calls

Vectorizing data-parallel code is a key factor to achieving good performance under Star-P. Some programmers, however, ignore this advice when writing task-parallel code. They reason that the entire task-parallel function is evaluated on the server – including the scalar variables. Therefore, they think that iterations occurring inside “ppeval” functions will be fast. Unfortunately this reasoning is wrong.

While it is true that scalars passed by “ppeval” live on the server, vectorizing your function remains important since iteration on the server still involves an interpreter. Therefore, you should still vectorize your function, even if it is evaluated using “ppeval”. The following example shows the effort needed and gains achieved by vectorization inside a “ppeval” call:

```matlab
% Top.m -- Top level fcn invokes two different versions of sum to see which is faster.
% Main function. Assume computation involves processing slices of 3D array.
n = 1000;
yarr = rand(3, n, 8);
zarr = rand(3, n, 8);

tic;
x_looping = ppeval('fcn_looping', n, ...
        split(yarr, 3), split(zarr, 3));
toc

tic;
x_vectorized = ppeval('fcn_vectorized', n, ...
        split(yarr, 3), split(zarr, 3));
toc

function x = fcn_looping(n, y, z)
% Unvectorized version -- Bad!
for i = 1:n
    if z(1,i) >= 0
        x(i) = y(1,i)*z(1,i) + y(2,i)*z(2,i) + y(3,i)*z(3,i);
    else
        x(i) = y(1,i)/z(1,i) + y(2,i)/z(2,i) + y(3,i)/z(3,i);
    end
end
```
The performance gains achieved by this vectorization inside of ppeval are shown in the following table as a function of the loop index.

<table>
<thead>
<tr>
<th>Loop Iterator ( n )</th>
<th>Looping Execution Time (sec)</th>
<th>Vectorized Execution Time (sec)</th>
<th>Speed-up Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.798259</td>
<td>0.751317</td>
<td>1.06</td>
</tr>
<tr>
<td>1000</td>
<td>1.444807</td>
<td>0.815917</td>
<td>1.77</td>
</tr>
<tr>
<td>10000</td>
<td>9.126191</td>
<td>1.231624</td>
<td>7.41</td>
</tr>
<tr>
<td>100000</td>
<td>351.754691</td>
<td>5.797857</td>
<td>60</td>
</tr>
</tbody>
</table>

Clearly, vectorizing the code becomes more important as the number of iterations increases.

**Performance Tip: Beware of Trailing Singleton Dimensions.**

In MATLAB, as well as inside “ppeval”, trailing singleton dimensions are ignored. (“Singleton dimension” refers to the situation in which the size of an array’s dimension is one.) For instance, “\( x = \text{rand}(100,100,1) \)” creates a two dimensional 100-by-100 array. It does not create a three dimensional array as you might expect. Simply creating arrays with trailing singleton dimensions does not present a performance issue. However, if at some point you try to index into your array using all three indices, “ppeval” can become slow. The following example illustrates the issue:

```matlab
% Set up arrays with trailing singleton dimensions.

n = 1000;
k = 5;
m = 1;
x = zeros(n,n,m); % Note this creates a n-by-n array NOT n-by-n-by-1
y = rand(n,n,m); % Note this creates a n-by-n array NOT n-by-n-by-1
x(:,:,1) = y(:,:,1);

% Now use "real" two dimensional arrays. As a note using % three dimensional arrays without trailing singleton % dimension also perform well.

x = zeros(n,n);
y = rand(n,n);
x(:,:,k) = y(:,:,k);
```

The performance gains achieved by avoiding trailing singletons inside of “ppeval” are
shown in the following table as a function of the array size.

<table>
<thead>
<tr>
<th>Size of n</th>
<th>With Trailing Singleton</th>
<th>Without Trailing Singleton</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.00181 sec</td>
<td>0.00015 sec</td>
<td>12</td>
</tr>
<tr>
<td>500</td>
<td>0.01403 sec</td>
<td>0.00026 sec</td>
<td>54</td>
</tr>
<tr>
<td>1,000</td>
<td>0.05477 sec</td>
<td>0.00031 sec</td>
<td>177</td>
</tr>
<tr>
<td>2,000</td>
<td>0.26016 sec</td>
<td>0.00034 sec</td>
<td>765</td>
</tr>
<tr>
<td>4,000</td>
<td>0.87707 sec</td>
<td>0.00041 sec</td>
<td>2139</td>
</tr>
</tbody>
</table>

**Performance Tip: Assignments to Structs can be Slow.**

Structures are useful programming constructs that allow you to group together data that are related to each other into one place. Furthermore, they can contain elements of varying data type. Currently, “ppeval” does not support structures or cell arrays as its input or output argument. However, this does not prevent the use of local structures inside of functions called by “ppeval”. Nonetheless, it is important to keep in mind that assignment to elements inside structures might be slow depending on the dimensionality and size of the element. In general, the higher the dimensionality, the greater the performance impact.

There are however two ways to work around this: 1. Remove all references to structures throughout your function and replace them with straight numeric variable types (i.e. vectors or matrices) or 2. Temporarily extract the data element you are working with from the structure, perform the calculations on it, and finally put the element back into the structure. The following example illustrates the latter possibility:

```matlab
% Set up variables
n = 1000;
m = 100;
a = rand(n,1);
F.b = zeros(n,m,m);   % Create 3D array inside structure F

% Perform calculation
for i = 1:m
    F.b(:,5,i) = a;
end

% Perform calculation with temporarily extracting array b
b = F.b;              % Extract b
for i = 1:m
    b(:,5,i) = a;
end
F.b = b;              % Put b back into F
```

Note that this technique works best if your calculation involves a large number of intermediate steps. The performance gains achieved in the above example are shown in the following table as a function of the array size.
Performance Tip: Avoid ND Indexing!

Within “ppeval” indexing into three or more dimensional arrays is time-consuming and should be avoided whenever possible. One workaround is to lower the dimensionality of the problem by storing sub-arrays into cell arrays. This can dramatically increase performance:

```matlab
% Setup variables
n = 1000;
k = 5;
X = zeros(n,n,k);  % Create 3D array

% Perform indexing operation on 3D array
for i = 1:k
    Y = X(:,:,i);
end

% Create cell array that will contain 2D sub arrays of X above
X = cell(1,k);
for i=1:k, X{i} = rand(n,n); end  % fill cell array with 2D arrays
% Perform indexing operation of cell+2D array
for i=1:k
    Y = X{i}(:,:,);  % Note that this trick will only work locally inside a “ppeval” call.
end
```

Note that this trick will only work locally inside a “ppeval” call. This is because “ppeval” can’t pass cell arrays as input or output arguments. The performance gains achieved in the above example are shown in the following table as a function of the array size.

<table>
<thead>
<tr>
<th>n</th>
<th>Assignment within Structure</th>
<th>Assignment outside Structure</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1.148 sec</td>
<td>0.014 sec</td>
<td>82</td>
</tr>
<tr>
<td>500</td>
<td>5.085 sec</td>
<td>0.066 sec</td>
<td>77</td>
</tr>
<tr>
<td>1000</td>
<td>13.461 sec</td>
<td>0.133 sec</td>
<td>101</td>
</tr>
<tr>
<td>2000</td>
<td>33.120 sec</td>
<td>0.254 sec</td>
<td>130</td>
</tr>
<tr>
<td>4000</td>
<td>62.457 sec</td>
<td>0.939 sec</td>
<td>66</td>
</tr>
</tbody>
</table>
Hardware considerations

The first reaction many people when granted access to a parallel computer is “Wow! Now I have lots of processors so I can solve my problem quickly!” Unfortunately, this isn't always the case. Remember: use the right tool for the job! To achieve meaningful performance enhancements using Star-P, your hardware setup must complement your parallel algorithm. Otherwise, you may incur a performance penalty when parallelizing your program!

Network Latency

Remember that Star-P is a client-server application which does its job by supervising the communication of data and results over a LAN connection between your desktop MATLAB client and a supercomputer server. Therefore, the amount of client-server communication and the speed at which the communication takes place can play a critical role in the performance you obtain from Star-P.

When talking about the speed of a communication link, there are two quantities which merit consideration: the link's bit-rate and its latency. Bit-rate is simply how many bits per second are transmitted once the transmission has started. Everybody knows what bit-rate means since we talk all the time about 100BaseT (100 million bits per second) or Gigabit Ethernet (one billion bits per second) LAN cards. Link latency, however, can be equally important for may applications, including Star-P, and it is seldom mentioned.

Loosely speaking, latency refers to the time delay between when you start to send the data, and when the data actually begins to show up at its destination. In the case of Star-P, the latency is the amount of time it takes for Star-P to move your data from the client to the numerical routines running on the server. Moving your data involves a long chain of autonomous events: Sending the data down your client machine's TCP/IP stack, sticking the IP packets into your Ethernet card's hardware buffers, transmitting the packet through one or more routers (which might buffer the data for a while), receiving the packet at the server, buffering it, signaling and servicing a communications interrupt, then moving the data up through the server's TCP/IP stack so that it becomes available for use on the supercomputer. That's a lot of work! Depending upon the details of your system's configuration, this long sequence of operations can take anywhere from microseconds to hundreds of milliseconds.

One crude way to gage the effect of latency on your program's performance under Star-P is to “ping” your server from your client machine, and look at the round-trip time on your network. Then, assume you want your communication time to be only 10% of your total
Star-P run time. This will place a lower limit on the time each built-in MATLAB function must take before it becomes advantageous to run using Star-P. For example, if ping shows that your client-server round trip time is 50mS, then your MATLAB function must take 500mS or longer to sensibly run on the server using Star-P. Otherwise, you're best off running the function on the client using straight MATLAB.

Also, keep in mind that every time the client and the server communicate, this 50mS penalty is paid by your program. Therefore, you are best advised to write your program in such a way that the client and server communicate as little as possible. Again, vectorizing your program is the key tactic for optimizing your code since a vectorized operation doesn't communicate until it is complete.

**Cluster vs. Shared Memory Supercomputers**

Recall that in data-parallel computations your matrix is spread over all processors in the parallel computer, and during calculation intermediate results are communicated amongst the compute nodes. Therefore, if your program uses lots of data-parallel constructs, then your parallel server must provide a high-bandwidth, low latency communication channel between its compute nodes. This usually means that you need a shared-memory system – or an equivalently tight-coupled machine – as your parallel server. The increasingly popular Beowulf cluster – in which inter-processor communication takes place over an Ethernet network – will possibly suffer from large communication latencies, making it a poor choice for running data-parallel algorithms.

On the other hand, if your program is extremely task-parallel, in which each individual task takes many minutes, hours, or longer to complete, then either a Beowulf cluster or a shared memory system present a sensible choice for your computer. Indeed, in the limiting task-parallel case – where each independent task may take days or weeks to complete – a Beowulf cluster might make more sense than a tightly-coupled parallel machine. The reason is that if one of the cluster's compute nodes fails, you can always remove and replace that one node while the remaining compute nodes carry on with your calculation. However, with a tightly-coupled system, if one processor dies then you may need to shut down the whole machine to replace the dead node, prematurely terminating your entire calculation.
Sharpen Your Tools: Star-P Settings for Maximum Performance

To be good craftsman, it's important to sharpen your tools so they give you their best performance. The following Star-P settings are useful when you need to squeeze the last little bit of performance from your code. Changing these settings will not magically make your program run twice as fast – rather, they might give you a speed improvement of a few percent. If your code takes weeks to complete a single run, then changing these settings may save you a few hours. In any event, you are best off first prototyping and developing your code with Star-P's default settings, and then changing these settings when you are ready to go into production crunching your largest datasets.

- **ppsetoption('log','off')** By default, each call to the Star-P server causes an entry in a log file on the server system. Some performance benefits can be achieved by disabling logging in the server since the HPC must no longer spend time writing information to the log. This is particularly useful since file IO can be time consuming (compared to pure RAM access operations, that is).

- **ppsetoption('mallochooks','off')** By default, StarP will run a series of checks and sanity tests before malloc'ing memory for your function. These checks could take a few additional processor cycles. By setting this option to 'off', you eliminate the (minimal) overhead time associated with this checking. However, you increase the risk that your session will simply crash instead of warning you about memory problems.

- **ppsetoption('YieldCPU', 'off')** By default, when your session is inactive the StarP server will give up ("yield") the CPU to be used by other processes on the HPC machine. This results in the client encountering a slight delay (on the order of a second) the first time it contacts the server after a period of inactivity. To avoid such delays, set this option to 'off', which will tell the StarP server to keep the CPU in an idle loop even when the client is idle.

- **ppsetoption('ppgcFreq', 100000)** This sets the time interval between garbage collections to a very large value. That way, your program's run won't be interrupted by the garbage collector. However, since no garbage collection is performed, you will need to be extra conscious of your program's -- and your StarP session's -- memory utilization.
An interesting recent development in the practice of software development has been the adoption of the notion of “design patterns”, a concept originating from the world of architecture. In brief, a design pattern is a template solution to a particular problem known to arise in the course of design. It would be difficult to rigorously define a set of patterns for you to use when parallelizing your code under Star-P. However, much of the benefit of design patterns may be obtained by looking at examples showing how certain types of code may be successfully parallelized. Therefore, below are presented some patterns showing you various examples of how serial MATLAB code was successfully ported to Star-P.

### Data-parallel: Crunching Large Matrices

If your program performs lots of atomic mathematical operations on gigantic matrices as a whole, then your algorithm is best described as data-parallel. Data-parallel functions available to you in Star-P include the usual binary operators (+, -, *, / and so on), as well as all of MATLAB’s basic matrix handling functions like inv(), norm(), and so on.

A classic data-parallel operation is the Fourier Transform. Here's an example in which we use a two-dimensional Fourier Transform to filter an image.

```matlab
% SpatialFilter.m -- Reads image, then does 2D filtering operation
% via Fourier Transform and zeroing high-frequency elements.
load('durer.mat');  % Loads B&W image matrix called X and colormap
% called map
colormap(map);      % set color map for remainder of program.
Xpp = ppback(X);   % Move matrix to server.  All subsequent operations
% will be data-parallel and occur on server

% Take Fourier Transform and move zero frequency to middle of matrix
Xfpp = fft2(Xpp);
Xfpp_shift = fftshift(Xfpp);

% Now do filtering operation
imgsize = size(Xfpp_shift);
RowCenter = floor(imgsize(1)/2);
ColCenter = floor(imgsize(2)/2);
```
% Keep only 2n+1 x 2n+1 Fourier coeffs. Set remainder to zero.
\( n = 30; \) % number of coeffs to keep.
Yfpp_shift = zeros(imgsize);
Yfpp_shift(RowCenter-n:RowCenter+n , ColCenter-n:ColCenter+n) = ...  
Xfpp_shift(RowCenter-n:RowCenter+n , ColCenter-n:ColCenter+n);

% Inverse transform image
Yfpp = fftshift(Yfpp_shift);
Ypp = abs(ifft2(Yfpp));

% Move filtered image to client and display it.
Y = ppfront(Ypp);
image(Y)

Before and after images are shown in Figure 6 below. The nice feature of Star-P is that once you have moved your data to the server, then you may manipulate it using all the standard MATLAB functions. That is, you are able to crunch your large matrices implicitly using data-parallelism, without doing any more work than “ppback”!

![Figure 6](image)

**Figure 6**: On the left is the black and white image “Melencolla I” originally drawn by the German artist Dürer. This image is found in the “examples” directory of the standard MATLAB installation. On the right is the image after processing using the data-parallel spatial filtering code presented above.

**Data-parallel: Interacting Bodies and Swarms**

Another class of problems which might benefit from a parallel computer include dynamical simulations containing large numbers of interacting elements. Examples of this problem class include stellar dynamics, molecular dynamics, swarms, neural nets, and the like.

The behavior of such systems is governed by some kind of force or coupling law between individual elements under simulation. The coupling is generally expressed as an interaction matrix. Therefore, such problems can be treated as data parallel, since the
interactions between the individual elements are commonly expressed using a gigantic matrix which must be updated at every computational step. As long as the problem can be expressed appropriately, it makes sense to spread the matrix across all processors available, and then use MATLAB primitives to operate on the matrix in data-parallel mode. The challenge with porting such applications to parallel computers lies in finding an efficient method to compute the force matrix which exploits the parallelism of the problem without incurring too much communication overhead between the different compute nodes. As discussed above, vectorizing your code is a critical component in achieving this goal.

A good example of a data-parallel, interacting body problem is a simple simulation of a galaxy of stars interacting via gravitational attraction in two dimensions. Such a calculation is a natural candidate for parallelization when you want to scale the size of the galaxy up to many, many stars.

For reasons outside of the scope of our discussion here, gravity in two dimensions is mediated by a force proportional to $1/r$, where $r$ is the distance between two bodies. (Normal three dimensional gravity obeys a $1/r^2$ law.) The interactions of a large number of stars can be expressed using a force matrix, where the $(i, j)$th element of the matrix represents the force felt by the $i$th star due to its attraction to the $j$th star. The positions and velocities of the stars themselves are held in row vectors. In this case, Newton's law says that the motion of each star in the galaxy can be expressed as

$$m_i \frac{d^2 \vec{x}_i}{dt^2} = \sum_j \vec{F}(i, j)$$

where the magnitude of the force is

$$F(i, j) = \frac{G m_i m_j}{|\vec{x}_i - \vec{x}_j|}$$

Expressing these equations in MATLAB is pretty easy. However, expressing them in a vectorized way is harder. Here's some non-vectorized MATLAB code which starts with a bunch of stars in random positions, then loops over time, first calculating the force matrix, then updating each star's position as it moves under the gravitational force exerted upon it by all the other stars. In this simulation, all stars are taken to have the same mass, although that restriction can easily be relaxed without causing any problems in the code.

```matlab
% Stars -- Simple simulation of 2D galaxy of point stars
%         interacting via 1/r gravity.
numsteps = 100;  % Number of time steps to simulate
delta_t = 1;
m = 10;          % Mass of star
G = 1e7;         % Gravitational constant
numstars = 300;  % Number of stars in galaxy
```
% First create random initial positions
x = 6e6 * (rand(1, numstars) - 0.5); % row vector
y = 6e6 * (rand(1, numstars) - 0.5);

% Give stars zero initial velocities
vx = zeros(1, numstars);
vy = zeros(1, numstars);

for time = 1:numsteps
  % compute force vector matrix for each star
  for i=1:numstars
    for j=1:numstars
      if i==j
        F(i, j) = 0;
      else
        F(i, j) = G/sqrt( (x(i) - x(j))^2 + ... 
                        (y(i) - y(j))^2 ); % 2D Gravity goes as 1/r
        theta(i, j) = atan2( (y(j) - y(i)), (x(j) - x(i)) );
        Fx(i, j) = F(i, j) * cos(theta(i, j)); % Resolve force vector into
        Fy(i, j) = F(i, j) * sin(theta(i, j)); % x and y directions.
      end
    end
  end

  % Now compute change in velocity
  for i = 1:numstars
    delta_vx(i) = (delta_t/m) * sum(Fx(i, :));
    delta_vy(i) = (delta_t/m) * sum(Fy(i, :));
  end
  vx = vx + delta_vx;
  vy = vy + delta_vy;

  % Update positions
  x = x + delta_t * vx;
  y = y + delta_t * vy;
end % Loop to next time step

The meat of this code occurs in the double loop over i and j. Inside that loop, both the magnitude and the direction of the gravitational force experienced by the ith due to the jth particle is computed. (The force is resolved into Fx and Fy components.)

At first glance, this calculation doesn't look vectorizable. However, it is, using a trick. The trick involves using "repmat" to create two matrix copies of the x and y vectors – one the transpose of the other – and then use them when doing the subtractions. Here's the code after vectorization, and after it has been ported to Star-P. After vectorization, the only additional step required to run this code on the parallel server is to tag the variable "numstars" with a *p.

% Stars -- Simple simulation of 2D galaxy of point stars
% interacting via 1/r gravity.

numsteps = 100; % Number of time steps to simulate
delta_t = 1;
m = 10; % Mass of star
G = 1e7; % Gravitational force constant
numstars = 300*p; % Number of stars in galaxy

% -------- First create random initial positions --------
x = 6e6 * (rand(1, numstars) - 0.5); % row vector
y = 6e6 * (rand(1, numstars) - 0.5);

% ------- Give stars zero initial velocities -------
vx = zeros(1, numstars);
vy = zeros(1, numstars);

for time = 1:numsteps
  % ------- compute force vector matrix for each star -------
  % Vectorized calc.
x1 = repmat(x, [numstars 1]);
x2 = repmat(x', [1 numstars]); % transposed
y1 = repmat(y, [numstars 1]);
y2 = repmat(y', [1 numstars]); % transposed

  F = G*m^2 ./ sqrt( (x2-x1).^2 + (y2-y1).^2 ); % 1/r gravity
  DiagIndex = 1:(numstars+1):numstars^2; % Main diag indices
  F(DiagIndex) = zeros(numstars, 1); % Set them to zero.

  % Compute vector force on each star due to each other star
  theta = atan2( (y2 - y1), (x2 - x1) ); % 4 quadrant arctan
  theta(DiagIndex) = zeros(numstars, 1);
  Fx = F .* cos(theta);
  Fy = F .* sin(theta);

  % ---------- Now update star's velocity ----------
delta_vx = (delta_t/m) * sum(Fx);
delta_vy = (delta_t/m) * sum(Fy);
  vx = vx + delta_vx;
  vy = vy + delta_vy;

  % ---------- Now update positions ----------
x = x + delta_t * vx;
y = y + delta_t * vy;
end

A sample run showing the positions of 300 stars after 200 simulation steps is shown in Figure 7. As you might imagine, 300 stars forms a pitifully small galaxy! The advantage of running this simulation as a data-parallel calculation under Star-P is that you can scale the up number of stars dramatically by running the calculation on a parallel supercomputer.
Many simple reduction and scan operations can be efficiently performed using data-parallel primitives. But first, what are reduction and scan operations?

- A reduction operation takes a vector argument, sequentially performs an associative binary operation on pairs of the vector's elements, and returns a scalar result. Example reductions include addition, multiplication, min, max, logical and, and so on.
- A scan operation takes a vector argument, operates from left to right on the vector elements with an associative binary operator, and returns another vector. Example scan operations include "cumsum" and "cumprod".

Star-P provides many data-parallel primitives implementing reduction and scan operators, including "cumsum", "cumprod", "min", "max" and so on. If you can recast your algorithm to use one of these functions, you will experience a significant time speed-up.
Here's an example, in which we generate a series of partial sums which converge towards the numerical value of pi. The top level program generates a vector of term indices (i.e. n = 1, 2, 3, 4, etc), and passes this vector to a function which calculates the value of the nth term. Then, the scan operator "cumsum" is used to calculate the partial sum after n terms. Finally, we plot the residuals (residual error) – the difference between our series sum and the actual value of pi.

```
% PiSeries_DP.m -- Creates row vector whose elements
% are the partial sum of a series expansion for Pi.

% This is data-parallel implementation
max_n = 50000*p; % Number of terms to calc
n_vec = 1:max_n;
term_vec = term_dp(n_vec-1); % calc series expansion terms
partial_sum_vec = 4*cumsum(term_vec); % do partial sum

% Check results against actual value of pi
pi_vec = repmat(pi, 1, max_n);
residuals_vec = partial_sum_vec - pi_vec;

% Plot convergence
semilogy(residuals_vec)

% Sub-function
function x = term(n)
    x = ((-1).^n) ./ (2.*n + 1);
    return
```

There are no tricks in this program, except that the sub-function must be computed using element-wise binary operators (i.e. ".^" and ".*/") to support vectorized calculations.
The results are shown in Figure 8. We used a log-log plot to show the residuals; it shows the convergence trend quite clearly. Although 5e4 terms to calculate pi is not large enough to justify using a supercomputer, imagine if you have a more complex, slowly converging sum to calculate! In that case, using Star-P to perform cumsum on a parallel machine might well be the only way to calculate enough partial sums to get an answer.

**Task-parallel: Unrolling a Loop – Generating the Mandelbrot Set**

If your program consists of a large function or operator applied to independent data, it is best described as task-parallel. One particularly common variant of this type of program involves a function applied to independent sections of a gigantic matrix -- for example, a filter operation applied to each row or column of the matrix, as in data averaging, or a 2D operation applied to slices of a 3D array, as in video image processing. Another example might be a computationally-intensive function applied to each element in a 2D matrix, for
example computing a Mandelbrot set. (shown in Figure 9)

The strict definition of the Mandelbrot set is technical. For our purposes, we'll gloss over the technical details and note that it involves independently iterating points chosen from the complex plane, and recording how long it takes each iterated point to escape a set bound. Since we perform this iteration for each point in the complex plane, a serial algorithm usually performs a “for” loop over all points in the plane. As we shall see, this “for” loop may be parallelized.

A more detailed description of the Mandelbrot set algorithm runs like this:
1. Take a patch of the complex plane bounded by (-1 < real part < 2.5) and (-1.5 < imaginary part < 1.5). Call it c.
2. For every point in c, iterate the following function for some number of times: \( z = z^2 - c \). For the initial iteration, use \( z = 0 \). With each iteration, increment a counter.
3. Upon each iteration, check that the absolute value of \( z \) remains less than some threshold. In our case we'll choose a threshold of 100. If \( z \) exceeds the threshold, stop the iteration.
4. Stop iterating after some maximum limit if \( z \) never exceeds the threshold. In our case, we choose 500 iterations as the limit.
5. The result of this procedure is a matrix of integers -- one for each point on the complex plane which is part of c. Each integer tells how many iterations it took for that value of c to exceed the threshold. To produce the pretty Mandelbrot set image, we plot this matrix using a different color for each integer value in our matrix.

Here's some serial MATLAB code implementing this algorithm.

```matlab
% PlotMandelbrot -- makes color plots of Mandelbrot set.
% This is serial version.

% Set up the portion of the complex plane of interest.
crmin = -1;
crmax = 2.5;
cimin = -1.5;
cimax = 1.5;
numpoints = 400;       % Distretization in x and y
maxiterations = 500;   % max times to iterate each point.

escape_threshold = 100;

% Create c
crvec = linspace(crmin, crmax, numpoints);
civec = linspace(cimin, cimax, numpoints);
[crmat, cimat] = meshgrid(crvec, civec);
c = crmat + i*cimat;

% Initialize image which will be colored depending upon the number of 
% iterations until escape.
Mandelbrot = zeros(numpoints, numpoints);

% Iterate each point in c until it escapes (or doesn't)
for x=1:numpoints
    for y=1:numpoints
        z = 0;
        for i=1:maxiterations
            z = z^2 - c(x, y);
            if abs(z) > escape_threshold
                break;
            else
                Mandelbrot(x, y) = Mandelbrot(x, y) + 1;
            end
        end
    end
end
```

- 40 -
end

% Display beautiful color plot.
image(crvec, civec, Mandelbrot)

The problem with performing this calculation serially is that it involves a double iteration (in x and y) over a third calculation -- that of iterating the z values. This is very compute-intensive. Since each z iteration for different values of x and y is independent, this function is a perfect candidate for task-parallel computation. Here's the same program, written as a task-parallel algorithm under Star-P.

% PlotMandelbrot -- makes color plots of Mandelbrot set.
% This is task-parallel version

crmin = -1;
crmax = 2.5;
cimin = -1.5;
cimax = 1.5;

numpoints = 400;
maxiterations = 500;
escape_threshold = 100;

% Create c
crvec = linspace(crmin, crmax, numpoints);
civec = linspace(cimin, cimax, numpoints);
[crmat, cimat] = meshgrid(crvec, civec);
c = crmat + i*cimat;

% Iterate each point until it escapes (or doesn't). Use ppeval to
% replace nested "for" loops. Split upon axis 0 to send each
% matrix element separately. Return is one big row vector, must
% reshape before using.
Mandelbrot = ppeval('IteratePoint', split(c, 0), ...
            maxiterations, escape_threshold);

% create square matrix from return
Mandelbrot = ppfront(reshape(Mandelbrot, numpoints, numpoints));

% Display beautiful color plot.
image(crvec, civec, Mandelbrot)

The sub-function "IteratePoint", which does the individual iterations, is this:

function iterations = IteratePoint(c, maxiterations, escape_threshold)
z = 0;
iterations = 0;
for i=1:maxiterations
    z = z^2 - c;
    if abs(z) > escape_threshold
        break;
    else
        iterations = iterations + 1;
    end
end

The important thing to recognize in the parallel code is that the double loop has been replaced by a “ppeval” call. The “ppeval” call sends the matrix “c” into individual
elements (split axis 0), so that each process node receives a different “c”. When “ppeval” is done iterating all the values of “c”, it returns one long vector containing the number of iterations. We must call “reshape” to return the matrix to its original square shape. Then we bring the matrix to the client and plot it to get the beautiful image shown in Figure 9.

**Task-parallel: Monte Carlo and other statistical algorithms**

The canonical example of a task-parallel operation is the Monte Carlo algorithm, commonly used when simulating events which depend upon randomness. In Monte Carlo, one ordinarily uses a "for" loop to iterate over separate trials in a serial algorithm. However, since each trial is independent, Monte Carlo programs are very easily parallelized using Star-P’s “ppeval” function.

Here’s an extremely simple Monte Carlo example in which we toss a coin 100 times, and then see how many times we get heads. Obviously, the laws of probability say that we should get 50 heads on average. However, since each coin toss is an independent probabilistic event, we might get 53 heads, or 48, or some other number with every trial. Indeed, there is always the possibility – however small – that we’ll throw the coin 100 times and get 100 heads! To investigate the randomness of tossing a coin we can use a simple Monte Carlo program. In our program we throw 100 coins, and count the number of heads. Then, we repeat this trial ten thousand times, and plot a histogram showing the number of times each sum occurred. The serial code looks like this:

```matlab
% serial code.
Ntrials = 10000;
Ncoins = 100;

TossHistory = [];
for i=1:Ntrials
    CoinToss = 2*rand(Ncoins, 1) – 1;
    Nheads = sum(CoinToss > 0); % Vectorized coin toss.
    TossHistory = [TossHistory, Nheads];
end
hist(TossHistory, 30:70)
```

As a task-parallel algorithm under Star-P, we would move the inner body of the loop into a separate function, and then invoke it using "ppeval". Here’s the top level code:

```matlab
% task-parallel code.
Ntrials = 10000;
Ncoins = 100;

i=1:Ntrials;
TossHistory = ppeval( 'DoToss', bcast(Ncoins), split(i) );

hist(TossHistory, 30:70)
```

and the function which performs each trial (i.e. 100 coin flips) is this:

```matlab
% perform coin tosses, return number of heads obtained.
```
function Nheads = DoToss(Ncoins, seed)
    rand('state', seed);
    CoinToss = 2*rand(Ncoins, 1) - 1;
    Nheads = sum(CoinToss > 0);

Running this code under Star-P produces the histogram shown in Figure 10.

There is a very interesting -- and important -- technique to notice in the Star-P code. In the original, serial code, we never seeded the random number generator. We didn't need to since the random number generator gave a new random number each time "rand" was called. However, when using Star-P's "ppeval" to evaluate functions in parallel on different compute nodes, each function's random number generator has an identical seed. Therefore, the "random" processes running on different processors will give identical behavior! Therefore, to run Monte Carlo functions on a parallel computer where each process is statistically independent from the others, we need to provide each Monte Carlo trial with an independent random seed from the top level calling program. That's what we are doing with the (former) iteration variable "i" -- when calling the random trial subfunctions, we pass each instance a different random seed by splitting "i" in the ppeval call, and using the received value as each processes' random seed.

![Figure 10](image)

**Figure 10:** Histogram formed by tossing a coin 100 times and summing the number of heads. This experiment is performed 10000 times, and the different counts achieved are plotted in a histogram. As expected, the histogram is gathered around count = 50, corresponding to equal probability of tossing heads or tails. Parallelization of this algorithm involves using "ppeval" to run separate trials on different compute nodes.
The Right Tool For the Job: Parallelization Using Star-P

Star-P was developed to enable you to run your MATLAB applications on a parallel computer with only some minor tweaks to your code. To achieve this goal, Star-P offers a variety of different methods to achieve optimal run-time performance. Whether through implicit data-parallel operations, explicit task-parallel function invocation, or some combination of both, Star-P has a solution for any particular situation. In this sense, Star-P is like a toolbox full of tools, each of which has been adapted and honed to serve its specific purpose. Like any craftsman, it’s up to you, the programmer, to know which tool is appropriate for the task at hand. We hope to have shown you that learning which Star-P tool to use in any situation is quite easy, and that after only a little bit of practice you will be a master craftsman of parallel numerical programs!
Appendix: Online Star-P Resources

Interactive Supercomputing:
- [http://www.interactivesupercomputing.com/support/](http://www.interactivesupercomputing.com/support/) -- Interactive Supercomputing’s Support Group login page. The support pages provide Star-P documentation, code tutorials and examples, and other pedagogical materials.

Parallel Computing Basics:
Information about parallel computing in general. Most online resources tend to focus on low-level details (i.e. Parallel architectures and libraries like MPI) which Star-P handles automatically. However, it never hurts to have an understanding of these deeper issues.

MATLAB Resources:
- Sources useful for MATLAB code (This list is not meant to be all-inclusive):
  - [http://matlabdb.mathematik.uni-stuttgart.de/index.jsp](http://matlabdb.mathematik.uni-stuttgart.de/index.jsp)
  - [http://xtargets.com/snippets](http://xtargets.com/snippets)
- E-mail group (requires subscription): [http://tech.groups.yahoo.com/group/matlab/](http://tech.groups.yahoo.com/group/matlab/)
- MATLAB Tutorials, Tips, and tricks:
  - [http://web.cecs.pdx.edu/~gerry/MATLAB/masterOutline.html](http://web.cecs.pdx.edu/~gerry/MATLAB/masterOutline.html)
- Usenet: comp.soft-sys.matlab – Discussion board with lots of useful information.

Octave Resources:
Octave is a GPLed mathematics system very similar to MATLAB. Oftentimes, Octave code will run under MATLAB without modification (and vice-versa).
- [http://www.gnu.org/software/octave/](http://www.gnu.org/software/octave/) -- Project home page,

Scilab Resources:
Scilab is an open-source (albeit non-GPL compatible) mathematics system similar to MATLAB. Unlike Octave, the language syntax does not exactly mimic MATLAB.
About Interactive Supercomputing

Interactive Supercomputing Corporation (ISC) develops Star-P, a software platform that drives productivity by significantly increasing application performance while keeping development costs low. Across a broad range of security, intelligence, manufacturing, energy, biomedical, financial, and scientific research applications, ISC enables faster prototyping, iteration, and deployment of large-scale solutions on high performance computing servers.

Star-P was originally developed at the Massachusetts Institute of Technology, with support from the National Science Foundation. ISC was launched in 2004 to commercialize Star-P, holds an exclusive license from MIT to commercialize the technology, and has independently filed multiple patents. Since its launch in 2005, Star-P has been adopted at leading government labs, research institutions, and commercial enterprises.