Getting Started with Star-P: Taking Your First Test-Drive

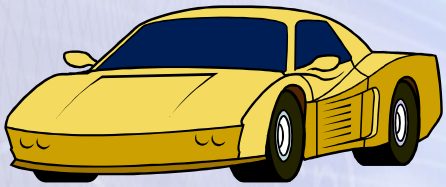

Table of Contents

You're Handed the Keys – Introduction

You're a MATLAB user, and have heard computer industry buzz about Interactive Supercomputing's Star-P software. You understand that it allows you to exploit the power of a parallel computer to speed up your MATLAB programs, but don't know the details. However, today – for the first time – have access to a machine running Star-P. Therefore it's time to buckle up and take a test-drive of the Star-P system! This "getting started" guide is meant to give you the first-time driver's experience of using Star-P, showing you how Star-P can bring supercomputer power to bear on your own MATLAB calculations. Follow the examples, and try them on your own workstation to see what Star-P can do! If you have detailed questions while taking this test-drive, please consult the detailed Star-P documentation available on Interactive Supercomputing's website.

Engines on! Starting your first interactive Star-P session

One of the design goals of Star-P is to produce an interactive environment for mathematical computing comfortable and familiar to long time users of MATLAB. Therefore, starting Star-P is nearly identical to starting MATLAB. Depending upon whether your local workstation runs Windows or Linux, do this:

- **Windows**: You will find a Star-P icon under your "Start -> Programs" button. Most installations will also place a Star-P icon on your Windows desktop. Click on one of these icons to start your Star-P session.

- **Linux**: You can start Star-P by typing the fully qualified path to the Star-P executable (something like "/usr/local/bin/starp") at a shell prompt, along with command flags pertinent to your local installation. However, your local system administrator will likely have written a script of a different name allowing you to easily start Star-P; consult him to see how Star-P has been configured on your system.

Either way, once you've started it, Star-P will open up your familiar MATLAB environment exactly as you've always used it. The only thing telling you that you are running Star-P is the Star-P copyright notice appearing in the middle of the command window. (See Figure 1).

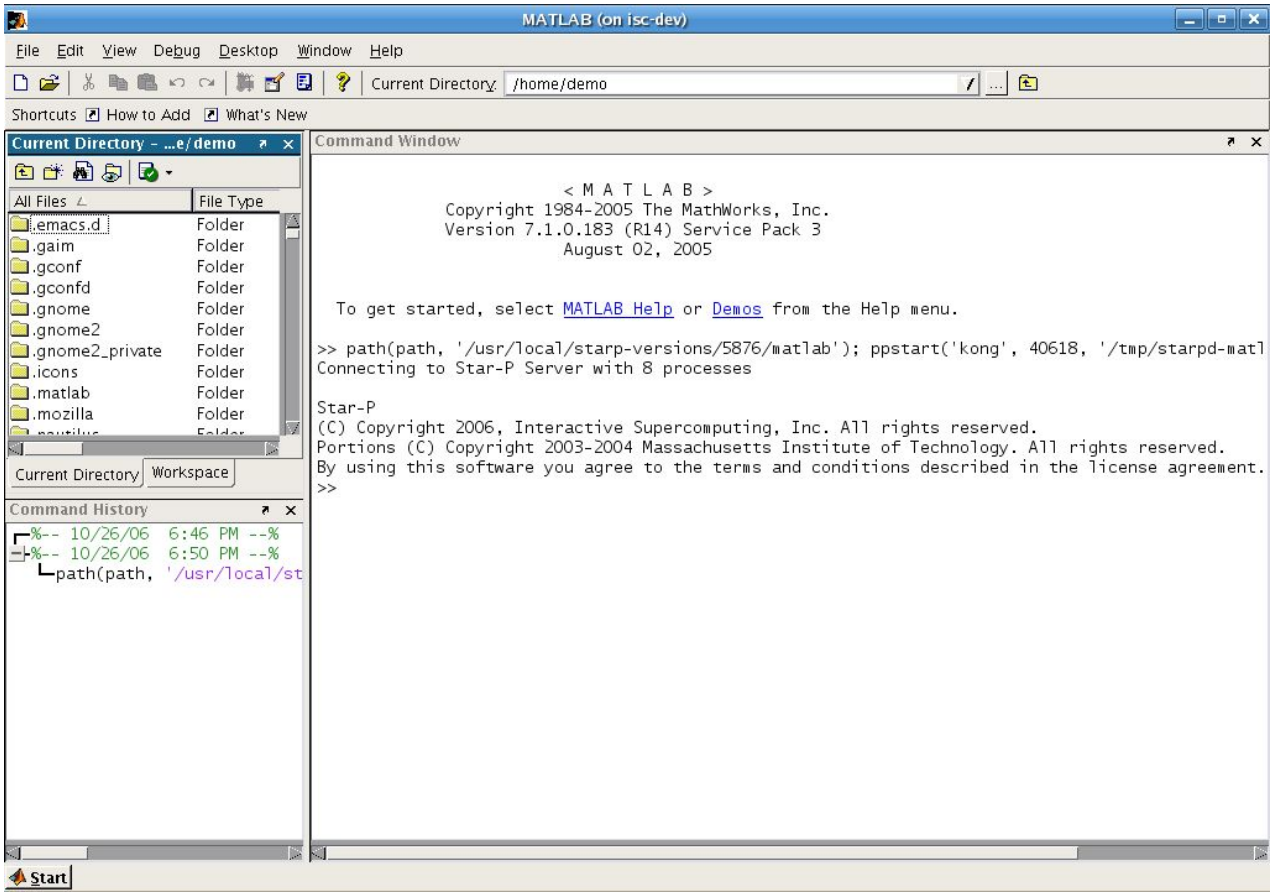

***Figure 1**: As you can see in this screenshot, Star-P runs inside the MATLAB GUI environment which you are used to.*

As evident in the screenshot, starting Star-P gives you the usual MATLAB working environment, including the command window, the directory window, and the command history window. Indeed, the GUI ***is*** the MATLAB GUI – Star-P runs inside of MATLAB's working environment. You never need to leave your familiar MATLAB environment to access the power of Star-P!

Shift to First and Push the Pedal! Your First Star-P Calculations

Now that we've fired up a Star-P session, let's try running some calculations on the supercomputer. Understanding what is happening during computation requires visualizing the Star-P hardware setup, which is shown in Figure 2. As you probably already know, a typical Star-P setup includes your client PC (where you work interactively) and a back-end server HPC (a parallel computer of some sort) linked via a LAN connection. Since the client and server are connected via a network connection, the server can sit on the same desk as the client, or it can sit down the hall in a server room, or it can live on the other side of the globe. Star-P software runs on both the client and the server, mediating exchange of commands and data between the two computers.

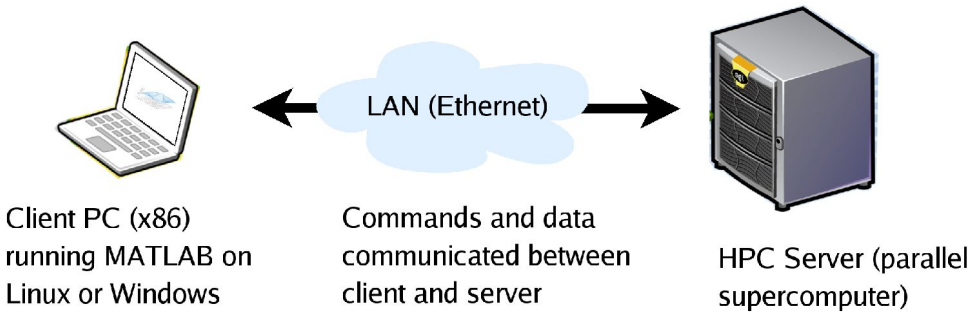

***Figure 2**: Star-P hardware components. The Star-P client (left) is typically your desktop computer. The Star-P server (right) is the High Performance Computer (HPC) on which the numerical libraries reside. The two computers are normally connected via a LAN connection. Star-P software runs on both the client and the server, and seamlessly mediates the communication between the MATLAB session running on the client and the numerical libraries running on the server.*

While interacting with your MATLAB session, Star-P will perform operations on both systems. To continue the automotive theme, your Star-P session is like a hybrid car which is powered by both an electric motor and a gasoline engine. Sometimes the electric motor turns the drive wheels, and sometimes the gasoline engine turns them depending upon driving conditions. Analogously, Star-P performs some calculations on your local client PC, and some calculations on a High-Performance Server computer (HPC); the magic of Star-P is that it knows which platform to use for each operation.

"So how", you may ask, "does Star-P know when to perform a calculation on the client, and when to perform it on the server?" The answer is: "It depends upon where the variables involved live". And where they live depends upon several things – most importantly how the variables were initially created.

First off, operations performed on only scalar variables are always performed on your local client PC[1] -- that's because scalars always live on the client. However, non-scalar variables (vectors, matrices, and arrays) can be manipulated on either the client or the server under Star-P. Therefore, when you start a new calculation, you need to tell Star-P where you want your initial non-scalar variables to live. To create matrices on the PC client, you just work in the usual way. But when you want to create a matrix for manipulation on your supercomputer, Star-P provides you special syntax -- the "*p" tag -- which you place on one of the dimensions of the matrix you are creating. The "*p" tag tells Star-P to create this matrix on the back end supercomputer. It additionally tells Star-P how to distribute the values of your matrix over the various compute nodes available on the server.

Here's a first example. Let's say you want to create a large random matrix, and then invert it. In MATLAB you would do this:

```
>> a = rand(100, 100);
>> b = inv(a);
```

1 Indeed, it only makes sense to use Star-P when you need to manipulate large non-scalar data; parallel computers generally don't offer much advantage over serial computers for simple scalar computations.

This syntax will still work under Star-P. It will simply create and invert your matrix on the local client PC, the same as you have always done.

To perform this matrix inversion on the server under Star-P, you would do almost the same thing – with the minor exception that you would use the "*p" tag when creating your matrix, like this:

```
>> a = rand(100, 100*p);
>> b = inv(a);
```

The "*p" tag tells Star-P that "a" should be created on the supercomputer server, and that MATLAB/Star-P can find the variable "a" held on the back-end server whenever it needs to use it in a calculation -- for example, when you calculate its inverse as shown in the next line above.

Let's take a quick detour and examine some features of the "*p" tag. Notice that we placed the "*p" on the second (column) index of "a". This signals Star-P to "distribute" the values of "a" column-wise on the parallel computer. That is, when Star-P creates the matrix on the back-end parallel computer, it creates a different column on each compute node. By the way, the number of columns and number of processors don't have to match. Star-P allocates the matrix columns amongst the different nodes as fairly as it can -- similar to how a card dealer might deal an entire deck of cards to different players in a card game.

Suppose you wanted to row distribute "a" (i.e. give each processor node a different row). To do so, you would say this:

```
>> c = rand(100*p, 100);
```

As you can see, to row-distribute the matrix, you place the "*p" tag on the first (i.e. row) index of "a".

Now let's get back on the main road. Once "a" is created on the supercomputer, it may be manipulated on the supercomputer. This is exemplified by the next step shown above, in which we take the inverse of "a", and assign it to "b". Since "a" lives on the server, the matrix "b" created from "a" will also live on the server. This is another feature of Star-P's operation: Non-scalar variables created from server variables also live on the server! You can think of server variables as *infecting* any new variables they create, and passing the server property on from generation to generation.

Therefore, the key to knowing whether Star-P performs computations on the client or on the server is to know where the variables it is operating upon live. And usually that is self-evident since the place where the variables live depends upon how they are created during the program flow. Let's say, however, that you're in an interactive MATLAB session, and you don't remember where a particular variable lives. How to find out where it lives? One way is to just type its name at the command line. As always with MATLAB, a variable living on the local client machine will be displayed, like this:

```
>> a = rand(5, 5)

a =

    0.9501    0.7621    0.6154    0.4057    0.0579
    0.2311    0.4565    0.7919    0.9355    0.3529
```

```
    0.6068    0.0185    0.9218    0.9169    0.8132
    0.4860    0.8214    0.7382    0.4103    0.0099
    0.8913    0.4447    0.1763    0.8936    0.1389
```

But a variable living on the server will have its *type* displayed instead:

```
>> b = rand(5, 5*p)

b =

        ddense object: 5-by-5p
```

The reason for this is that in common practice server variables are usually giant vectors or matrices which are simply too large to display reasonably. Therefore, Star-P will only display their type, which is usually "ddense" or "dsparse" -- the "d" prefix reminding you that the matrix is "distributed" on the HPC. Here's the matrix creation and inversion example from above without the silencing semicolon at the end of each line:

```
>> a = rand(100, 100*p)

a =

        ddense object: 100-by-100p
>> b = inv(a)

b =

        ddense object: 100p-by-100p
>> c = rand(100*p, 100)

c =

        ddense object: 100p-by-100
```

As you can see, Star-P reports the matrices' type, size, and distribution after each operation. In particular, comparing the report output for variables "a" and "c" you can see that "a" is column distributed, and "c" is row distributed by the position of the "p" in the matrix dimensions.

Finally, let's create some vectors in Star-P. You can create such a vector using StarP in a couple of different ways: Either using the colon operator (":") or using one of MATLAB's convenient vector creation functions, like "linspace". In either case, creating a vector in Star-P simply involves tagging one of the vector's dimensions with "*p" like this:

```
>> delta_time = 0.000001;
>> end_time = 10;
>> numpoints = end_time/delta_time + 1;
>>
>> % ----- Method 1
```

```
>> time_vector1 = 0:delta_time:end_time*p

time_vector1 =

        ddense object: 1-by-10000001p
>>
>> % ----- Method 2
>> time_vector2 = linspace(0, end_time, numpoints*p)

time_vector2 =

        ddense object: 1-by-10000001p
>>
>> % ---- Looks good, but are they the same?
>> sum(time_vector1 - time_vector2)

ans =

     0
```

As you can see, these two vector creation methods are equivalent, and work equally well.

In the Rearview Mirror – Section summary

- Computations occur on either client or server depending upon where the variables live.
    - Scalar variables always live on the client.
    - Non-scalar (i.e. vector, matrix, or array) variables can live on either the client or the server depending upon where they were created.
- Use the “*p” idiom to create non-scalar variables living on the server.
- Depending upon where you place the “*p” tag, 2 dimensional matrices you create with be either row or column distributed.
- Server variables are "infectious". If you create a new non-scalar variable from a server variable, it will also live on the server.

Shift into Second Gear – Binary Operations

Now that we understand the basics of creating and using non-scalar variables in Star-P, let's try a binary operation:

```
>> a = rand(100, 100*p);
>> b = rand(100, 100*p);
>> c = a*b

c =

        ddense object: 100-by-100p
```

As you can see, since "a" and "b" are server variables, their product "c" also lives on the server. In general, if your operation produces a vector, matrix, or array, and you operate upon server variables, the result will

also live on the server. Remember – the server property of a variable is infectious, meaning that operating on server matrices will create server matrices. Only if your operation returns a scalar will the result live on the client. In this case, the numerical result will be displayed. You've already seen one example in the last section (using “sum”). Here's another example:

```
>> d = rand(250*p, 250)

d =

        ddense object: 250p-by-250
>> sum(sum(d))

ans =

   3.1270e+04
```

Another instructive example of Star-P's behavior for scalar vs. higher-dimensional results can be seen in multiplication of vectors, as shown here:

```
>> % ----- Try multiplication of vectors
>> X = 1:1000*p

X =

        ddense object: 1-by-1000p
>> Y = rand(1, 1000*p)

Y =

        ddense object: 1-by-1000p
>>
>> % ----- Element-wise multiplication should yield vector
>> X .* Y

ans =

        ddense object: 1-by-1000p
>>
>> % ----- Scalar product should yield scalar
>> X * Y'

ans =

   2.5957e+05
```

You can try the various binary operators out for yourself on vector and matrix data to see what happens. As a general rule, Star-P will return the same result (i.e. same dimensionality and element value(s)) as you would get from MATLAB for each binary operation.

Rearview Mirror – Section summary

- Binary operations resulting in a scalar will return a scalar living on the client.
- Binary operations resulting in a non-scalar will return a variable living on the server. Put another way, the server property is infectious.
- Star-P's binary operations work exactly the same way as MATLAB.

The Instrument Panel – Looking at Your Data

Since Star-P only reports the size and distribution of a variable held on the server, you may well ask, "How can I look at the value of my matrix if it is held on the back-end supercomputer?" The answer is: generally you **don't** want to look at it. Indeed, large matrices are usually intermediate results of an ongoing calculation; the result you actually want to see at the end of processing may be just a handful of scalars, or a graph of some sort. Moreover, a key condition for Star-P to efficiently parallelize a MATLAB program is that the program should import or create its variables on the server at the beginning of execution, manipulate them there, and then bring the results to the front-end client for visualization or analysis at the end of processing. Since moving data between the server and the client involves communication time, unnecessarily moving data around -- particularly large, non-scalar objects -- can degrade the performance of your program, which is not the effect you would like when using in Star-P!

We will not discuss graphic visualization of data in great detail in this introductory tour. As far as just looking at your numbers goes, there are several ways to do it. If you do want to look at the values stored in a matrix living on the HPC, you can use the command "ppfront", like this:

```
>> b = rand(5, 5*p)

b =

        ddense object: 5-by-5p
>> c = ppfront(b)

c =

    0.4095    0.7527    0.9047    0.3043    0.4805
    0.0105    0.6617    0.8996    0.6245    0.1186
    0.0544    0.9303    0.8478    0.5921    0.3982
    0.3145    0.5157    0.3598    0.2731    0.6436
    0.4694    0.4140    0.8591    0.5877    0.7923
```

As you can see, ppfront will return the values of the matrix stored on the HPC, making them available for assignment to variables living on the front-end PC. An analogous function exists to move variables on the front-end to the back-end: "ppback". Use "ppback" like this:

```
>> d = rand(4, 4)

d =
```

```
    0.2028    0.1988    0.9318    0.5252
    0.1987    0.0153    0.4660    0.2026
    0.6038    0.7468    0.4186    0.6721
    0.2722    0.4451    0.8462    0.8381

>> e = ppback(d)

e =

        ddense object: 4-by-4p
```

Another way to examine a matrix is to examine individual scalar elements taken from the entire matrix. This works because an individual matrix element is a scalar, and scalars always live on the client. Here's an example:

```
>> a = rand(100, 100*p)

a =

        ddense object: 100-by-100p
>> a_local = a(1, 1)

a_local =

    0.7325
```

Since the matrix element “a(1, 1)” is a scalar, Star-P transfers it to the client. Assigning it to “a_local” means that you have made of copy of “a(1, 1)” locally on the client. Accordingly, the matrix “a” may change, but once “a_local” exists on the client, then its value won't be affected by changes of “a” on the server. Here's an example:

```
 >> a = rand(100, 100*p)

a =

        ddense object: 100-by-100p
>> a_local = a(1, 1)

a_local =

    0.3228

>> a = rand(100, 100*p)

a =

        ddense object: 100-by-100p
>> a(1, 1)
```

```
ans =

    0.7737

>> a_local

a_local =

    0.3228
```

The important point to remember here is that once a Star-P scalar variable is created from a server variable, it has a separate life from the server variable. That is, the scalar is a one-time copy of the variable, not a pointer to it. Moreover, to create the local copy, Star-P must communicate the scalar's value from the server to the client. This can impact your program's performance since communication takes time. Therefore, don't create scalar variables in the middle of a long calculation if you don't have to! In particular, using “for” loops to access individual elements in a large array can actually slow down your Star-P code since a client copy of each element must be made for each element! You should avoid “for” loop processing whenever possible; we recommend that you adopt a style of programming called “vectorization”. Vectorization is discussed further is the “Performance Tips” section below.

Rearview Mirror – Section summary

- Use "ppfront" and "ppback" to move large variables back and forth between the client and the server.
- Scalars always live on the client. If your calculation handles an individual matrix element, Star-P makes a local copy of the scalar.
- Avoid unnecessarily creating scalars from large matrices – creating scalars involves client/server communication, which takes time.

Pitstop – Some Parallel Computing Concepts

Let's pull in for a pit stop, where we can consider some concepts relevant to parallelizing computer programs. When thinking about parallel computing, we can loosely divide most algorithms used into two types: Data parallel and task parallel. These styles of computation are distinguished from each other by how often data is shared between different processor nodes. In detail:

- Data parallelism refers to performing computations involving short, identical operations on large datasets in parallel lock step, then communicating the results to other processors before the next operation. Parallel FFT (Fast Fourier Transform) algorithms are frequently cited as examples of data parallel computing, since they involve repeatedly performing a multiply-accumulate operation for each pair of numbers on a node, and then communicating the results with neighboring processors.
- Task parallelism refers to performing longer, non-communicating computations in parallel, and then gathering the results at the end for analysis. Task parallel operations could sensibly run on separate threads (or separate processes) on a serial computer. Monte Carlo simulations are regarded

as the canonical examples of task-parallel computing.

Star-P offers facilities for both data parallel and task parallel computation. As described above, creating matrix variables using the “*p” tag distributes them across the different compute nodes on your parallel back-end so that they may be operated upon using Star-P's built-in data parallel algorithms. For task parallel operations, Star-P features the "ppeval" statement, which explicitly invokes any desired function in parallel on a given dataset. We'll examine “ppeval” in the next section.

Rearview Mirror – Section summary

- Use “*p” for data parallel computation.
- Use “ppeval” for task parallel computation.

Back on the Road – ppeval for Task Parallelism

"Ppeval" is reminiscent of MATLAB's "feval", except that the function specified and all required data are sent to the back-end parallel processor for processing there. An example "ppeval" function call looks like this:

```
returnval = ppeval('functionname', scalar1, ...
                           split(mat1, 1), bcast(vec1))
```

In this example, "functionname" is the name of the function you wish to evaluate in parallel, and scalar1, mat1, and vec1 are the arguments you wish to send the function. According to "ppeval"'s argument handling rules, any scalar argument (such as scalar1) is broadcast to all compute nodes on the back end. Matrices and vectors can either be broadcast or split up into pieces which are divided amongst the different compute nodes according to your instructions. In this example, the matrix mat1 is split along its first dimension (i.e. split along rows), whereas the vector vec1 is broadcast to all nodes. Choosing the optimal way to distribute your multi-dimensional data to the back end is one of the design issues you will face when writing task-parallel programs under Star-P.

Where would you use "ppeval" in a real program? The simplest example is in "unrolling" for loops. That is, in serial computation you often see an operation performed like this:

```
for i=1:largeSampleNum
  result(i) = computation(i, extra_args);
end
```

That is, loop over some function, perform an independent calculation, and gather the results for later use. This construct is crying out for task-parallel evaluation! Using “ppeval”, Star-P allows you to parallelize this code as follows:

```
i = 1:largeSampleNum
result = ppeval('computation', split(i, 1), bcast(extra_args))
```

Clearly, the classic Monte Carlo simulation looks a lot like the above example; it's an obvious choice for parallelization using "ppeval". Indeed, there are several Monte Carlo examples available for you to browse on the Interactive Supercomputing website. Nonetheless, other types of task-parallel problems also exist, although they may not be as obvious. A common one are the so-called "reduction operations", such as sum, max, min, and and so on. These are associative operations which can be evaluated using some sort of hierarchy, such as a tree . The idea is to let each individual processor on the back-end evaluate a portion of the problem, return the sub-results to the local client, and then compute the final result from the partial results. Here's a simple example program in which we find the maximum value element in a large matrix:

```
% mymax.m -- use reduction to find max of large matrix
A = 1.23456789*rand(1000, 1000);

partialresults = ppeval('submax', split(A, 1)) % distribute cols
finalresult = max(partialresults)
fprintf('Largest element = %15.12e\n', finalresult)
```

Mymax.m calls this sub-function:

```
% submax.m -- fcn handles a portion of the max operation
function partialresult = submax(vect)
  partialresult = max(vect)
  return
```

Here's what happens when you run “mymax”:

```
>> mymax

partialresults =

        ddense object: 1-by-1000p

finalresult =

    1.2346

Largest element = 1.234566257663e+00
```

We multiply “A” by 1.234546789 to verify that the returned maximum random value is sensible – that is, it is close to, but not equal to, one.

To help understand what this code is doing, the algorithm used is shown pictorially in Figure 3. “Ppeval” seamlessly splits the matrix “A” into columns, sends one column to each compute node on the back-end supercomputer, initiates the execution of the function “submax” on the back-end, gathers the results from the different compute nodes, puts them in a vector, and returns the vector to the client PC for final evaluation.

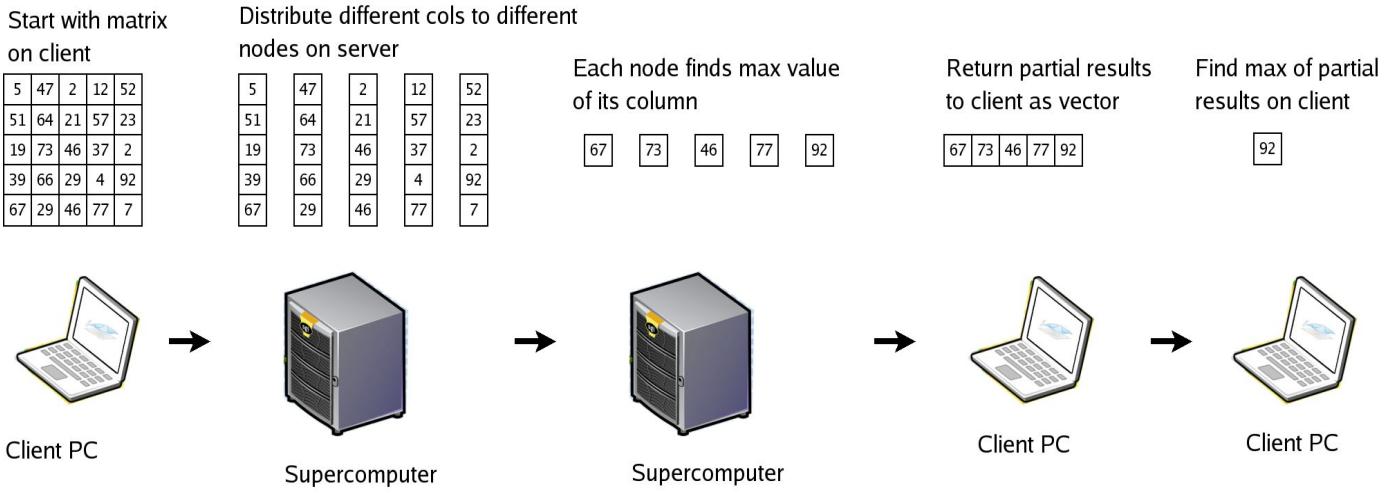

***Figure 3**: Finding the maximum value of a matrix: a simple reduction operation performed using "ppeval". Star-P's "ppeval" function takes care of transporting the data between the client and the server, making its operation completely seamless from the user's standpoint.*

Finally, if you have a legacy chunk of code you need to port, you might ask "which computational style do I use -- data parallel with "*p", or task parallel with "ppeval"? Providing this question with a complete and thorough answer is very complicated, and beyond the scope of our test drive. However, a zeroth order approximation to the answer is this: If your application crunches huge matrices using ordinary matrix operators, then data parallel computation (using "*p") is likely your preferred solution. On the other hand, if your application performs long, identical computations on unrelated datasets, then task-parallel computation (using "ppeval") is probably your best choice.

Rearview Mirror – Section summary

- Use "ppeval" for task parallel computation. Classic task parallel algorithms include Monte Carlo simulations and reduction operations.
- You can also use "ppeval" to unroll "for" loops.
- Choose between data- and task-parallelism based upon the structure of the problem you wish to solve.

Shift Into Overdrive – Further Examples

Now that we're feeling comfortable behind the wheel of Star-P, let's try some other examples. The idea here is to try out some of the usual things you would do in MATLAB, just to see how Star-P performs. Follow along as we shift up, push the pedal, and send the tachometer into the red zone!

First create a matrix "a" to play with.

```
>> % Create large matrix
>> a = rand(1000, 1000*p)

a =

        ddense object: 1000-by-1000p
>> % How big is it?
>> size(a)

ans =

     1000     1000p
```

Now let's create another matrix to play with: “a”'s inverse “c”.

```
>> % Now create its inverse
>> c = inv(a)

c =

        ddense object: 1000p-by-1000p
```

OK. Now what happens if I multiply them? Since they are inverses, the result should be the identity matrix.

```
>> % Hmmmm, what happens if I multiply them?
>> sum(sum(a*c))

ans =

   1.0000e+03

>> % What about the other way around?
>> sum(sum(c*a))

ans =

   1.0000e+03
```

That looks fine, since the identity matrix of a 1000x1000 matrix has 1000 ones on the diagonal, and zero everywhere else. How else can we check it?

```
>> % How about the norm of the product?
>> norm(a*c)

ans =

    1.0000

>> % And trace?
```

```
>> trace(a*c)

ans =

   1.0000e+03
```

That looks fine. The norm should be unity since “a” and “c” are inverses, and the trace is again the sum of 1000 ones on the diagonal. But how can we be sure that the product is really the identity matrix?

```
>> % How about a direct check of the numerical accuracy?
>> d1 = norm(c*a - eye(size(a)))

d1 =

   1.3468e-10

>> d2 = norm(a*c - eye(size(a)))

d2 =

   2.2137e-11
```

So everything looks good – we have shown that “a” and “c” are inverses of each other to within some round-off error. Keep in mind that these matrices have one million elements, so the normalized round-off error is on the order of 1e-17.

More importantly, you can see that the usual MATLAB matrix operations (eye, norm, trace) are fully supported under Star-P. This is a key feature of Star-P: it looks exactly like MATLAB! Therefore, parallelizing your MATLAB code can be as simple as just using the “*p” tag to initialize your variables.

Let's try something different – taking the Fourier Transform of a vector. To make our lives easy, let's take the Fourier Transform of a sinusoidal signal. We all know what the spectrum of a sinusoid is, after all – it's a delta function located at the base frequency! First we'll set up some variables used in the calculation.

```
>> % -----  Set up problem: create variables.
>> delta_time = 0.01;        % sample period
>> f_sample = 1/delta_time;  % sampling frequency
>> end_time = 1000;          % arbitrary end point for signal
>> numpoints = end_time/delta_time;
```

Next we'll create a time and a frequency axis for later use. These are normalized so that the units of our variables are sec and Hz.

```
>> % -----  Create time and frequency axis
>> time_vec = linspace(0, end_time, numpoints*p)

time_vec =

        ddense object: 1-by-100000p
```

```
>> f_vec = f_sample*linspace(0, 1, (numpoints)*p)

f_vec =

        ddense object: 1-by-100000p
```

Now create the sinusoidal signal itself. We'll arbitrarily give it a frequency of 5.67Hz.

```
>> % -----  Create sine wave signal to process
>> f = 5.67;              % Sine wave frequency
>> signal_vec = sin(2*pi*f*time_vec)

signal_vec =

        ddense object: 1-by-100000p
```

(Notice that pi is still 3.1415926535... in Star-P, and that multiplying a server vector with a bunch of scalars and then running it through the "sin" function returns another server vector as it should!) Now the fun part: create the signal's spectrum by taking its Fourier Transform:

```
>> % -----  Now create signal spectrum via fft
>> signal_spec = abs(fft(signal_vec))

signal_spec =

        ddense object: 1-by-100000p
```

(We took the "abs" of the vector so as not to deal with complex numbers.) OK, so what does the Fourier Transformed vector look like? Let's plot it.

```
>> plot(f_vec, signal_spec)
```

The result is shown in Figure 4.

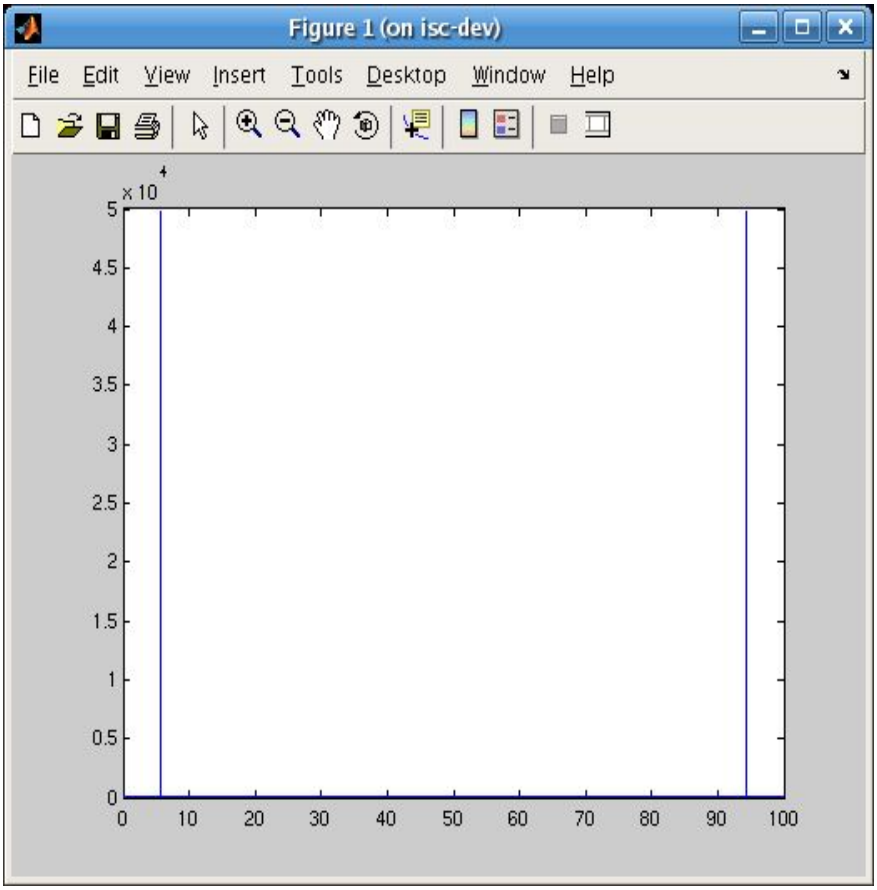

***Figure 4**: Fourier Transform of signal_vec showing two peaks in frequency. The lower one looks like it might be close to 5.67Hz, as desired.*

The resulting spectrum has two peaks. Why? Recall that the FFT returns a vector whose frequency axis wraps from positive to negative frequencies exactly ½-way through the vector. Therefore, the second peak (around 95Hz) must be the negative frequency mirror of the positive frequency peak at 5Hz. That is, the peak at the top end of the frequency range is just the mirror image of the peak at the bottom of the range. And we observe that the spike at the bottom of the range is around 5Hz, which is what we expect. So far, so good!

But how to prove that the peak is at 5Hz? The easiest way is to note that the Fourier Transform result has two peaks, and the rest of the spectrum is very small. Therefore, we can sort the signal_spec vector, identify the indices of the largest components, and then figure out what frequency components they correspond to. Here's how we do it:

```
>> % -----  Find top results
>> result(100000)

ans =

   4.9736e+04

>> result(99999)
```

```
ans =

   4.9736e+04

>> result(99998)

ans =

   2.9898e+03
```

OK, so the top two results correspond to the two peaks. They are the two largest values. The next result after that (i.e. result(99998)) is much smaller than the first two. Therefore, we can be confident that we have found the two peaks. And what frequency are the two peaks? First we need to get the indicies of the peaks:

```
>> % ----- Get indicies of peak values
>> npeak1 = indicies(100000)

npeak1 =

       94331

>> npeak2 = indicies(99999)

npeak2 =

        5671
```

Now we can see what frequencies these correspond to.

```
>> % -----  What frequencies are these?
>> fpeak1 = f_vec(npeak1)

fpeak1 =

   94.3309

>> fpeak2 = f_vec(npeak2)

fpeak2 =

    5.6701
```

So there it is! We have identified the peak at 5.67Hz which we deliberately placed in the spectrum by Fourier Transforming a sine wave with frequency of 5.67Hz.

There are two things to note about these two examples: 1. An interactive Star-P session behaves exactly like an interactive MATLAB session. 2. In both examples, we played with fairly large objects. Specifically, in the first example, our matrix was 1000x1000, meaning that it held one million double

precision values. Although this is by no means a gigantic matrix by supercomputer standards, it is fairly large for a matrix you might encounter in an everyday desktop MATLAB session. This illustrates the key ability of Star-P to enable you to handle supercomputer-sized objects while working on your desktop!

Keep in mind also that Star-P has been designed so that you can perform any of the standard MATLAB functions on back-end server variables which you can in plan-vanilla MATLAB[2]. That means that you can easily deploy your legacy MATLAB code on a parallel supercomputer, usually by simply inserting the *p tag into your code at the points where you create your matrices.

Let's do one final example. This time, we'll look at using "ppeval" for numerical integration. As you probably know, MATLAB provides a function called "quad" which you can use to numerically evaluate definite integrals over an interval. Let's look at a method useful for computing the integral of a known function over an extended range. As a simple example, consider finding a numerical value for Pi by integrating the area inside of a circle of unit radius. The idea is presented in Figure 5. A unit circle is defined by the equation x^2 + y^2 = 1. We can find the area of the unit circle by integrating the area under the curve defined by sqrt(1 – x^2) (between -1 and +1) and multiplying by 2 (to get both top and bottom halves of the circle). In ordinary MATLAB, you can express this integral in the following way:

```
>> 2*quad('sqrt(1-x.^2)', -1, 1)

ans =

    3.1416
```

But what if you had a much more complicated function, or wanted to integrate over a much larger range? In this case, the quad function might take a long time to return. How can you speed it up? Since summation is associative, you can split the total integral into sub-integrals, and perform each sub-integral on a separate processor of the parallel HPC! The idea is illustrated in the figure.

2 Consult ISC for more information about functions available in the various MATLAB add-on toolboxes since the number of supported toolbox functions grows from release to release.

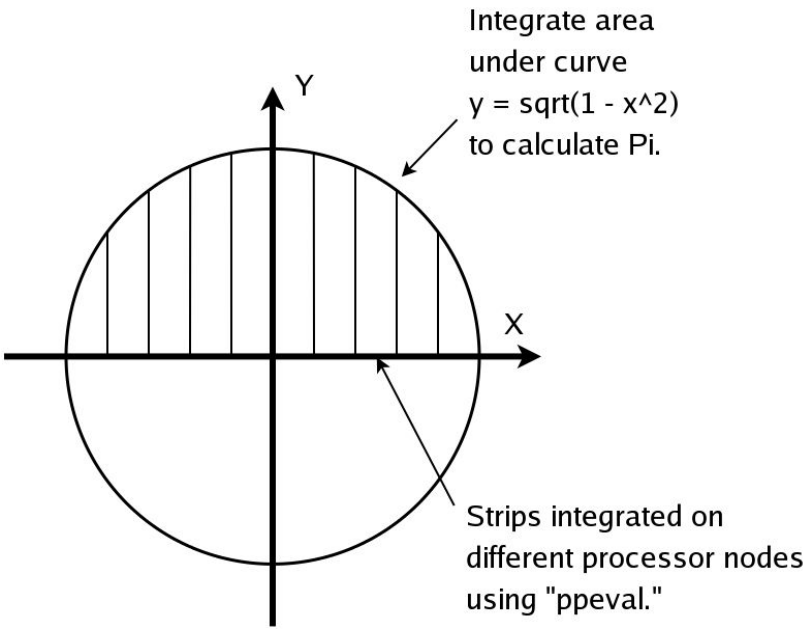

Figure 5: *Calculate Pi by integrating under curve sqrt(1-x^2). Different strips can be integrated on different compute nodes in parallel for maximum performance.*

Here's the MATLAB/Star-P code to perform the integration:

```
>> % Calculate pi by integration.  Different strips of integral
>> % are sent to different processors.
>> numstrips = 8;

>> i = 1:numstrips;
>> x = linspace(-1, 1, numstrips);   % Create x axis.
>> x1 = x(1:numstrips-1);            % strip min
>> x2 = x(2:numstrips);              % strip max
>> partial_sums = ppeval( 'quad', 'sqrt(1 - x.^2)', ...
                            split(x1), split(x2) )

partial_sums =

        ddense object: 1-by-7p
>> result = 2*sum(partial_sums)

result =

    3.1416
```

Although this example is trivial, it illustrates how to spread a function over multiple processors – a technique which can come in very handy when you wish to evaluate your own, more complex functions. Also, note that you can scale the number of strips up as much as you wish. Therefore, “ppeval” provides a way for you to increase the granularity of integration, a method which can again become very useful in more complex problems.

Rearview Mirror – Section summary

- All the usual MATLAB matrix functions are supported in Star-P.
- Star-P allows you to interact with large matrices exactly as you do with smaller ones in MATLAB.
- Use “ppeval” to spread associative functions (like integration) over multiple processors to speed up their evaluation.

Fill 'er up! Input/Output in Star-P

So far in our test drive, we've created our variables inside of a MATLAB session. However, just as you can't run a car without occasionally gassing up, so most numerical work relies upon processing external data, usually read off of disk. Therefore, you may be wondering "How do I read my data into Star-P?" Like in MATLAB, there several ways to do IO under Star-P. Each method's usefulness depends upon your specific goals for the data.

- **Read/write via MATLAB**. If your datasets are small enough, the simplest thing to do is simply read/write them into your client's MATLAB session using your favorite method, such as load/save, fread/fwrite, dlmread/dlmwrite, and so on. Then, use ppback/ppfront to move the data from the client to the server for further processing. This method is recommended for small datasets since moving data from the client to the server incurrs a time penalty. Communication takes time!

- **Read/write server variables using ppload/ppsave**. Once you have already created your data on the server, you can write out specific variables in your workspace using "ppsave". This command saves your server variables to the disk attached to the server, similar to MATLAB's "save" command. Then, you can load these variables into another Star-P session using ppload, like this:

```
>> A = rand(100, 100*p)

A =

      ddense object: 100-by-100p
>> ppsave('testsave.mat', 'A')

% ... later, in different session ...

>> ppload('testsave.mat') % loads workspace with A.
>> A

A =

        ddense object: 100-by-100p
```

  As is evident, "ppload/ppsave" use similar syntax as their MATLAB counterparts "load/save".

  You can also use ppload/ppsave to share variables between a plain-vanilla MATLAB session and a Star-P session. This is because "ppsave" stores your variables in MATLAB's uncompressed Level

5 Mat-File format, version 6. Using version 6 means that you should save variables out of MATLAB using the -v6 flag, like this:

```
save('foo','bar','-v6')
```

In the above example, MATLAB will save a file named "foo" with the variable "bar" in it. Once you have created the file, you will need to move it manually between the client and the server (e.g. using ftp or rcp) if you want to read the file from within a Star-P session.

- **Read/write HDF5 files on the server**. If you have particularly large variables living on the server, Star-P provides the option of reading and writing the HDF5 file format developed by researchers at the National Center for Supercomputing Applications at the University of Illinois. HDF5 is a standardized format tailored for the types of data frequently encountered by scientists. It is very flexible in that it can carry metadata about your data, and allows it to be grouped in various ways. HDF5 also allows for compression, if your data needs it. You can read more about the HDF5 file format on the Interactive Supercomputing website, or at its home page: http://hdf.ncsa.uiuc.edu/HDF5/

  To read/write HDF5 data within Star-P, use the commands "pph5read/pph5write". The syntax of these commands is similar to -- but not the same as -- "load/save". With "pph5read/pph5write" you first specify the name of the file holding the data, as usual. But then you specify the DATASET tags of the variables to be read. Without going into great detail, the DATASET tag is the name of the matrix which is stored alongside the data inside the HDF5 file. The DATASET tag may be different from the name of the data you use in MATLAB. Here's an example read:

```
>> [matrix1 matrix2] = pph5read
('/home/demo/sdb/FileIO/MoreMatricies.hdf5', ...
                                        'BigMat1', 'BigMat2')
matrix1 =

        ddense object: 2-by-3p

matrix2 =

        ddense object: 3-by-4p
```

  In the above case, the DATASET tags are "BigMat1" and "BigMat2"; when we read them in using "pph5read", we assign the result to matrices "matrix1" and "matrix2".

  Notice that we have used an absolute path to the filename we wish to read in the above example. By default, Star-P will try to read the file out of your $HOME directory on the server. Therefore, if the HDF5 file lives anywhere other than your $HOME directory, use a fully qualified path to your file.

  To write out an HDF5 file, use "pph5write". With "pph5write", you first specify the file's path, then the MATLAB name of the variable to write, and then the HDF5 DATASET tag to insert into

the file. As in reading HDF5 files, you can write multiple variables out at once -- just append the additional variable name/DATASET tag pair to the arguments of "pph5write", like this:

```
>>  pph5write('/home/demo/sdb/FileIO/MoreMatricies.hdf5', ...
                    matrix1, 'BigMat1', matrix2, 'BigMat2')
```

In this case, we are storing two matrices living on the server into the file with path "/home/demo/sdb/FileIO/MoreMatricies.hdf5". The variables being saved are "matrix1" and "matrix2"; they are stored under the DATASET tag "BigMat1" and "BigMat2".

Finally, if you have an HDF5 file in your local directory, but you don't know the DATASET tags of the variables in the file, you can use the command "pph5whos", which will return the names of all DATASETs stored in the file. Here's an example:

```
>> pph5whos('/home/demo/sdb/FileIO/BigMatrix.hdf5')
  Name               Size       Bytes      Class

  /myBigMatrix    200x90      144000      double array
```

Finally, note that "pph5whos" takes only one argument -- the name of the file you wish to examine.

Keep in mind that Star-P IO functions like "ppread/ppwrite" and "pph5read/pph5write" place their target files on the server machine. To verify that the files are there, or to view them, you can leave your Star-P session, log into the server, and use typical unix filesystem commands like "cd", "ls", and so on. Alternately, from within your MATLAB/StarP session you can issue the following filesystem commands to navigate around and inspect your files:

- "ppls"
- "ppcd"
- "pppwd"

These commands are similar to their MATLAB (and unix) counterparts; we won't go further into them here.

Rearview Mirror – Section summary

- For small files, use MATLAB's “load/save”, “dlmread/dlmwrite”, or your favorite functions to read the data into the client. Then use “ppback/ppfront” to move the data to/from the server.
- To read and write server variables from of your Star-P workspace, use “ppload/ppsave”.
- To read and write gigantic HDF5 files living on the server, use “pph5read/pph5write”.
- Use Star-P filesystem commands like “ppls”, “ppcd” and so on to navigate around your directories living on the server.

A Peek Under the Hood – How Star-P Works

Now that you've seen how Star-P can help you parallelize both data-parallel and task-parallel MATLAB algorithms, it's useful to delve a little bit deeper into how Star-P actually works. Knowing a little bit about Star-P's internals can help when you craft your own code to run under Star-P.

First off, let's look at Star-P's software architecture. Star-P includes software components running on both the client and the server. A simplified block diagram of the software architecture is shown in Figure 6. On your PC, the Star-P client interfaces with your MATLAB session, recognizes when you wish to perform a computation on the server, and then oversees that operation on the HPC. Special Star-P server code interfaces to MPI to manage the parallelization of your calculations. The actual calculations are performed by commonly available numerical libraries installed on each node of the supercomputer.

Importantly, you only need one MATLAB license -- for the MATLAB installation on the client. You don't need to have a MATLAB license for code running on the server since Star-P uses dedicated numerical libraries to perform your server-based calculations. This is an important cost-saving feature of Star-P for people using large cluster systems!

Other components of the Star-P system include an administration server, as well as logging and performance monitoring facilities. For simplicity we won't discuss them here; please consult the documentation available on the ISC website for complete coverage of those topics.

So how does Star-P know about whether to perform operations on the client or the server? As you already know, this depends upon where the variables involved live. However, you may be wondering about the mechanism Star-P uses to know this. In detail, Star-P works by creating a new set of variable types corresponding to matrices and arrays living on the server. Then, Star-P overloads[3] all of MATLAB's functions and operators with its own code able to handle these new variable types. In particular, the new variable types are "ddense", "dsparse", and "ddensend", for "distributed dense", "distributed sparse", and "distributed dense N-dimensional", respectively. When you create a new matrix using the "*p" tag, the matrix created will have one of these distributed types, and the matrix data is accordingly distributed across the various processors of the HPC server. Then, when you operate upon the matrix using one of MATLAB's many operators or functions, MATLAB recognizes that the calling argument is one of these new data types, and consequently passes control to the Star-P code which explicitly performs the matrix operations on the HPC server instead of on the client.

Since the new data type for matrices and arrays is what triggers parallel execution, the semantics of Star-P have been carefully designed to propagate the distributed data type as frequently as possible. To determine if the return value of a function is distributed or not, the following rules are obeyed:

- Operators whose returns are bigger than the size of the input (such as “kron” and “meshgrid”) will return distributed objects if any of their inputs are distributed.

3 "Overloading" is a concept from object-oriented software development. It refers to the ability of some programming languages (such as C++) to extend the definition of operators (such as “+”, “==”, and named functions) to handle new data types. This ability is obtained when operators are allowed to have different implementations depending on the types of their arguments.

- Operators which return data objects the same size as their input (such as +, *, “cumsum”, “fft”, and so on) will return distributed objects if any of their input arguments are distributed.
- Operators which reduce the dimensionality of their input (such as “max” or “sum”), will return distributed objects if any of the calling arguments is distributed, and the returned object is larger than a scalar (1x1 matrix).
- Operators which return a fixed number of values, independent of the size of the input (such as “eigs”, “svd”, and “histc”) will return local MATLAB (non-distributed) objects even if one or more of the calling arguments are distributed.
- Operators which return a single scalar value will always return a local scalar. This occurs not only as a corollary of the above rule, but is also a consequence of the fact that scalars always live on the local client under Star-P.

But what about the "*p" tag? What does it actually do to enable creation of the new data types? The answer is that "p" itself is a function with a return value of unity (i.e. 1), but of variable type "dlayout". Therefore, when you create an array by multiplying one of the dimensions by "*p", you are actually changing the variable type of that dimension to "dlayout", but leaving its value the same! Here's an example showing how "*p" modifies any value it touches:

```
>> a = 10*p

a =
     10p
>> whos

  Name      Size                     Bytes  Class

  a         1x1                        258  dlayout object
Grand total is 4 elements using 258 bytes
```

Now when MATLAB tries to create the array, MATLAB sees the "dlayout" type in the array's dimension, and passes control to the Star-P code which handles the dlayout data type. This is the Star-P code which can perform distributed object creation. Once Star-P is in control, it creates the desired array on the server, and gives the new array a data type of ddense, dsparse, or ddensend depending upon the particulars of the object's creation call.

Since the only function of the dlayout class is to declare the dimensions of objects to be distributed, you won't often see it returned when you create variables. In general, operators return dlayout types only when they involve array construction, or are simple operators often used in calculations on array bounds (for example, max, floor, abs).

By the way, since "p" is just a MATLAB function defined by Star-P, should you ever want to stop performing your calculations on the server and simply run a normal MATLAB session, you can set p = 1 at the beginning of your program. That will override Star-P's redefinition of "p"; in particular, multiplication by "p" (i.e. the "*p" tag) will no longer return a "dlayout" type, but will return an integer type instead. In turn, this means that normal MATLAB code will be called to create your matrices, and the matrices themselves will live on the local client PC.

Finally, once the data is created on the back end server, Star-P incorporates code which initiates and

manages the parallel calculation (via MPI), can invoke functions from any of a number of numerical

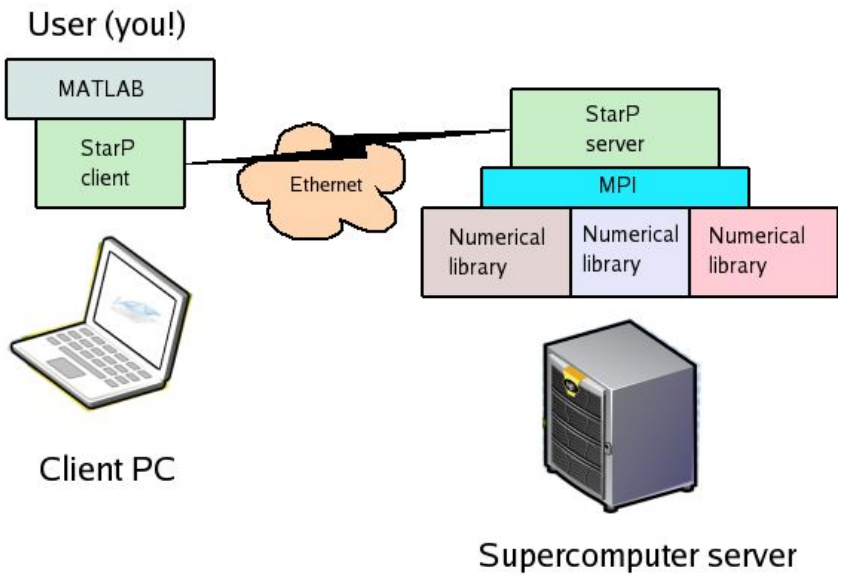

***Figure 6**: Simplified software architecture of ISC's Star-P system. Star-P includes both client and server components which seamlessly exchange commands and data between the user's MATLAB session on the client with parallel number crunching libraries on the server.*

libraries to operate on the distributed data, and knows how to return the calculation's results to your local MATLAB session. In other words, Star-P handles the complexities of running MPI code so that you can concentrate on the details of your calculation!

Rearview Mirror – Section summary

- Distributed variables living on the server have one of the following new types, defined by Star-P: “ddense”, “dsparse”, or “ddensend”.
- Star-P performs calculations on distributed variables since all MATLAB functions are overloaded with code knowing how to handle distributed variables.
- The “*p” tag applied to an array dimension creates a “dlayout” type, which then invokes Star-P code to create the associated array on the server.

Race for the Finish Line – Tips for Maximum Star-P Performance

At this point you may be wondering "Is that all there is to Star-P? Is it really that easy?" The answer is: Yes! Writing parallel code under Star-P typically involves using the “*p” tag on your variables for data parallel calculations, and using "ppeval" for task parallel computation. If you judiciously use these unique Star-P facilities in your code, you can usually improve the performance of your MATLAB programs in one or more of the following ways:

- First, if you have an algorithm which lends itself readily to parallelization, then Star-P can speed up

your MATLAB code simply by parallelizing it. Simply put, the more processors working simultaneously on your calculation, the faster it will complete!

- Second, if you have gigantic matrices which exceed the capacity of your local PC, then exporting your matrices to an HPC with a huge memory space means that you can now run your program on datasets which might have been previously impossible.
- Similarly, the Windows version of MATLAB currently remains a 32 bit application. Support for 64 bit applications under Windows is spotty (as of this writing). However, Star-P -- like most supercomputer applications -- is 64 bit native. Therefore, using Star-P as a backend for your Windows-based MATLAB provides you the advantages of a 64 bit application.

However, the greatest advantage to using Star-P is that you can gain the above performance improvements **without** needing to recode your MATLAB program into a new language, or learn MPI. That means that you can parallelize your existing application **right now**, with **no risk** and with **no downtime!**

That being said, obtaining the maximum performance from the Star-P platform requires that you keep a couple of points in mind as you prepare your MATLAB code to run under Star-P. First, under Star-P some of your variables (typically, your matrices) are kept on the back end server, while others live on your local client. Star-P keeps track of which variables live where, and enables seamless computation using all of your variables within your MATLAB session by sending data back and forth over the network connection when required. However, since client/server communication requires time, moving variables back and forth implies a time penalty. Most numerical calculations don't require lots of client/server communication -- once you've created your variables on the server, they stay there until the end of your calculation. However, in certain circumstances the user can trigger communication of extremely large matrices between the PC client and the HPC server. Naturally, in this case your calculation's execution time becomes limited by the transmission time of your data -- a situation you likely wish to avoid. Worse, if you happen to move a matrix from the server to the client, and it is too large to fit into your client machine's memory, then you can crash your client PC! Note that simple awareness of what data lives where is all you need in order to avoid these situations. Our catch phrase for this is "follow the data!" Avoid unnecessary client/server communication by understanding where your variables live.

Second, remember that Star-P works on your large matrices by overloading MATLAB's functions and operators to perform computations on the HPC server when they encounter a variable of type "ddense" or "dsparse". This means that for each MATLAB function, Star-P incorporates highly optimized algorithms to operate on non-scalar data directly on the HPC server. On the other hand, operations on scalars are performed on the client PC. This means that if you use "for" loops to operate on individual components of a matrix living on the server, a scalar value must be transferred from the server to the client. Consequently, you necessarily incur a communication penalty for each matrix element you operate on! Therefore, always use Star-P's built-in functions to operate on your matrices.

This practice is called "vectorization"; it means that you avoid "for" loops and use Star-P's built-in functions to create and manipulate your vectors and matrices all at once. Here's a trivial example:

```
% Non-vectorized
for j=1:1000;
  x(j) = sin(2*pi*j/1000);
end;
```

```
% Vectorized
x = sin(2*pi*[1:1000]/1000);
```

Interactive Supercomputing's "support" website holds several applications notes discussing vectorization in detail, and describes several clever ways to vectorize your code. We recommend you to spend some time perusing the materials and examples available there. However, the main point is this: both MATLAB and Star-P provide syntax allowing you to operate on entire matrices at once -- use it to obtain the best performance from your MATLAB/Star-P code!

Rearview Mirror – Section summary

- Minimize the amount of client/server communication required during your calculation. "Follow the data!" -- know where your variables live and don't move large variables between client and server unnecessarily.
- Thoroughly vectorize your code for best performance. Avoid "for" loops wherever possible!

In the Garage – Some Useful Diagnostic Tools

You're just about ready to use Star-P on your own computations! But before you sit down at your own Star-P session, here are a few utility commands which you might find useful when writing or porting code to Star-P.

- **help:** "Help" is a regular MATLAB command. We only mention it here to point out that all Star-P commands are discussed by "help"; some of the returned help strings are quite long and informative! Use "help" if you forget the syntax of any particular command.

- **ppwhos**: This command works similarly to MATLAB's "whos" command. Ppwhos returns a list of all variables living on the HPC server, along with their type. Remember the catchphrase: "Follow the data!" Use ppwhos to help you follow your variables as you process them, or if you forget where a particular variable lives. Here's an example:

```
>> A = rand(1000, 1000*p)

A =

        ddense object: 1000-by-1000p
>> B = linspace(0, 1, 10000*p)

B =

        ddense object: 1-by-10000p
>> ppwhos
Your variables are:
  Name       Size            Bytes        Class
  A          1000x1000p      8000000      ddense array
  B          1x10000p        80000        ddense array
```

```
Grand total is 1010000 elements using 8080000 bytes
MATLAB has a total of 0 elements using 0 bytes
Star-P server has a total of 1010000 elements using 8080000 bytes
```

- **pptic/pptoc**: These commands function analogously to MATLAB's tic/toc. Pptic/pptoc tell you how much time a calculation takes running on the server. They also report the time spent communicating between client and server. These commands come in handy when your code takes longer to execute than you expect, or if you want to tweak your code to obtain optimum performance.

```
>> A = rand(10000, 1000*p)

A =

        ddense object: 10000-by-1000p
>> C = rand(10000, 1000*p)

C =

        ddense object: 10000-by-1000p
>>
>>  % Now time multiplication on the server.
>> pptic, A * C', pptoc

ans =

        ddense object: 10000-by-10000p
Client/server communication info:
    Send msgs/bytes    Recv msgs/bytes     Time spent
  0e+00 / 0.000e+00B  3e+00 / 3.460e+02B   1.361e-01s
Server info:
  execution time on server: 8.250e+00s
  #ppchangedist calls: 2
```

As you can see, “pptic/pptoc” returns not only timing information. They also return statistics about communication between client and server. This information can be useful when diagnosing performance problems under Star-P. Consult the Star-P “support” website for more information about this topic.

- **ppver**: This command reports information about the version of Star-P you are running. Should you ever need to communicate with Interactive Supercomputing about a bug or other issue with Star-P, you should first run this command to find out exactly which version of Star-P you have. Then please report this information to your contact person at Interactive Supercomputing.

- **ppgetlog**: Every time you initiate a Star-P session, a session log is generated on your server which records the detailed steps taken by Star-P while running your program. In the unfortunate case where your program experiences a problem (or crashes), you can frequently use information held in this log to figure out what went wrong with your job. And if you ever involve Interactive

Supercomputing with tracking down a problem in your program, you may be asked to supply the server log of a session which evinces your bug.

Join the Winner's Circle – Final Words

Congratulations! You've crossed the finish line and are now ready to use Star-P for your own problems. As you see, Star-P makes it easy to parallelize your MATLAB applications. Just remember to initialize your data-parallel variables with the "*p" tag and use "ppeval" (or equivalent) to invoke task-parallel functions. It's that easy!

Once you start using Star-P on your applications, you will be pleased to see how quickly you can become adept at parallelizing your own code. As you explore its abilities and use its features in your own work, don't forget to check the Interactive Supercomputing website for more detailed information if you have a question or encounter a problem. Also, it's always useful to browse the ISC "support" web pages for coding tips and tricks relevant to your application. With just a little bit of reading and experimenting, Star-P will allow you to quickly harness the power of a parallel supercomputer on your MATLAB applications, without having to recode your program in a new language or learn MPI!

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.