Star-P® Programming Guide for Use with MATLAB®

Release 2.7

12/11/08

Release 2.7 - 12/11/08

COPYRIGHT

Copyright © 2004-2008, Interactive Supercomputing, Inc. All rights reserved. Portions Copyright © 2003-2004 Massachusetts Institute of Technology. All rights reserved.

Trademark Usage Notice

Star-P® and the "star" logo are registered trademarks of Interactive Supercomputing, Inc. MATLAB® is a registered trademark of The MathWorks, Inc. Other product or brand names are trademarks or registered trademarks of their respective holders. ISC's products are not sponsored or endorsed by The MathWorks, Inc. or by any other trademark owner referred to in this document.

INTERACTIVE supercomputing

Contents

Star-P® Introduction

Extending MATLAB with Star-P® 2
Parallel Computing Basics 4
About the Star-P® Programming Guide for Use with MATLAB® 5

Starting Star-P® with MATLAB

Getting Help at the IDE Window 7
Using the HTML-Based Help 8
Using the Text-Based Help 8
Getting Command Syntax Information 8
Syntax grammar and conventions used in the Star-P® documentation 9
Get syntax information for a particular function 9
Starting Star-P® on a Linux Client System. 9
Starting Star-P® on a Windows Client System. 10
Star-P® Dashboard 11
Star-P® Sample Session 13
User Specific Star-P® Start-Up Configuration 14
Star-P® Start-Up Command Line Options 15
Launching Star-P® with a MATLAB .m script 19
Cluster Configurations 19

Data Parallelism with Star-P® and MATLAB

Star-P® Naming Conventions 22
Examining Star-P® Data 22
Reusing Existing Scripts 23
Examining/Changing Distributed Matrices 24
Special Variables: p and np 24
Assignments to p 26
Supported Data Types 27
Real and Complex Data 27
Creating Distributed Arrays 28
The *p Syntax. 28
Distributed Data Creation Routines 29
Distributed Array Bounds 30
Indexing into Distributed Matrices or Arrays 30
Types of Distributions 32
Distributed Dense Matrices and Arrays 32
Row distribution 33
Column distribution. 33
Distributed Dense Multidimensional Arrays 34
Distributed Sparse Matrices 34
How Star-P® Represents Sparse Matrices 35

Distributed Cell Objects (dcell) . . . 36
Combining Data Distribution Mechanisms . . . 36
Mixing Local and Distributed Data . . . 37
Distributed Classes used by Star-P® . . . 38
Propagating the Distributed Attribute . . . 41
Propagation of Distribution . . . 44
Functions of One Argument . . . 44
Functions of Multiple Arguments . . . 45
Indexing Operations . . . 48
Summary for Propagation of Distribution . . . 49
Explicit Data Movement with ppback and ppfront . . . 49
Loading And Saving Data on the Parallel Server . . . 51
HDF5, Hierarchical Data Format Version 5 . . . 53
Representation of data in the HDF5 file . . . 55
Limitations . . . 56
Differences from MATLAB HDF5 support . . . 56
Converting data from other formats to HDF5 . . . 56

Task Parallelism with Star-P® and MATLAB

The ppeval Function: The Mechanism for Task Parallelism . . . 59
Star-P® Naming Conventions . . . 60
Transforming a for Loop into a ppeval Call . . . 61
Step 1: Identify a for loop that is embarrassingly parallel. . . . 61
Step 2: Determine the input and output variable of the loop . . . 61
Step 3: Transform the body of the for loop into a function . . . 62
Step 4: Call function defined in Step 3 with ppeval . . . 62
ppeval Syntax and Behavior . . . 63
ppeval Syntax Grammar . . . 63
Requirements of Functions Passed to ppeval . . . 64
Input Arguments . . . 64
Default Behavior . . . 64
Splitting . . . 64
Broadcasting . . . 65
Supported Input Argument Types . . . 65
Serial ppeval of Functions with Scalar Inputs . . . 65
Client vs. Server Variables . . . 66
Distribution of input variables . . . 67
Output Arguments . . . 67
Examples of ppeval Usage . . . 68
ppevalsplit . . . 69
Choosing Your Task Parallel Engine (TPE) . . . 70
Star-P® M TPE . . . 71
Star-P® Octave Engine . . . 71
C/C++ Engine for Running Compiled C/C++ Package Functions . . . 72
Per Process Execution . . . 72
Calling Non-”M” Functions from within ppeval . . . 73
Workarounds and Additional Information . . . 75
String Arrays . . . 75
Splitting on a Scalar . . . 75

Global Variables . . . 75

Tips and Tools for High Performance Star-P® Code
Performance and Productivity . . . 77
Tips for Data Parallel Code . . . 79
Vectorization . . . 79
Star-P® Solves the Breakdown of Serial Vectorization . . . 82
Solving Large Problems: Memory Issues . . . 84
Tips for Task Parallel Code . . . 85
Use of Structs and Cell Arrays . . . 85
Vectorize for Loops Inside of ppeval Calls . . . 86
Performance Note on Iteration Timing . . . 87
Using External Libraries . . . 89
Integer Arithmetic in Star-P® Compared with MATLAB® . . . 89
Accuracy of Star-P® Routines . . . 89
Configuring ppsetoption for High Performance . . . 90
Performance Tuning and Monitoring . . . 91
Diagnostics and Performance . . . 91
Client/Server Performance Monitoring . . . 91
Coarse Timing with pptic and pptoc . . . 91
Summary and Per-Server-Call Timings with ppprofile . . . 94
Maximizing Performance . . . 98
Maintaining Awareness of Communication Dependencies . . . 98
Communication between the Star-P® Client and Server . . . 98
Implicit Communication . . . 99
Communication Among the Processors in the Parallel Server . . . 101
Enhanced Performance Profiling in Star-P® . . . 103
Using ppperf . . . 103
Interpretation of ppperf's output . . . 108
Using ppperf's graphical mode . . . 113
Using ppperf to Eliminate Performance Bottlenecks . . . 117
ppperf command summary . . . 131
UNIX Commands to Monitor the Server . . . 132

Star-P® Functions
Basic Server Functions Summary . . . 135
General Functions . . . 138
fseek . . . 138
np . . . 139
p . . . 139
pp . . . 139
ppbench . . . 139
ppclear . . . 140
ppgetoption . . . 141
ppsetoption . . . 141
ppgetlog . . . 141
ppgetlogpath . . . 142
ppinvoke . . . 143
pploadpackage . . . 143

ppunloadpackage . . . 144
ppfopen . . . 145
ppquit . . . 145
ppwhos . . . 145
pph5whos . . . 146
Data Movement Functions . . . 147
ppback . . . 147
ppfront . . . 148
ppchangedist . . . 148
pph5write . . . 149
pph5read . . . 150
ppload . . . 150
ppsave . . . 151
Task Parallel Functions . . . 151
bcast, ppbcast . . . 151
split, ppsplit. . . . 152
ppeval. . . . 152
ppevalsplit . . . 154
ppevalcloadmodule . . . 154
ppevalcunloadmodule . . . 154
Performance Functions . . . 155
ppperf . . . 155
ppprofile . . . 156
pptic/pptoc . . . 157

Supported MATLAB® Functions
Data Parallel Functions Listed Alphabetically . . . 161
Task-Parallel Functions Listed by Default Platform TPE . . . 170

Application Examples
Application Example: Image Processing Algorithm . . . 207
How the Analysis Is Done . . . 207
Application Examples. . . . 208
Images For Application Examples . . . 208
M Files for the Application Examples . . . 208
Application Example Not Using Star-P® . . . 209
patmatch_color_noStarP.m File . . . 210
patmatch_calc.m . . . 211
Application Example Using Star-P® . . . 212
patmatch_colordemo_StarP.m File . . . 212
Application Example Using ppeval . . . 213
About ppeval . . . 213
About the ppeval Example . . . 213
patmatch_color_ppeval.m . . . 214

Solving Large Sparse Matrix and Combinatorial Problems with Star-P®
Graphs and Sparse Matrices . . . 217
Graphs: It's all in the connections . . . 217
Sparse Matrices: Representing Graphs and General Data Analysis . . . 219

Data Analysis and Comparison with Pivot Tables 220
Laplacian Matrices and Visualizing Graphs 223
On Path Counting 224

INTERACTIVE supercomputing

Chapter 1

Star-P® Introduction

Star-P® drives productivity by significantly increasing application performance while keeping development costs low. It is intended for scientists, engineers and analysts with large and complex problems that cannot be solved productively on a desktop computer. The Star-P® software platform seamlessly integrates desktop clients with high-performance servers. By offloading computation, memory and file intensive operations to the server, Star-P® enables easy to use desktop application development, while creating the potential for execution at supercomputer speeds.

Star-P® extends easy to use Very High Level Languages (VHLLs) such as MATLAB®[1] and Python to support simple, user-friendly parallel computing on a spectrum of computing architectures: multi-core desktops and servers, large shared memory servers, and clusters. Star-P® fundamentally transforms the workflow, substantially shortening the “time to solution” by allowing the user to easily adapt their application for use on parallel resources.

This chapter provides an overview of using Star-P® in the MATLAB® VHLL environment. It includes sections on the following topics:

- “Extending MATLAB with Star-P®” describes how Star-P® parallelizes MATLAB programs with minimal modification.
- “Parallel Computing Basics” introduces you to the various domains of parallel computing and how Star-P® fits into various domains.
- “About the Star-P® Programming Guide for Use with MATLAB®” summarizes the topics covered in this document.

1. MATLAB® is a registered trademark of The MathWorks, Inc. Star-P® and the "star" logo are registered trademarks of Interactive Supercomputing, Inc. Other product or brand names are trademarks or registered trademarks of their respective holders. ISC's products are not sponsored or endorsed by The MathWorks, Inc. or by any other trademark owner referred to in this document.

Extending MATLAB with Star-P®

With Star-P®, existing MATLAB scripts and functions can be re-used to run larger problems in parallel with minimal modification, and new parallel MATLAB code can be developed in a fraction of the time normally required to develop parallel applications in traditional programming languages, such as C, C++, or Fortran with MPI. Parallel programming with Star-P® in MATLAB requires learning a bare minimum of additional programming constructs. Implementing Data Parallelism with Star-P® does not require the addition of any new functions to your MATLAB code, and adding Task Parallelism requires only one additional construct.

To implement Data Parallelism, Star-P® overloads ordinary MATLAB commands with the `*p` construct. This simply multiplies (*) array dimension(s) by a symbolic variable (`p`) denoting that a matrix dimension is to be distributed. A class of overloaded MATLAB programs becomes parallel with the insertion of this construct. The `*p` syntax tells data construction routines (for example, `rand`) to build the matrix on the parallel HPC back-end, and perform the indicated operation (for example, matrix inversion) there as well. Creating a distributed random matrix and taking its inverse with MATLAB can be expressed with the following two lines of code:

```
A = rand(100,100);
B = inv(A);
```

To express the same operations in Data Parallel using Star-P®, requires only one slight change:

```
App = rand(100,100*p);
Bpp = inv(App);
```

Once the `*p` construct has been applied to a variable, all subsequent operations on that variable will occur in parallel on the HPC and result in new variables that are also resident on the HPC. This important inheritance feature of Star-P® allows you to parallelize your MATLAB code with minimal effort. For more information on distributed data operations, see “Data Parallelism with Star-P® and MATLAB”.

For implementing Task Parallel functionality, Star-P® introduces the `ppeval` function into MATLAB. The `ppeval` function, which is called in a similar manner as the MATLAB function `feval`, allows one to pass a string containing a valid MATLAB function `foo` as well as all of `foo`’s calling arguments. The `ppeval` function then packages `foo`, along with all functions called within `foo`, and ships those functions to the HPC server. The calling arguments of `foo` are also shipped to the HPC and can be either broadcast to all processors using the `bcast` function or split amongst the processors using the `split` function.

The following code is an example usage of the `ppeval` function:

```
App = rand(100,100,100*p);
```

```
Bpp = ppeval('inv',App);
```

Or equivalently:

```
Bpp = ppeval('inv',split(App,3));
```

In this example, the `ppeval` call splits the variable **App** into 100 individual slices (along the last dimension). The slices are divided among available processors on the server, and then each processor iterates over its received slices, performing an `inv` operation on each slice. The results from all processors are then combined, preserving original order, and returned as the output variable. More information about task parallel functionality can be found in "Task Parallelism with Star-P® and MATLAB"

To use Star-P® with MATLAB, the user needs only one copy of the Mathworks' product to serve as a front-end, which need not be the parallel machine. No copies of MATLAB are required on the parallel computer.

Users have the benefit of working in the familiar MATLAB environment. When new releases of MATLAB are distributed, the user merely plugs in the new copy and Star-P® continues to execute.

Despite Star-P®'s ability to add functionality for distributed matrices and parallel operations, don't forget that you are still using MATLAB as your desktop development tool. This means that you can run an existing MATLAB program in Star-P® with almost no changes, and it will run strictly on your desktop (client) machine, never invoking the Star-P® system after initialization. Of course, this would be a waste of HPC resources, if you ran this way all the time. But it is a convenient way of porting the compute-intensive portions of your code one at a time, allowing the unported portions to execute in MATLAB proper.

In the Star-P® context, there are many features of the MATLAB environment that are still relevant for developing applications with distributed objects and operations. The MATLAB debugger and the script and function editor are two of the most useful MATLAB functions when you're programming with Star-P®. The designers of Star-P® have taken great pains to fit within the MATLAB mindset, using the approach "It's still MATLAB." So if you're wondering whether a MATLAB operation works in Star-P®, just try it. Most operations work in the obvious way.

> **Note:** If a MATLAB function that has high value for you does not work, please let us know via support@interactivesupercomputing.com

Star-P® greatly simplifies the parallelization of new and existing MATLAB code by allowing the user to either run code on the local MATLAB client or on the HPC back-end appropriately taking advantage of the respective strengths.

Parallel Computing Basics

This section reviews various domains of parallel computing. We present these concepts for users who are new to parallel computing and then discuss their implementation by Star-P®.

Parallel computing textbooks list many models for parallelizing programs, including:

- Data Parallel Computation
- Message Passing
- Task Parallel Computation

You may wish to go to a website that has several points related to parallel computing, such as, http://beowulf.csail.mit.edu/18.337 or any of the numbers of textbooks that cover these topics. In brief, the current version of Star-P® is best expressed as a data parallel language or a global array language. The prototypical example of data parallelism is matrix addition:

```
Cpp = App + Bpp;
```

where `App` and `Bpp` are matrices. When we add two n-by-n matrices, we perform n^2 data parallel additions. In other words, we perform the same operation (addition) simultaneously on each of the n^2 numbers.

The name "data parallel" is often extended to operations that have communication of dependencies among some of the operations, but at some level can be viewed as identical operations happening purely in parallel. Two simple examples are matrix multiplication (`Cpp=App*Bpp`) and prefix sums (`Dpp=cumsum(App)`).

A beneficial description of Star-P® for many users is that Star-P® is a global array syntax language. By providing a global array syntax in Star-P®, the user variable `App` refers to the entirety of a distributed object on the back end server. The abstraction of an array that contains many elements is a powerful construct. With one variable name such as `App`, you are able to package up a large collection of numbers. This construct enables higher level mathematical operations expressed with a minimal amount of notation. On a parallel computer, this construct allows you to consider data on many processors as one entity.

By contrast, message passing or "node-oriented" languages force you as a programmer to consider only local data and create any global entity completely outside the scope of the language. Data is passed around through explicit calls to routines such as `send` and `receive` or `SHMEM get` and `put`. The lack of support for the global entity places more of a cognitive burden on you, the programmer. Star-P® allows users to implement their programs in parallel without having to master the intricacies of MPI in Fortran, C, or C++.

"Task parallel" or "embarrassingly parallel" computations are those operations where there is little or no dependency among the computational pieces. Each piece can easily be placed on a distinct processor. While not strictly required, such computations typically depend on a

relatively small amount of input data, and produce relatively small amounts of output data. In such circumstances, the implementation may not store any persistent data on distributed memory. An example is Monte Carlo simulation of financial instruments, where the calculations for each sample are done completely in isolation from every other sample. While Star-P® may be considered a data parallel language, it also has task parallel functionality through the use of its `ppeval` operation.

Most of the operations for which Star-P® will deliver good performance will be operations on global arrays, so most of this document treats arrays as global arrays. An important exception to this is the `ppeval` function, which supports task parallelism and works on global arrays, but in a less straightforward manner. A global array that is an input to the `ppeval` function is partitioned into sections, each of which is converted to an array that is local to a single instance of a MATLAB function on a single processor. The reverse process is used for output arrays; the assemblage of the sections into global arrays.

About the Star-P® Programming Guide for Use with MATLAB®

The remainder of this document provides chapters that cover the following topics:

- "Starting Star-P® with MATLAB" takes you through a sample session to illustrate how to start up Star-P® from a graphical or command line interface with various command-line options. A simple program is shown that illustrates the use of Star-P®'s ability to parallelize MATLAB code.
- "Data Parallelism with Star-P® and MATLAB" describes Star-P®'s global-array language capabilities for creating, manipulating, loading and saving large distributed data.
- "Task Parallelism with Star-P® and MATLAB" describes Star-P®'s `ppeval` function for performing embarrassingly parallel operations on either local or distributed data.
- "Tips and Tools for High Performance Star-P® Code" provides suggestions for maximizing the performance of code written for both data and task parallel computations, and describes tools for monitoring and profiling MATLAB code using Star-P®.
- "Star-P® Functions" summarizes functions that are not part of the standard MATLAB language and describes their implementation.
- "Supported MATLAB® Functions" lists the MATLAB functions that are supported in both data and task parallel modes, as well as MATLAB toolbox functions that are supported only in task parallel computations.

INTER*CTIVE supercomputing

Chapter 2

Starting Star-P® with MATLAB

This chapter is intended for users who have a working Star-P® installation on a client system as well as a high performance computing server. It includes the following topics:

- "Getting Help at the IDE Window" explains how to use and invoke help.
- "Starting Star-P® on a Linux Client System" provides information for users running Star-P® under Linux.
- "Starting Star-P® on a Windows Client System" provides information for users running Star-P® under Windows
- "Star-P® Dashboard" includes information on a graphic window provided by Star-P® for monitoring server status at start-up and a means of killing a Star-P® server session.
- "Star-P® Sample Session" walks you through a few short example operations that can be performed in a working Star-P® session.
- "User Specific Star-P® Start-Up Configuration" includes information for users wishing to configure particular start-up options within a start-up script.
- "Star-P® Start-Up Command Line Options" provides information about launching a desired Star-P® session from a terminal command prompt.

Getting Help at the IDE Window

When working at the IDE, you can invoke online help in the following ways:

- Using the HTML-Based Help
- Using the Text-Based Help
- Getting Command Syntax Information

Using the HTML-Based Help

You can get Star-P® HTML-based help within the MATLAB IDE by entering the starpdoc command.

```
>> starpdoc rosser
```

You can also get information on how to use starphelp by using the MATLAB command help. For example:

```
>> help starpdoc
starpdoc Get browser-displayed help related to Star-P® parallel computing

Syntax 1:
starpdoc  % Bring up the main Star-P® online help page

Syntax 2:
starpdoc <Star-P®-M-function-name> | <Star-P®-M-library-name> | syntax
```

Using the Text-Based Help

You can get Star-P® HTML-based help within the MATLAB IDE by entering the starpdoc command.

```
>> starphelp rosser
```

You can also get information on how to use starphelp by using the MATLAB command help. For example:

```
>> help starpdoc
starpdoc Get text-display help related to Star-P® parallel computing

Syntax 1:
starphelp  % Bring up the main list of Star-P® help

Syntax 2:
starphelp <Star-P®-M-function-name> | <Star-P®-M-library-name> | syntax
```

Getting Command Syntax Information

You can get Star-P®-specific conventions and syntax information by way of the following methods:

- Syntax grammar and conventions used in the Star-P® documentation
- Get syntax information for a particular function

Syntax grammar and conventions used in the Star-P® documentation

For general syntax grammar usage and conventions, you can invoke either starphelp or starpdoc using the <syntax> option.

```
>> starphelp syntax

Syntax Grammar Conventions for Star-P® M Documentation
------------------------------------------------------

Convention...                                Meaning...
```

Get syntax information for a particular function

You can individual functions by calling either form of Star-P® help with a function name as its argument.

```
>> starphelp rosser
cross Return the cross product of two vectors

Syntax 1:
<vector-cross-product> = cross( <input-vector-1> , <input-vector-1> )
```

Starting Star-P® on a Linux Client System

Your system administrator will usually have installed the Star-P® software on the systems (client(s) and server) you will be running on in advance. The default location of the `starp` software is `/usr/local/starp/<version>`. Assuming this install location is in your shell path, then the following sequence will start the Star-P® client (on a system named `your_system`) and connect to the Star-P® server configured by the administrator, which happens to be a system named `remote_server`.

```
your_system% starp
user@remote_server's password: **********

                                    < M A T L A B >
                       Copyright 1984-2009 The MathWorks, Inc.

  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.

Connecting to Star-P® Server with 2 processes

Star-P® Version 2.7.0
(C) 2004-2008, Interactive Supercomputing, Inc. All rights reserved.
Portions (C) Copyright 2003-2004 Massachusetts Institute of Technology. All
rights reserved.
```

```
By using this software you agree to the terms and conditions described
in the license agreement.  Type help agreement
client log file: /home_directory/.starp/log/2008_04_05_1111_54/starpclient.log
 >>
```

As you can see, the HPC server will typically require a password for user authentication. You will either need to supply this password upon every start-up or configure SSH so it is not needed on every session initiation. Otherwise, there are few visible signs that the Star-P® server is running on a distinct machine from your client.

This last line (“>>”) is the MATLAB prompt. At this point you can type the commands and operators that you are familiar with using from prior MATLAB experience, and can start to use the Star-P® extensions described in “Data Parallelism with Star-P® and MATLAB” and “Task Parallelism with Star-P® and MATLAB”.

A full description of the **`starp`** command and its command line options is provided in the section “Star-P® Functions”, or by typing the following at the command prompt:

```
$ ./starp --help
```

Starting Star-P® on a Windows Client System

By default, the Star-P® installation on a Windows XP system will create a shortcut on the desktop, as well as an entry in the list of programs under the Windows Start menu.

The default location for the Star-P® programs will be `C:\Program Files\starp`; if you can’t find them there, check with your system administrator to see if an alternate location was used. For installation instructions, see the “Star-P® Installation and Configuration Guide”.

To invoke the Star-P® software, either double-click the desktop icon, or click on:

Start -> **All Programs** -> Star-P® **Client Software** -> Star-P® **M Client**

Figure 2-1 Star-P® Desktop icon

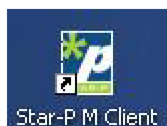


Figure 2-2 Star-P® Login screen

If passwordless SSH has not been configured for the user name in your current Star-P® properties configuration file (the default file being starpd.properties), a dialogue box will appear prompting you for a password. If no user name appears in the configuration file, then the user name associated with your current Windows session will be utilized.

Once the connection has been established, MATLAB will start, with Star-P® enabled.

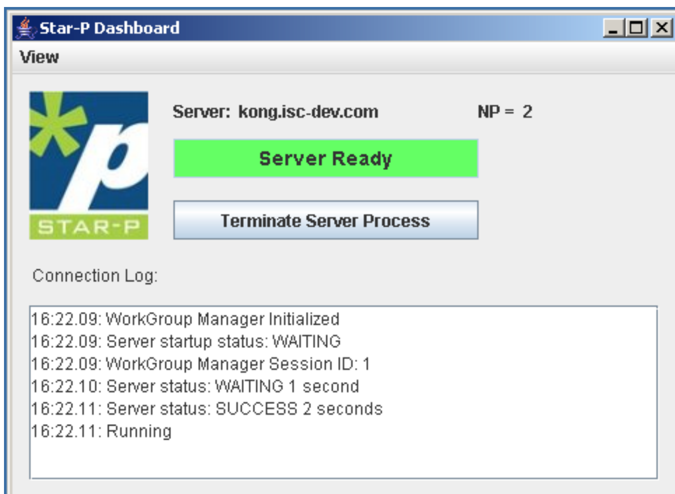
Star-P® can also be started from a Windows command line prompt using the `starp` command. A full description of the `starp` command and its options is provided in the section "Star-P® Start-Up Command Line Options", or type `starp --help` at the Windows command line.

Star-P® Dashboard

The Star-P® Dashboard is designed to

- inform the user of the progress of Star-P® server startup
- inform the user of the connection status of the Star-P® client and server
- provide an interface to allow the user to kill the server should it be necessary.

Figure 2-3 Star-P® Dashboard Interface



The Server Status window displays information about the server startup process and information about the success or failure of Kill button operations.

The Server Status window displays information about the server startup process, information about connectivity to the server, and information about the success or failure of kill button operations.

The server status light on the dashboard provides a simple visual indicator representing the primary set of possible states. At any time, it may display one of the following values:

- Server Initializing
- Server Ready
- Server Busy
- Connection Lost

During the start-up phase, the dashboard will indicate that the server is initializing. Then when a command is submitted to the server, it switches to the “busy” state, and returns to the “ready” state when the command completes. If connectivity to the server is lost at any time, this will be reflected by the status light. Connectivity is tested by periodic heartbeats that pass between the client and the server.

By default, the dashboard always appears when connecting to the Star-P® server. The dashboard can then be hidden or shown using the following pair of commands, which take no arguments:

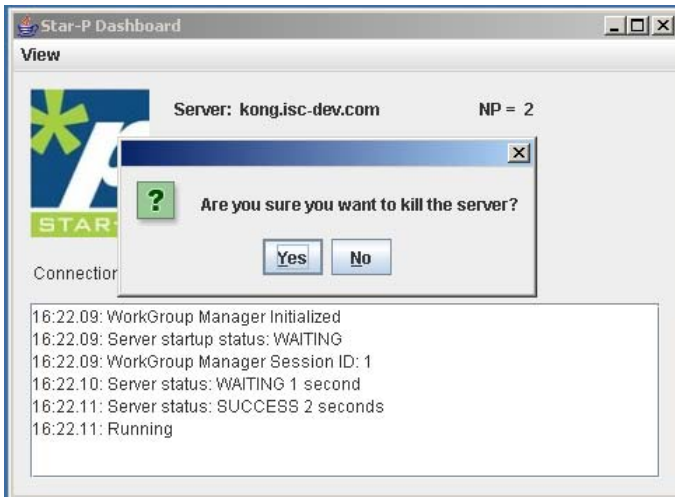
```
ppshowdashboard
pphidedashboard
```

If you desire to change the default settings for dashboard initialization, then you can uncomment the environment variable line `starpd.dashboard.no_gui=1` in the

`starpd.properties` file, located in the `/`<Star-P® `install directory`>`/config` directory. By initializing a Star-P® session with this setting, the Star-P® dashboard is not accessible during that Star-P® session.

The Kill Star-P® Server button is not intended for routine use, but only for situations where the user is unable to exit Star-P® in the usual way. Upon pressing the Kill Star-P® Server button, the user will click Yes when the confirmation dialog appears.

Figure 2-4 Star-P® Kill Button Confirmation



The Star-P® Dashboard opens set to Always On Top mode. However it can be minimized or the user can unset Always On Top using the View menu.

If you are running Star-P® on a system without graphical display capability (for example, a UNIX shell with no DISPLAY environment set), the Dashboard will not be visible or accessible.

Star-P® Sample Session

The use of Star-P® can best be illustrated with a sample session:

First, we check to see whether the server is alive, and the number of processes running.

```
>> np
ans =
8
```

Next, we create a 100x100 random dense matrix distributed by columns.

```
>> App = rand(100,100*p);
App =
        ddense object: 100-by-100p
```

Then, we create a 100x100 random dense matrix distributed by rows.

```
>> Bpp = randn(100*p,100);
```

Using a standard MATLAB instruction, we can solve the system AX=B:

```
>> Xpp = App\Bpp;
```

Now we can check the accuracy of our answer.

```
>> norm(App*Xpp-Bpp)
ans =
3.4621e-13
```

Next we can get information about variables in our current workspace.

```
>> ppwhos
Your variables are:
  Name        Size          Bytes      Class
  App         100x100p      80000      ddense array
  Bpp         100px100      80000      ddense array
  Xpp         100px100      80000      ddense array
  ans         1x1           8          double array
Grand total is 30001 elements using 240008 bytes
MATLAB has a total of 1 elements using 8 bytes
Star-P® server has a total of 30000 elements using 240000 bytes
```

Finally, to end Star-P® execution, you can use either the `quit` or the `exit` command:

```
 >> quit
your_system =>
```

At this point you are ready to write a Star-P® program or port a MATLAB program to Star-P®.

User Specific Star-P® Start-Up Configuration

You may have a set of Star-P® options that you want to choose every time you run Star-P®. Just as MATLAB will execute a `startup.m` file in the current working directory when you start MATLAB, Star-P® will execute a `ppstartup.m` file. Note that Star-P® itself executes some initial commands to create the link between the Star-P® Client for use with MATLAB and the Star-P® server. The `ppstartup.m` file will be executed after those Star-P® initialization commands. Thus the order of execution is:

- `startup.m` % MATLAB configuration commands
- Star-P®-internal initialization commands
- `ppstartup.m` % Your Star-P® configuration commands

For example, this mechanism can be useful for choosing a particular sparse linear solver to use (see "ppsetoption" documentation in "Star-P® Functions") or for loading your own packages (see the "Star-P® Software Development Kit (SDK) Tutorial and Reference Guide").

Star-P® Start-Up Command Line Options

Star-P® can be used with default options enabled, but advanced users might prefer to override defaults at start-up time. The start-up executable is named `starp`. The `starp` application reads its default start-up options from the `starpd.properties` file. For information on how to edit these properties directly, please see the section titled Administration Topics in the Star-P® Installation and Configuration Guide.

Note: You can get help regarding Star-P® startup options by executing the following command: `starp --help`.

The `starp` executable provides the following command line options:

- `-a, --hpcaddress hpcaddress[node1,node2,node3,...,nodeN]`

  Hostname or address of HPC to which to connect. Also may be a comma delimited list of machines comprising a cluster, head node first.

- `-c, --config config_file`

  The configuration file to load.

- `-d <pack> | <scatter>`

  Distribute Star-P® processes when not using a workload manager. Where acceptable values are one of the following:

Option Value	Description
`scatter`	Distribute processes in a round-robin fashion. Default.
`pack`	Fill up individual nodes before allocating processes on additional nodes.

- `-e, --serverenv <serverenv>`

  Environment variables to be set on the server

- `-f, --filter`

Run MATLAB in filter mode so that it reads from `stdin` and writes to `stdout`, for testing.

- `-h, --help`

  Print help text associated with the other arguments you provide.

- `-j, --wlmargs <wlmargs>`

  Arguments to be passed to the Workload Manager. These arguments will override default Workload Manager options normally supplied by Star-P®.

- `-l, --license <license>`

  Specify the location of the license file (overriding `LM_LICENSE_FILE`)

- `-m, --machine <machine>`

  Specify a machine file (must be a client side file)

- `-o, --cfgopt <cfgopt=value>`

  Set a configuration file option

- `-p, --numprocs numprocs`

  Number of processes to request.

- `-q, --wlmqueue <wlmqueue>`

  Workload Manager queue to be used by this Star-P® session

- `-r, --startcommand <startcommand>`

  Start Star-P® and execute the command `<startcommand>`

- `-s, --starppath starp_path`

  Path to Star-P® installation on the HPC

- `--sshport <sshport>`

  Specify a non-standard SSH port for communication with the HPC server

- `-t, --datadir data_path`

  Path that will be used by the HPC Server for file I/O. Star-P® HPC Server reads and writes data to the directory you specify with this path.

- `-u, --hpcuser`

username used to SSH to HPC

- `-v, --version`

  Print the Star-P® version number and exit

- `-w, --wlmextra <wlmextra>`

  Extra arguments to be passed to the Workload Manager, will not override Workload Manager options normally supplied by Star-P®.

- `-x, --exclude <exclude>`

  Specify which nodes of a cluster not to use (mutually exclusive with “use”)

- `-z, --use <use>`

  Specify which nodes of a cluster to use (mutually exclusive with “exclude”)

When running in a cluster, it is also useful to understand the precedence order of potential machine files.

- Any nodes specified in a machine file passed in using `-m`, or specified in a `-x` or `-z` option that are not also included in the default machine file, will not be used by `<starp>`.
- A user default machine file (`~/.starp/.config/machine_file.user_default`) by default, or, `<starp-usr-config>/<usr>/` if overwritten during the installation, will take precedence over the system default machine file (`<StarP_dir>/config/machine_file.system_default`) and will not need to represent a subset of the system default machine file.
- `-m,--machine machine_file_path`

  The path to a machine file to be used for this instance of `starp`. The file format is one machine name per line, with no empty lines at the end of the file. `node` specified by `-a` argument must be included in the file. Example contents of this file would be: `node1 node2 ... nodeN`

- `-x,--exclude [node] or [node1,node2,...,nodeN] or [node2-nodeN]`

  Exclude a node, a set of nodes or a range of nodes from the current instance of `starp`. This argument will be used as a modifier against either a machine file passed in using the `-m` argument, or against either the user's or the system's default machine file. This flag is mutually exclusive with `-z`.

- `-z,--use [node] or [node1,node2,...,nodeN] or [node2-nodeN]`

  Use a node, a set of nodes or a range of nodes for the current instance of starp. This argument will be used as a modifier against either a machine file passed in using the

`-m` argument, or against either the user's or the system's default machine file. This flag is mutually exclusive with `-x`.

By providing command-line options, you can override some of the information normally supplied by the `starpd.properties` file. The following example shows the minimal set of command-line options required for running Star-P®. In this case, the command would cause MATLAB to start up, running eight Star-P® Server processes on a machine with the hostname `altix` as the user `joe`:

```
starp -a altix -p 8 -s /usr/local/starp -u joe -t /home/joe
```

Examples

The following are useful examples:

- If you are running on a cluster and you want to specify a list of nodes in the cluster to be excluded from a particular run of `<starp>` (perhaps `node3` and `node7` are down for maintenance), your `<starp>` command line would look like this:

```
starp -a node1 -x node3,node7 -p 8 -s /usr/local/starp -u joe -t /home/joe
```

Using this command line, a new machine file for this one run of `<starp>` will be generated using the default machine file, but with node3 and node7 removed.

Note: If node3 or node7 are not members of the default machine file, they will be ignored as defined in Cluster Configurations at the end of this section.

- If you are running on a cluster and you want to specify a range of nodes in the cluster to be excluded from a particular run of `<starp>` (perhaps a rack of nodes has been taken offline), your <starp> command line would look like this:

```
starp -a node1 -x node3-node14 -p 8 -s /usr/local/starp -u joe -t /home/joe
```

Using this command line, a new machine file for this one run of `<starp>` will be generated using the default machine file, but with `node3` through `node14` utilized.

Note: If either `node3` or `node14` are not members of the default machine file, `<starp>` will return a "bad range" error.

Note: If `node14` appears before `node3` in the default machine file, `<starp>` will return a "bad range" error.

- If you are running on a cluster and you want to specify a custom machine file for a particular run of `<starp>`, your `<starp>` command line would look like this:

```
starp -a node1 -m [machine file path] -p 8 -s /usr/local/starp -u joe -t
/home/joe
```

Using this command line, the machine file specified by `[machine file path]` will be used for this one run of `<starp>`.

Note: The machine file specified by [machine file path] must represent a subset of the user's or system's default machine file.

- If you are running on a cluster and you want to specify a custom machine file for a particular run of `<starp>` and you only want to use a subset of that machine file, your `<starp>` command line would look like this:

```
starp -a node1 -m [machine file path] -z node3-node14 -p 8 -s /usr/local/starp
-u joe -t /home/joe
```

Using this command line, the machine file specified by [machine file path] will be used for this one run of `<starp>`.

Note: The machine file specified by `[machine file path]` must represent a subset of the user's or system's default machine file.

Note: If either node3 or node14 are not members of the default machine file, `<starp>` will return a "bad range" error.

Note: If node14 appears before node3 in the default machine file, `<starp>` will return a "bad range" error.

Launching Star-P® with a MATLAB .m script

A limited form of batch processing can be used in Star-P® that is separate from the realm of full workload management systems that are also supported by Star-P®. This process involves use of command line options listed above as well as the name of a desired script you wish to run within your VHLL environment. If you wish to run a `.m` script named `myscript.m`, you would redirect the contents of a MATLAB .m file into the starp command like this:

```
starp -a server -u user -p 4 -t . -s /usr/local/starp-<version> < myscript.m
```

Cluster Configurations

There are 2 files and several command line arguments that can affect cluster configuration.

- If no `machine_file.user_default` exists, the system will create one for the current session containing only the name of the current machine.
- If a `machine_file.system` default exists, a `machine_file.user_default` does not exist, and no (related) command line args are specified, then the `machine_file.system_default` will be used.
- If a `machine_file.user_default` exists, it takes precedence over any `machine_file.system_default` file.
- If command line args are used, the args must represent a subset of the selected `machine_file`.

INTER*CTIVE supercomputing

Chapter 3

Data Parallelism with Star-P® and MATLAB

This chapter contains information on creating, manipulating, loading, and saving data in parallel and includes the following:

- "Star-P® Naming Conventions"
- "Examining Star-P® Data"
- "Special Variables: p and np"
- "Supported Data Types"
- "Creating Distributed Arrays"
- "Types of Distributions"
- "Propagation of Distribution"
- "Explicit Data Movement with ppback and ppfront"
- "Loading And Saving Data on the Parallel Server"

The Star-P® extensions to MATLAB allow you to parallelize computations by declaring data as distributed. This places the data in the memory of multiple processors. Once the data is distributed, then operations on the distributed data will run implicitly in parallel. Since declaring the data as distributed requires very little code in a Star-P® program, performing the MATLAB operations in parallel requires very little change from standard, serial MATLAB programing.

Another key concept in Star-P® is that array dimensions are declared as distributed, not the array proper. Of course, creating an array with array dimensions that are distributed causes the array itself to be distributed as well. This allows the distribution of an array to propagate through not only computational operators like + or `fft`, but also data operators like `size`. Propagation of distribution is one of the key concepts that allows large amounts of MATLAB code to be reused directly in Star-P® without change.

Star-P® Naming Conventions

Star-P® commands and data types generally use the following conventions, to distinguish them from standard MATLAB commands and data types:

- Most Star-P® commands begin with the letters **pp**, to indicate parallel. For example, the Star-P® `ppload` command loads a distributed matrix from local files. Exceptions to this rule include the split and bcast commands.
- Star-P® data types begin with the letter `d`, to indicate "distributed". For example, the Star-P® `dsparse` class implements distributed sparse matrices.

The following convention for displaying Star-P® related commands and classes is used throughout this chapter.

Command/Variable	Font
p & other `dlayout` variables	**bold green font**
Distributed variables	**bold blue font**
Star-P® functions	**bold black font**

Examining Star-P® Data

This section describes how you can look at your variables, see their sizes and determine whether they reside on the client as a regular MATLAB object or on the server as a Star-P® object. The MATLAB `whos` command is often used for this function, but `whos` is unaware of the true sizes of the distributed arrays. Star-P® supports a similar command called `ppwhos`. Here is sample calling sequence and output:

```
>> n = 1000;
>> app = ones(n*p);
>> bpp = ones(n*p,n);
>> ppwhos
Your variables are:
  Name        Size             Bytes        Class
  app         1000x1000p       8000000      ddense array
  bpp         1000px1000       8000000      ddense array
  n           1x1              8            double array
Grand total is 2000001 elements using 16000008 bytes
MATLAB has a total of 1 elements using 8 bytes
Star-P® server has a total of 2000000 elements using 16000000 bytes
```

Note that each dimension of the arrays includes the "p" if it is distributed. Size and Bytes reflect the size on the server for distributed objects, and transition naturally to scientific notation when their integer representations get too large for the space.

```
>> n = 2*10^9;
>> xpp = ones(1,n*p);
>> ppwhos
Your variables are:
  Name        Size                Bytes             Class
  n           1x1                 8                 double array
  xpp         1x2000000000p       1.600000e+10      ddense array
Grand total is 2000000001 elements using 1.600000e+10 bytes
MATLAB has a total of 1 elements using 8 bytes
Star-P® server has a total of 2000000000 elements using 1.600000e+10 bytes
```

Note that the MATLAB `whos` command, when displaying distributed objects, only shows the amount of memory they consume on the front-end, not including their server memory. This does not reflect their true extent. For example, the output from `whos` for the session above looks like the following:

```
>> n = 1000;
>> app = ones(n*p);
>> bpp = ones(n*p,n);
>> whos
  Name          Size                  Bytes       Class
  app           1000x1000             1728        ddense
  bpp           1000x1000             1728        ddense
  n             1x1                   8           double
```

The `ppwhos` command gives the full and correct information.

Reusing Existing Scripts

The following routine is the built-in MATLAB routine to construct a Hilbert matrix:

```
>> H = hilb(4096);
```

Because the operators in the routine (`:`, `ones`, `subsasgn`, `transpose`, `rdivide`, `+`, `-`) are overloaded to work with distributed matrices and arrays, typing the following would create a 4096 by 4096 Hilbert matrix on the server.

```
>> Hpp = hilb(4096*p)
Hpp =
        ddense object: 4096-by-4096p
```

By exploiting MATLAB's object-oriented features in this way, existing scripts can run in parallel under Star-P® with minimal modification.

Examining/Changing Distributed Matrices

As a general rule, you will probably not want to view an entire distributed array, because the arrays that are worth distributing tend to be huge. For example, the text description of 10 million floating-point numbers is vast. But looking at a portion of an array can be useful. To look at any portion of a distributed array bigger than a scalar, it will have to be transferred explicitly to the client MATLAB program. But looking at a single element of the array can be done simply. Remember from above that result arrays that are 1x1 matrices are created as local arrays on the MATLAB client.

```
>> app = rand(1000*p,1000);
>> size(app)
ans =
     1000p    1000
>> app(423,918)
ans =
    0.2972
>> app(2,3), app(2,3) = 5; app(2,3)
ans =
    0.8410
ans =
     5
>> app(1:5,1:5)
ans =
        ddense object: 5p-by-5
```

As you can see, examining a single element of the array returns its value. Examining multiple elements creates another distributed object, which remains on the server, as in the last command above. To see the values of these elements, you will need to use `ppfront` to move them to the front-end. For information on `ppfront` and `ppback` see "Explicit Data Movement with ppback and ppfront".

```
>> app = rand(1000*p,1000);
>> ppfront(app(1:5,1:5))
ans =
    0.9256    0.3075    0.4824    0.7822    0.6045
    0.6478    0.7912    0.8058    0.8359    0.0778
    0.4349    0.7521    0.0216    0.5591    0.2883
    0.9269    0.9317    0.9427    0.1967    0.3970
    0.2723    0.2860    0.3665    0.1203    0.3310
```

Special Variables: p and np

In Star-P® you use two special variables to control parallel programming. While they are technically functions, you can think of them as special variables. The first is `p`, which is used in declarations such as the following to denote that an array should be distributed for parallel processing.

```
>> zpp = ones(100*p);
```

The second variable with special behavior is `np`, denoting the number of processors that have been allocated to the user's job for the current Star-P® session. Because these are not unique names, and existing MATLAB programs may use these names, care has been taken to allow existing programs to run, as described here. The behavior described here for **p** and `np` is the same as the behavior for MATLAB built-in variables such as `i` and `eps`, which represent the imaginary unit and floating-point relative accuracy, respectively.

The variables **p** and `np` exist when Star-P® is initiated, but they are not visible by the `whos` or `ppwhos` command.

After Star-P® initializes in a new session, the following commands yield no output.

```
>> whos
>> ppwhos
```

Even though the variables **p** and `np` do not appear in the output of `whos` or `ppwhos`, they do have values:

```
>> p
ans =
      1p
>> np
ans =
      8
```

The variable `np` will contain the number of processors in use in the current Star-P® session. In this example, the session was using eight processors.

Because these variable names may be used in existing programs, it is possible to replace the default Star-P® definitions of **p** and `np` with your own definitions, as in the following example:

```
>> p
ans =
     1p
>> np
ans =
     8
>> n = 100;
>> app = ones(n*p);
>> bpp = ones(n*p,n);
>> cpp = bpp*bpp;
>> p = 3.14;
>> z = p*p;
>> z
z =
    9.8596
>> p
p =
    3.1400
>> ppwhos
```

```
Your variables are:
  Name       Size          Bytes      Class
  app        100x100p      80000      ddense array
  ans        1x1           8          double array
  bpp        100px100      80000      ddense array
  cpp        100px100      80000      ddense array
  n          1x1           8          double array
  p          1x1           8          double array
  z          1x1           8          double array
Grand total is 30004 elements using 240032 bytes
MATLAB has a total of 4 elements using 32 bytes
Star-P® server has a total of 30000 elements using 240000 bytes
>> clear p
>> p
ans =
     1p
>> ppwhos
Your variables are:
  Name       Size          Bytes      Class
  app        100x100p      80000      ddense array
  ans        1x1           258        dlayout array
  bpp        100px100      80000      ddense array
  cpp        100px100      80000      ddense array
  n          1x1           8          double array
  z          1x1           8          double array
Grand total is 30003 elements using 240274 bytes
MATLAB has a total of 3 elements using 274 bytes
Star-P® server has a total of 30000 elements using 240000 bytes
```

Note that in the first output from `ppwhos`, the variable **p** is displayed, because it has been defined by the user, and it works as a normal variable. But once it is cleared, it reverts to the default Star-P® definition. If you define `p` in a function, returning from the function acts like a `clear` and the definition of **p** will revert in the same way.

The variable name `np` works in the same way.

Assignments to p

The variable **pp** is a synonym for **p**. If you use a mechanism to control client versus Star-P® operation (execution solely on the client versus execution with Star-P®), the assignment of `p = 1` anywhere in the MATLAB script will alter the **p** function. In this case, use a construct similar to the following:

```
if StarP
    p = pp;
else
    p = 1;
end
```

Anytime you clear the variable `p`, for example `clear p`, the symbolic nature of **p** is restored.

Supported Data Types

Real and Complex Data

Real and complex numbers in Star-P® are supported as in MATLAB. Matrices of double precision real and complex data can be directly created and manipulated by use of the `complex`, `real`, `imag`, `conj`, and `isreal` operators and the special variables `i` and `j` (equal to the square root of -1, or the imaginary unit), and they can be the output of certain operators.

Note: Complex integer types are not supported within the Star-P® Task Parallel Engine (TPE). However, it does support floating point complex types such as double.

```
>> n   = 1000;
>> app  = rand(n*p,n)
app =
        ddense object: 1000p-by-1000
>> bpp  = rand(n*p,n)
bpp =
        ddense object: 1000p-by-1000
>> cpp  = app + i*bpp
cpp =
        ddense object: 1000p-by-1000
>> ccpp = conj(cpp)
ccpp =
        ddense object: 1000p-by-1000
>> dpp  = real(cpp)
dpp =
        ddense object: 1000p-by-1000
>> epp  = imag(cpp)
epp =
        ddense object: 1000p-by-1000
>> fpp  = complex(app)
fpp =
        ddense object: 1000p-by-1000
>> ppwhos
Your variables are:
  Name       Size            Bytes         Class
  app        1000px1000      8000000       ddense array
  bpp        1000px1000      8000000       ddense array
  cpp        1000px1000      16000000      ddense array (complex)
  ccpp       1000px1000      16000000      ddense array (complex)
  dpp        1000px1000      8000000       ddense array
  epp        1000px1000      8000000       ddense array
  fpp        1000px1000      16000000      ddense array (complex)
  n          1x1             8             double array
Grand total is 7000001 elements using 80000008 bytes
MATLAB has a total of 1 elements using 8 bytes
Star-P® server has a total of 7000000 elements using 80000000 bytes
```

Besides these direct means of constructing complex numbers, they are often the result of specific operators, perhaps the most common example being FFTs.

```
>> app = rand(1000,1000*p)
app =
        ddense object: 1000-by-1000p
>> bpp = fft2(app)
bpp =
        ddense object: 1000-by-1000p
>> ppwhos
Your variables are:
  Name       Size            Bytes         Class
  app        1000x1000p      8000000       ddense array
  bpp        1000x1000p      16000000      ddense array (complex)
Grand total is 2000000 elements using 24000000 bytes
MATLAB has a total of 0 elements using 0 bytes
Star-P® server has a total of 2000000 elements using 24000000 bytes
```

Creating Distributed Arrays

Using Star-P®, data can be created as distributed in several ways:

- The data can be initially allocated as distributed using the `*p` syntax in conjunction with a variety of constructor routines such as `zeros`, `ones`, `rand`, `randn`, `spones`, `sprand`, `sprandn`, as described in "Distributed Data Creation Routines".
- An array bounds variable can be created using the `*p` syntax, which is then used to create distributed arrays.
- Most commonly, a distributed object can be created by propagation when an operation on a distributed object creates a new distributed object, as described in "Propagating the Distributed Attribute".
- The data can be loaded from disk to a distributed object with the `ppload` routine, which is similar to the MATLAB `load` routine.
- The data can be explicitly distributed with the `ppback` server command.
- A new distributed array can be created by indexing a section of a distributed array, as described in "Indexing into Distributed Matrices or Arrays".

The *p Syntax

The symbol `p` means "distributed" and can add that attribute to a variety of other operators and variables by the multiplication operator *. Technically, `p` is a function, but it may be simpler to think of it as a special variable. Any scalar that is multiplied by `p` will be of class `dlayout`. For more information about `p`, see "Special Variables: p and np".

```
>> p
ans =
```

```
     1p
>> whos
  Name       Size               Bytes  Class      Attributes
  ans        1x1                  362  dlayout
```

Note: While it might seem natural to add a `*p` to the bounds of a `for` loop to have it run in parallel, unfortunately that doesn't work. The simplicity of this type of approach has not been lost on the designers of Star-P®, and a functionality of this type or similar may appear in future releases.

Distributed Data Creation Routines

Matrices can be declared as distributed in Star-P® by appending `*p` to one or more of the dimensions of the matrix. For example, any of the following will create `App` as a distributed dense matrices:

```
>> App = rand(100*p,100  );
>> App = rand(100  ,100*p);
>> App = rand(100*p);
>> App = rand(100*p,100*p);
>> App = 1:100*p;
```

The first and second examples create matrices that are distributed in the first and second dimensions, respectively. The last two examples create a matrix that is distributed in the second dimension. For more detail, see "Types of Distributions".

Similarly, distributed sparse matrices can be created by the following declaration:

```
App = sprandn(100*p,100,0.03);
```

You can declare multidimensional arrays to be distributed by appending `*p` to any one dimension of the matrix. Star-P® supports the same set of data creation operators for multidimensional arrays as MATLAB does.

The operators `ones`, `zeros`, `rand`, `sprand`, `eye`, and `speye` all have the same behavior as `randn` and `sprandn`, respectively, for dense and sparse operators. The `horzcat` and `vertcat` operators work in the obvious way; concatenation of distributed objects yields distributed objects.

The `meshgrid` operator can create distributed data in a similar way, although this example may not be the way you would use it in practice:

```
>> [xpp ypp] = meshgrid(-2:.2:2*p,-2:.2:2*p);
>> size(xpp), size(ypp)
ans =
     21      21p
ans =
     21      21p
```

Also, the `diag` operator extends a distributed object in the obvious way.

```
>> qpp = rand(100*p,1);
>> rpp = diag(qpp,0);
>> size(qpp), size(rpp)
ans =
     100p      1
ans =
     100p    100
```

The `reshape` command can also create distributed arrays, even from local arrays.

```
>> a = rand(100,100);
>> app = reshape(a,100,100*p)
app =
        ddense object: 100-by-100p
>> ppwhos
Your variables are:
  Name       Size          Bytes      Class
  a          100x100       80000      double array
  app        100x100p      80000      ddense array
Grand total is 20000 elements using 160000 bytes
MATLAB has a total of 10000 elements using 80000 bytes
Star-P® server has a total of 10000 elements using 80000 bytes
```

The details of these different distributions are described in "Types of Distributions".

Note: The data sizes shown in the examples illustrate the functionality of Star-P® but do not necessarily reflect the sizes of problems for which Star-P® will provide significant benefit.

Distributed Array Bounds

Some programs or functions take as input not an array, but the bounds of arrays that are created internally. The `*p` syntax can be used in this situation as well, as shown in the following:.

```
>> n = 1000*p;
>> whos
      Name       Size       Bytes  Class      Attributes
      n          1x1          362  dlayout
>> App = rand(n)
App =
        ddense object: 1000-by-1000p
```

Indexing into Distributed Matrices or Arrays

Indexing allows creation of new matrices or arrays from subsections of existing matrices or arrays. Indexing on distributed matrices or arrays always creates a distributed object, unless

the result is a scalar, in which case it is created as a local object. Consider the following example:

```
>> app = rand(1000*p);
>> bpp = rand(1000*p);
```

Operations that result in distributed matrices:

```
% Indexing (sub)sections of the elements of a distributed
% array result in a distributed object
>> cpp = app(1:end,1:end);
>> dpp = app(18:23,47:813);
>> fpp = app(:);     %linearize 2D array into 1D vector doing assignment
                     %via the linearization approach works naturally
>> bpp(:) = 0;
```

Operations that result in local objects, data transferred to front-end:

```
>> e = app(47,418); % scalar goes to front-end
>> nnz(app)         % scalar answer 'ans' goes to front-end
ans =
     1000000
>> nnz(bpp)         % scalar answer 'ans' goes to front-end
ans =
     0
>> ppwhos
Your variables are:
  Name       Size            Bytes        Class
  app        1000x1000p      8000000      ddense array
  ans        1x1             8            double array
  bpp        1000x1000p      8000000      ddense array
  cpp        1000x1000p      8000000      ddense array
  dpp        6x767p          36816        ddense array
  e          1x1             8            double array
  fpp        1000000px1      8000000      ddense array
Grand total is 4004604 elements using 32036832 bytes
MATLAB has a total of 2 elements using 16 bytes
Star-P® server has a total of 4004602 elements using 32036816 bytes
```

In order to propagate the distribution of data as broadly as possible, Star-P® interprets indexing operations on distributed objects as creating new distributed objects, hence the distributed nature of `bpp` and `dpp` in the example. The one exception is where the resulting object is a scalar (1x1 matrix), which always resides on the front-end.

Note that creating a new matrix or array by indexing, as in the creation of `dpp` above, may involve interprocessor communication on the server, as the new matrix or array will need to be evenly distributed across the processors (memories) in use, and the original position of the data may not be evenly distributed.

It may seem logical that you could create a distributed object by adding the `*p` to the left-hand side of an equation, just as you can to the right-hand side. But this approach doesn't work, either in MATLAB in general or in Star-P® specifically for distributed arrays.

```
>> a*4 = rand(100,100);
??? a*4 = rand(100,100);
Error: The expression to the left of the equals sign is not a valid target for
an assignment.
>> a*p = rand(100,100);
??? a*p = rand(100,100);
Error: The expression to the left of the equals sign is not a valid target for
an assignment.
>> a(:,:) = rand(100,100);
>> a(:,:*p) = rand(100,100);
??? a(:,:*p) = rand(100,100);
Error: Unexpected MATLAB operator.
```

Note: There is an incompatibility between MATLAB and Star-P® in this area. In MATLAB, when you type the command `app` or `bpp`, as soon as that assignment is complete, you can modify either `app` or `bpp` and know that they are distinct entities, even though the data may not be copied until later. For technical reasons Star-P® can get fooled by this deferment. Thus if you modify either `app` or `bpp`, the contents of both `app` and `bpp` get modified. Because of the semantics of the MATLAB language, this is only relevant for assignments of portions of `app` or `bpp`; i.e., `app(18,:) = ones(1,100*p)` or `app(1234) = 3.14159`. There are several ways to avoid the deferment and force the data to be copied immediately to avoid this problem. One example would be (for a 2D matrix) to do the copy with `app = bpp(:,:)`. Another example that works for all non-logical arrays is `app = +bpp`.

Note: Related to the previous note, if a shallow copy of a variable is created using the command `app = bpp`, then the deletion of either `app` or `bpp` using `clear` or `ppclear` on `app` or `bpp` will delete the data for both `app` and `bpp` but will not delete the symbols for both variables. To avoid the this scenario, use an assignment statement of the form `app = bpp(:,:)` or `app = +bpp`.

Types of Distributions

Distributed Dense Matrices and Arrays

Star-P® uses the MATLAB terminology of two-dimensional matrices and multidimensional arrays of numbers. Like MATLAB, a full set of operations is defined for matrices, but a smaller set for arrays. Arrays are often used as repositories for multiple matrices and operated on in 2D slices, so the set of supported operators reflects this.

Star-P® supports row and column distribution of dense matrices. These distributions assign a block of contiguous rows/columns of a matrix to successive processes.

A two-dimensional distributed dense matrix can be created with any of the following commands:

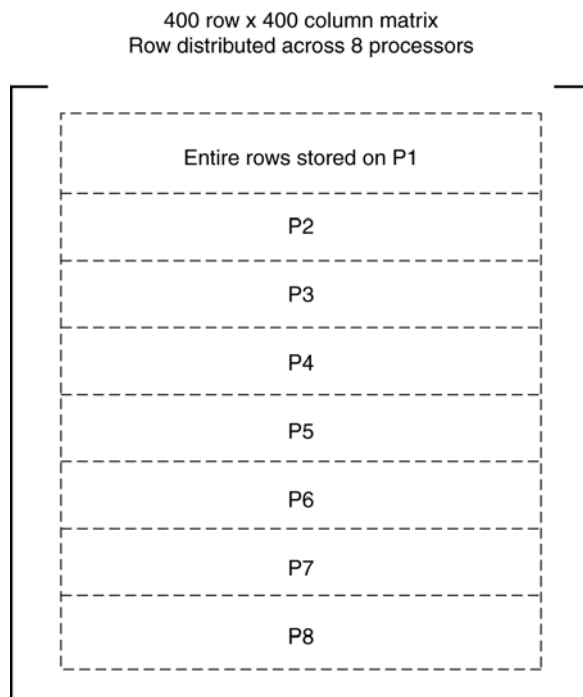
```
>> bpp = rand(400  ,400*p);
>> app = rand(400*p,400  );
```

The `*p` designates which of the dimensions are to be distributed across multiple processors.

Row distribution

In the example above, `app` is created with groups of rows distributed across the memories of the processors in the parallel server. Thus, with 400 rows on 8 processors, the first 400/8 == 50 rows would be on the first processor, the next 50 on the second processor, and so forth, in a style known as row-distributed. Figure 3-1 illustrates the layout of a row-distributed array.

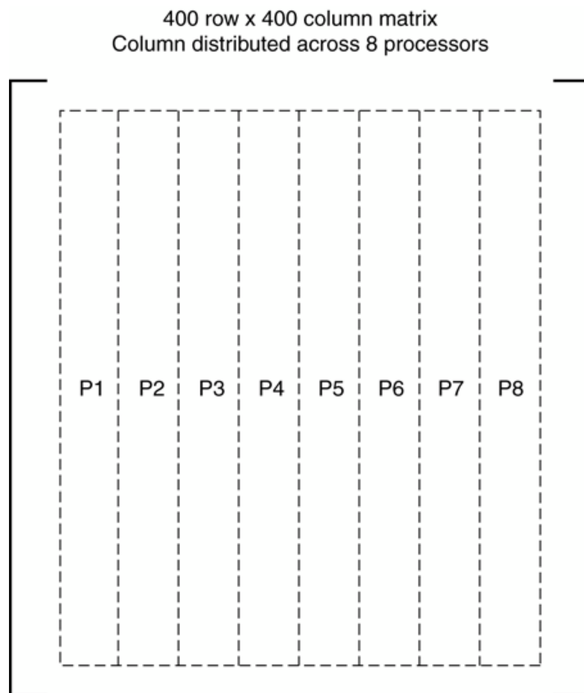
Figure 3-1 Row Distribution



Column distribution

Column-distribution works just the same as row distribution, except column data is split over available processors; `bpp` is created that way above. Figure 3-2: illustrates the layout of a column distributed array. When a `*p` is placed in more than one dimension, the matrix or multi-dimensional array will be distributed in the rightmost dimension containing a `*p`. For example, if there was a `*p` in both dimensions of the constructor for a two dimensional matrix, it would result in a column distribution.

Figure 3-2 Column Distribution



Distributed Dense Multidimensional Arrays

Distributed multidimensional arrays are also supported in Star-P®. They are distributed on only a single dimension, like row- and column-distributed 1D or 2D matrices. Hence if you create a distributed object with the following command, then **app** will be distributed on the third dimension:

```
>> n = 10;
>> app = rand(n,n,n*p,n);
```

If you should happen to request distribution on more than one dimension, the resulting array will be distributed on the rightmost non-singleton requested dimension. A singleton is defined as a matrix dimension with a size equal to 1.

```
>> app = zeros(10*p,10,10*p,10*p,10*p)
app =
        ddense object:10-by-10-by-10-by-10-by-10p
```

Multidimensional distributed dense arrays support a subset of operators on 2D arrays. See the full list in "Star-P® Functions".

Distributed Sparse Matrices

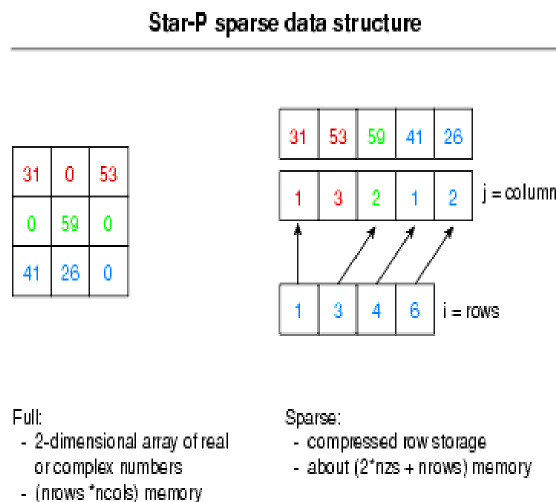
Distributed sparse matrices in Star-P® use the compressed sparse row format. Distributed sparse matrices are represented as `dsparse` objects. This format represents the nonzeros in

each row by the index (in the row) of the nonzero and the value of the nonzero, as well as one per-row entry in the matrix data structure. This format consumes storage proportional to the number of nonzeros and the number of rows in the matrix. Sparse matrices in Star-P® typically consume 12 bytes per double-precision element, compared to 8 bytes for a dense matrix. The matrix is distributed by rows, with the same number of rows per processor (modulo an incomplete number on the last processor(s)). Note that, as a consequence, it is possible to create sparse matrices that do not take advantage of the parallel nature of the server. For instance, if a series of operations creates a distributed sparse row vector, all of that vector will reside on one processor and would typically be operated on by just that one processor.

How Star-P® Represents Sparse Matrices

While one might imagine the data stored in three columns headed by i, j, A_{ij}, in fact the data is stored as described by this picture:

Figure 3-3 Star-P® Sparse Data Structure



Notice that if you subtract the row index vector from itself shifted one position to the left, you get the number of elements in a row. This makes it clear what to do if element (2,2) with the value of 59 gets deleted in Figure 3-3:, resulting in no elements left in the second row. The indices would then point to [1 3 3 5]. In other words, noticing that the number of non-zeros per row is [2 0 2] in this case, you could perform a `cumsum` on [1 2 0 2] and obtain [1 3 3 5].

Figure 3-4 Star-P® Distributed Sparse Data Structures

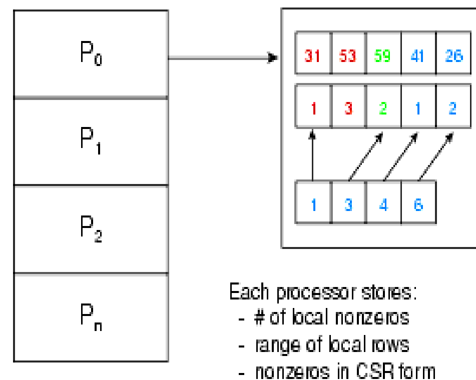


Figure 3-4 shows what happens when the sparse data structure from Figure 3-3: is distributed across multiple processors by Star-P®. The number of rows is divided among the participating processors, and each processor gets the information about its local rows. Thus operations that occur on rows can happen locally on a processor; operations that occur on columns require communication among the processors.

Distributed Cell Objects (dcell)

The dcell is analogous to MATLAB cells. The `dcell` type is different from the other distributed matrix or array types, as it may not have the same number of data elements per `dcell` iteration and hence doesn't have the same degree of regularity as the other distributions. This enables dcells to be used as return arguments for `ppevalsplit()`. For more information on ppevalsplit, see "ppevalsplit" in "Star-P® Functions".

Combining Data Distribution Mechanisms

The data distribution mechanisms can be combined in a program. For instance, the array **App** can be loaded from a file and then its dimensions used to create internal work arrays based on the size of the passed array.

```
>> ppload imagedata App
>> [rows cols] = size(App)
rows =
        1000
cols =
     1000p
>> Bpp = zeros(rows,cols);
>> ppwhos
Your variables are:
  Name        Size              Bytes         Class
  App         1000x1000p        8000000       ddense array
  Bpp         1000x1000p        8000000       ddense array
  cols        1x1               258           dlayout array
```

```
  rows        1x1               8             double array
Grand total is 2000002 elements using 16000266 bytes
MATLAB has a total of 2 elements using 266 bytes
Star-P® server has a total of 2000000 elements using 16000000 bytes
```

Similarly, input data created by `ones` or `zeros` or `sprand` can be used as input to other functions, scripts, or toolboxes that are not aware of the distributed nature of their input, but will work anyway. For example, the function `foo` is defined as follows:

```
function c = foo(a)       % now executing in the function foo
[rows cols] = size(a);
b = rand(rows,cols);      % creat a symmetric + diagonal matrix
c = b + b' + eye(rows);   % based on the size of the input
% ......
```

In this example, the following code will then work because all the operators in `foo` are defined for distributed objects as well as regular MATLAB objects:

```
>> App = rand(1000*p);
>> Cpp = foo(App)
Cpp =
        ddense object: 1000-by-1000p
>> ppwhos
Your variables are:
  Name        Size              Bytes         Class
  App         1000x1000p        8000000       ddense array
  Cpp         1000x1000p        8000000       ddense array
Grand total is 2000000 elements using 16000000 bytes
MATLAB has a total of 0 elements using 0 bytes
Star-P® server has a total of 2000000 elements using 16000000 bytes
```

These mechanisms are designed to work this way so that a few changes can be made when data is input to the program or initially created, and then the rest of the code can be untouched, giving high re-use and easy portability from standard MATLAB to Star-P® execution.

Mixing Local and Distributed Data

The examples up until now have covered operations that included exclusively local or distributed data. Of course, it is possible to have operations that include both. In this case, Star-P® typically moves the local object from the client to the server, following the philosophy that operations on distributed objects should create distributed objects. In the example here, you can see this by the `pptoc` output showing 80KB received by the server.

```
>> A = rand(100);
>> Bpp = rand(100*p);
>> pptic; Cpp = A + Bpp; pptoc;
Client/server communication report:
  Sent by server: 2 messages, 1.560e+02 bytes
  Received by server: 2 messages, 8.017e+04 bytes
```

```
  Total communication time: 6.706e-03 seconds
Server processing report:
  Duration of calculation on server (wall clock time): 7.675e-03s
  #ppchangedist calls: 0
----------------------------------------------------------------------
Total time: 1.648e-01 seconds
```

And of course, note that all scalars are local, so whenever a scalar is involved in a calculation with a distributed object, it will be sent to the server.

```
>> Bpp = rand(100*p)
Bpp =
        ddense object: 100-by-100p
>> App = Bpp * pi
App =
        ddense object: 100-by-100p
```

The mixing of local and distributed data arrays is not as common as you might think. Remember that Star-P® is intended for solving large problems, so distributed arrays will typically be bigger than the memory of the client system. So, a typically sized distributed array would not have an equal size client array to add to it.

There are cases where mixed calculations can be useful. For example, if a vector and a matrix are being multiplied together, the vector may be naturally stored on the client, but a calculation involving a distributed array will move it to the server.

```
>> qpp = rand(10000*p,16);
>> r = rand(16,1);
>> spp = qpp*r
spp =
         ddense object: 10000p-by-1
```

Distributed Classes used by Star-P®

You may have been wondering about these class types you have been seeing in the output of `ppwhos`, namely `dlayout`, `ddense`, `dsparse`, and `densend`. Classes are the way that MATLAB supports extensions of its baseline functionality, similar to the way C++ and other languages support classes. To create a new class, it must have a name and a set of functions that implement it.

The `ddense` class may be the simplest Star-P® class to understand. It is a dense matrix, just like a MATLAB dense matrix, except it is distributed across the processors (memories) of the HPC server system. When you create a distributed dense object, you will see its type listed by `ppwhos`, as in the following example:

```
>> n = 1000;
>> App = ones(n*p);
>> Bpp = ones(n*p,n);
>> ppwhos
```

```
Your variables are:
  Name       Size           Bytes        Class
  App        1000x1000p     8000000      ddense array
  Bpp        1000px1000     8000000      ddense array
  n          1x1            8            double array
Grand total is 2000001 elements using 16000008 bytes
MATLAB has a total of 1 elements using 8 bytes
Star-P® server has a total of 2000000 elements using 16000000 bytes
```

Creating a new class is simple. Having it do something useful requires operators that know how to operate on the class. MATLAB allows class-specific operators to be in a directory named `@ddense`, in the case of class `ddense`. For instance, if you wanted to know where the routine is that implements the `gradient` operator, you would use the MATLAB `which` command, as in the following example:

```
>> which gradient
/usr/local/matlab/toolbox/matlab/datafun/gradient.m
>> which @ddense/gradient
<starp_root>/matlab/@ddense/gradient.p  % ddense method
>> which @ddensend/gradient
<starp_root>/matlab/@ddensend/gradient.p  % ddensend method
>> which @dsparse/gradient
<starp_root>/matlab/@dsparse/gradient.p  % dsparse method
```

In the above example, `<starp_root>` is the location where the Star-P® client installation took place.

The `which sum` command tells you where the routine is that implements the `sum` operator for a generic MATLAB object. The `which @double/sum` command tells you where the MATLAB code is that implements the `sum` operator for the MATLAB double type. The `which @ddense/sum` command tells you where the Star-P® code is that implements it for the Star-P® `ddense` class. The MATLAB class support is essential to the creation of Star-P®'s added classes.

Similarly to the `ddense` class, the `dsparse` class implements distributed sparse matrices. Since the layout and format of data is different between dense and sparse matrices, typically each will have its own code implementing primitive operators. The same holds for the `ddensend` class implementing multidimensional arrays.

However, as shown in the `hilb` example below, there are non-primitive MATLAB routines which use the underlying primitives that are implemented for `ddense` and `dsparse`. These routines will work in the obvious way, and so no further class-specific version of the routine is necessary.

```
>> which hilb
/usr/local/matlab/toolbox/matlab/elmat/hilb.m
>> which @ddense/hilb
'@ddense/hilb' not found.
>> which @ddensend/hilb
'@ddensend/hilb' not found.
```

The `dlayout` class is not as simple as the `ddense` and `dsparse` classes, because the only function of the `dlayout` class is to declare dimensions of objects to be distributed. Thus, you will see that operators are defined for `dlayout` only where it involves array construction (e.g. `ones`, `rand`, `speye`) and simple operators often used in calculations on array bounds (for example, `max`, `floor`, `log2`, `abs`). The complete set of functions supported by `dlayout` are found in "Supported MATLAB® Functions". The only way to create an object of class `dlayout` is to append a `*p` to an array bound at some point, or to create a distributed object otherwise, as via `ppload`.

To create `dlayout` objects without the `*p` construct we can import data with `ppload` and extract the `dlayout` objects from `size` of the imported variable.

```
>> n = 1000;
>> app = rand(n*p)
app =
        ddense object: 1000-by-1000p
>> [rows, cols] = size(app)
rows =
        1000
cols =
     1000p
>> ppload imagedata App
>> Bpp = inv(App)
Bpp =
        ddense object: 1000-by-1000p
>> [Brows, Bcols] = size(Bpp)
Brows =
        1000
Bcols =
     1000p
>> ppwhos
Your variables are:
  Name       Size            Bytes        Class
  App        1000x1000p      8000000      ddense array
  Bpp        1000x1000p      8000000      ddense array
  Bcols      1x1             258          dlayout array
  Brows      1x1             8            double array
  app        1000x1000p      8000000      ddense array
  cols       1x1             258          dlayout array
  n          1x1             8            double array
  rows       1x1             8            double array
Grand total is 3000005 elements using 24000540 bytes
MATLAB has a total of 5 elements using 540 bytes
Star-P® server has a total of 3000000 elements using 24000000 bytes
```

As a result, `dlayout` is something you may see often in `ppwhos` displays.

Propagating the Distributed Attribute

Since the distributed attribute of matrices and arrays is what triggers parallel execution, the semantics of Star-P® have been carefully designed to propagate distribution as frequently as possible. In general, operators which create data objects as large as their input (`*`, `+`, \ (linear solve), `fft`, `svd`, etc.) will create distributed objects if their input is distributed. Operators which reduce the dimensionality of their input, such as `max` or `sum`, will create distributed objects if the resulting object is larger than a scalar (1x1 matrix). Routines that return a fixed number of values, independent of the size of the input (like `eigs`, `svds`, and `histc`) will return local MATLAB (non-distributed) objects even if the input is distributed. Operators whose returns are bigger than the size of the input (e.g. `kron`) will return distributed objects if any of their inputs are distributed. Note that indexing, whether for a reference or an assignment, is just another operator, and follows the same rules.

The following example creates a distributed object through the propagation of a distributed object. In this case, since **`App`** is created as a distributed object through the **`*p`** syntax, **`Bpp`** will be created as distributed.

```
>> App = ones(100*p);
>> Bpp = 2 * App;
>> ppwhos
Your variables are:
  Name        Size           Bytes       Class
  App         100x100p       80000       ddense array
  Bpp         100x100p       80000       ddense array
Grand total is 20000 elements using 160000 bytes
MATLAB has a total of 0 elements using 0 bytes
Star-P® server has a total of 20000 elements using 160000 bytes
```

Note that in this example, both `ones` and “*” are overloaded operations and will perform the same function whether the objects they operate on are local or distributed.

The following computes the eigenvalues of **`Xpp`**, and stores the result in a matrix **`Epp`**, which resides on the server.

```
>> Xpp = rand(1000*p);
>> Epp = eig(Xpp);
```

The result is not returned to the client, unless explicitly requested, in order to reduce data traffic.

Operators which reduce the dimensionality of their input naturally transition between distributed and local arrays, in many cases allowing an existing MATLAB script to be reused with Star-P® having little or no change. Putting together all of these concepts in a single example, you can see how distribution propagates depending on the size of the output of an operator. (Note that the example omits trailing semicolons for operators that create distributed objects so their size will be apparent.)

Assume that the script `propagate.m` consists of the following commands:

```
>> type propagate
[rows, cols] = size(a)
b = rand(rows,cols)
c = b+a
d = b*a
e = b.*a
f = max(e)
ff = max(max(e))
gg = sum(sum(e))
size(ff), size(gg)
h = fft(e)
i = ifft(h)
[i j v] = find(b > 0.95)
q = sparse(i, j, v, rows, cols)
r = q' + speye(rows);
s = svd(d);
t = svds(d,4);
ee = eig(d);
```

In that case, distribution will propagate through its operations as follows (note that we are omitting the use of a suffix **pp** variable notation here, since the script is being reused without modification):

```
>> a = ones(1000*p,1000)
a =
        ddense object: 1000p-by-1000
% now executing the commands in script 'propagate'
>> [rows, cols] = size(a)
rows =
     1000p
cols =
        1000
>> b = rand(rows,cols)
b =
        ddense object: 1000p-by-1000
>> c = b+a
c =
        ddense object: 1000p-by-1000
>> d = b*a
d =
        ddense object: 1000p-by-1000
>> e = b.*a
e =
        ddense object: 1000p-by-1000
>> f = max(e)
f =
        ddense object: 1-by-1000p
>> ff = max(max(e))
ff =
    1.0000
>> gg = sum(sum(e))
gg =
```

```
   4.9991e+05
>> size(ff), size(gg)
ans =
     1     1
ans =
     1     1
>> h = fft(e)
h =
        ddense object: 1000p-by-1000
>> i = ifft(h)
i =
        ddense object: 1000p-by-1000
>> [i j k] = find(b > 0.95)
i =
        ddense object: 49977p-by-1
j =
        ddense object: 49977p-by-1
k =
        ddense object: 49977p-by-1
>> q = sparse(i, j, k, rows, cols)
q =
        dsparse object: 1000p-by-1000
>> r = q' + speye(rows);
>> s = svd(d);
>> t = svds(d,4);
>> ee = eig(d);
% end of 'propagate' script, back to main session
>> ppwhos
Your variables are:
  Name        Size             Bytes        Class
  a           1000px1000       8000000      ddense array
  ans         1x2              16           double array
  b           1000px1000       8000000      ddense array
  c           1000px1000       8000000      ddense array
  cols        1x1              8            double array
  d           1000px1000       8000000      ddense array
  e           1000px1000       8000000      ddense array
  ee          1000px1          16000        ddense array  (complex)
  f           1x1000p          8000         ddense array
  ff          1x1              8            double array
  gg          1x1              8            double array
  i           49977px1         399816       ddense array
  j           49977px1         399816       ddense array
  k           49977px1         399816       ddense array
  q           1000px1000       807696       dsparse array (sparse)
  r           1000px1000       822688       dsparse array (sparse)
  rows        1x1              258          dlayout array
  s           1000px1          8000         ddense array
  t           4x1              32           double array

Grand total is 5253832 elements using 42862162 bytes
MATLAB has a total of 10 elements using 330 bytes
```

```
Star-P® server has a total of 5253822 elements using 42861832 bytes
```

As long as the size of resulting arrays are dependent on the size of an input array and hence will likely be used in further parallel computations, the output arrays are created as distributed objects. When the output is small and likely to be used in local operations in the MATLAB front-end, it is created as a local object. For this example, with two exceptions, all of the outputs have been created as distributed objects. The exceptions are `rows`, which is a scalar of class `dlayout`, and `t`, whose size is based on the size of a value passed to `svds`. Even in cases where dimensionality is reduced, as with `find`, when the resulting object is large, it is created as distributed.

Propagation of Distribution

A natural question often asked is, “What is the distribution of the output of a given function expressed in terms of the inputs?” In Star-P®, there is a general principle on distribution that has been carefully implemented in the case of indexing and for a large class of functions. Perhaps like irregular verbs of a natural language, there are also a number of special cases, that do not follow these rules, some of which we list here.

In Star-P®, the output of an operation does not depend on the distribution of its inputs. The rules specifying the exact distribution of the output may vary in future releases of Star-P®.

Note: Performance and floating point accuracy may be affected, see "Accuracy of Star-P® Routines" for more information.

Let’s first recall the distributions available for data in Star-P®:

Type	Distribution
`ddense`	row, column
`ddensend`	linear distribution along any dimension
`dsparse`	row distribution only

The distributions of the output of operations follow the “calculus of distribution”. To calculate the expected distribution of the output of a given function, express the size of the output in terms of the size of the inputs. Note that matrices and multidimensional arrays are never distributed along singleton dimensions (dimensions with a size of one), unless explicitly created that way.

Functions of One Argument

In the simplest case, for functions of one argument where the size of the output is the size of the input, the output distribution matches that of the input.

Examples for Functions with One Argument

Operations with one input:

The cosine function operates on each element, and the output retains the same distribution as the input:

```
>> App = rand(1000*p, 4)
App =
        ddense object: 1000p-by-4
>> Bpp = cos(App)
Bpp =
        ddense object: 1000p-by-4
```

A conjugate transpose exchanges the dimension sizes of its input, so it also exchanges the dimensions' distribution attributes:

```
>> App = rand(1000*p, 4)
App =
        ddense object: 1000p-by-4
>> Bpp = App'
Bpp =
        ddense object: 4-by-1000p
```

Other example single argument functions:

```
App.^2, lu(App), fft(App), fft2(App)  (ddense and ddensend where applicable)
```

Exceptions:

Certain Linear Algebra functions such as `qr`, `svd`, `eig` and `schur` benefit from a different approach and do not follow this rule. See "Single ddense arguments" below.

Functions of Multiple Arguments

For functions with multiple input arguments, we again express the size of the output in terms of the size of the inputs. When the calculation provides an ambiguous result, the output will be distributed in the rightmost dimension that has a size greater than one.

For operations in which the output size is the same as both inputs, such as element-wise operations (`App+Bpp`, `App.*Bpp`, `App./Bpp`, etc), we consider the distribution of both inputs. If both inputs are row distributed, then the output will be row distributed. If the combination of inputs has more than one distributed dimension, then the default of distributing on the rightmost dimension applies.

Table 3-1 Rules for Propagation of Distribution

operations	Distribution of App	Distribution of Bpp	Output Distribution
.^, lu, fft, fft2	row or column	N/A	matches input
+, .*, ./	row	row	row
	column	column	column
	row	column	column
	column	row	column

For example, with matrix multiplication,

```
Cpp = App * Bpp
size(Cpp) == [size(App,1) size(Bpp,2)] : rows_of_App -by- cols_of_Bpp
```

For `Cpp=App*Bpp`, if `App` and `Bpp` are both row distributed, the output will have its first dimension distributed as a result of the fact that `App` has its first dimension distributed. Its second dimension will not be distributed since `Bpp`'s second dimension is not distributed. Therefore `Cpp` will be row distributed as well.

For `Cpp=App*Bpp`, if `App` and `Bpp` are both column distributed, similar logic forces the output to be column distributed.

For `Cpp=App*Bpp`, if `App` is row distributed and `Bpp` is column distributed, the calculus of distribution indicates that both dimensions of the output should be distributed. Since this is not permissible, the rightmost dimension is distributed, resulting in a column distribution.

For `Cpp=App*Bpp`, If `App` is column distributed and `Bpp` is row distributed, the calculus of distribution indicates that neither dimension of the output should be distributed. Once again, we fall back on the default of distributing the rightmost (column) dimension.

Examples for Functions with Multiple Arguments

As a less trivial example, consider `Cpp = kron(App,Bpp)`. The size of the dimensions of `Cpp` are calculated through the following formula:

```
size(Cpp) = size(App) * size(Bpp)
```

The resulting distribution would be ambiguous, so it defaults to the standard of distributing the rightmost dimension:

```
>> App = rand(1000*p, 4);
>> Bpp = rand(10, 100*p);
>> Cpp = kron(App,Bpp)
```

```
Cpp =
        ddense object: 10000-by-400p
```

As another example, consider transpose:

```
Cpp = App.'
size(Cpp) = [size(App,2) size(App,1)]
```

For transpose, if `App` is row distributed, the output will be column distributed. If `App` is column distributed, the output will be row distributed.

Exceptions for Multiple Arguments

The following operations benefit from special-case rules and must be accounted for one by one. The following list is only the non-trivial cases.

Table 3-2 Single ddense arguments

	1 output	2 outputs	3 outputs
qr(ddense) or qr(ddense,0)	matches input	matches input	column
svd(ddense) or svd(ddense,0) or svd(ddense,’econ’)	row	matches input	matches input
eig(ddense) (no-sym)	row	matches input (not officially supported)	
eig(ddense) (sym)	row	matches input	

Table 3-3 Distribution output of kron

Operation	Distribution of App	Distribution of Bpp	Output Distribution
`kron(App,Bpp)`	row	row	row
`kron(App,Bpp)`	row	column	row
`kron(App,Bpp)`	column	column	column
`kron(App,Bpp)`	column	row	column

Indexing Operations

Indexing operations follow the same style of rules as other operations. Since the output size depends on the size of the indices (as opposed to the size of the array being indexed), the output distribution will depend on the distribution of the arguments being used to index into the array. If all objects being used to index into the array are front-end objects, then the result will default to distribution along the rightmost dimension.

Some indexing examples:

For `[r c] = size(Bpp); Bpp = reshape(App,r,c);`, so we have:

```
>> App = rand(9, 4*p);
>> Bpp = reshape(App,  6, 6*p)
Bpp =
        ddense object: 6-by-6p
>> Cpp = reshape(Bpp, 36,   1)
Cpp =
        ddense object: 36p-by-1
```

Indexing is a particularly tricky example, because `subsref` has many different forms. `Bpp = App(:,:)` has the same distribution as `App`, because `size(Bpp) == size(App)`. `Bpp` = `App`(:) vectorizes (linearizes) the elements of `App`, so the output will be row or column distributed accordingly.

Other linear indexing forms inherit the output distribution from the indexing array:

```
>> App = rand(10*p,10)
App =
        ddense object: 10p-by-10
>> Ipp = ppback(magic(10))
Ipp =
        ddense object: 10-by-10p
>> Bpp = App(Ipp)
Bpp =
        ddense object: 10-by-10p
```

But, consider `Bpp = App(Rpp,C)` with the following:

```
>> App = rand(100*p,100);
>> Rpp = randperm(100*p)
Rpp =
        ddense object: 1-by-100p
>> C = randperm(100);
>> Bpp = App(Rpp,C)
Bpp =
        ddense object: 100p-by-100
```

Here, `size(Bpp)` is defined as `[prod([1 100p]) prod([1 100])] [100p 100]` which simplifies to `[prod([10p 10p]) prod([10 10])]` and then `[100p 100]`. So the distribution of

- `Bpp`'s row dimension is inherited from both of `Rpp`'s dimensions, and
- `Bpp`'s column dimension is inherited from C.

As a final example, consider logical indexing:

```
>> App = rand(100*p,100);
>> Ipp = App > 0.5
Ipp =
        ddense object: 100p-by-100
>> Bpp = App(Ipp)
Bpp =
        ddense object: 5042p-by-1
```

This might be unexpected, but is so because `App(Ipp)` is essentially the same as `App(find(Ipp))`, and `find(Ipp)` returns a row-distributed column vector.

Summary for Propagation of Distribution

To summarize:

- Output distributions follow the "calculus of distribution" in which the rules for determining the size of the output define the rules for the distribution of the output, though a selection of Linear Algebra functions do not follow these rules.
- Typically, functions with one input and one output will have outputs that match the distribution of the input.
- When the output distribution will be ambiguous or undefined by the standard rules, the output will be distributed along its rightmost dimension.
- Outputs are never distributed along singleton dimensions (dimensions with a size of one).

Explicit Data Movement with `ppback` and `ppfront`

In some instances a user wants to move data explicitly between the client and the server. The `ppback` command and its inverse, `ppfront`, do these functions.

```
>> n = 1000;
>> mA = rand(n);
>> mB = rand(n);
>> ppwhos
Your variables are:
  Name        Size           Bytes        Class
  mA          1000x1000      8000000      double array
```

```
  mB         1000x1000      8000000      double array
  n          1x1            8            double array
Grand total is 2000001 elements using 16000008 bytes
MATLAB has a total of 2000001 elements using 16000008 bytes
Star-P® server has a total of 0 elements using 0 bytes

>> App = ppback(mA)
App =
        ddense object: 1000-by-1000p
>> ppwhos
Your variables are:
  Name       Size            Bytes         Class
  App        1000x1000p      8000000       ddense array
  mA         1000x1000       8000000       double array
  mB         1000x1000       8000000       double array
  n          1x1             8             double array
Grand total is 3000001 elements using 24000008 bytes
MATLAB has a total of 2000001 elements using 16000008 bytes
Star-P® server has a total of 1000000 elements using 8000000 bytes
```

`ppfront` is the inverse operation, and is in fact the only interface for moving data back to the front end system. This conforms to the principle that once you, the programmer, have declared data to be distributed, it should stay distributed unless you explicitly want it back on the front end. Early experience showed that some implicit forms of moving data back to the front end were subtle enough that users sometimes moved much more data than they intended and introduced correctness (due to memory size) or performance problems.

Note that the memory size of the client system running MATLAB, compared to the parallel server, will usually prevent full-scale distributed arrays from being transferred back to the client.

```
>> App = rand(1700,1700*p)
App =
        ddense object: 1700-by-1700p
>> ppwhos

Your variables are:
  Name       Size            Bytes          Class
  App        1700x1700p      23120000       ddense array
Grand total is 2890000 elements using 23120000 bytes
MATLAB has a total of 0 elements using 0 bytes
Star-P® server has a total of 2890000 elements using 23120000 bytes
>> b = ppfront(App);
>> ppwhos

Your variables are:
  Name       Size            Bytes          Class
  App        1700x1700p      23120000       ddense array
  b          1700x1700       23120000       double array
Grand total is 5780000 elements using 46240000 bytes
MATLAB has a total of 2890000 elements using 23120000 bytes
```

```
Star-P® server has a total of 2890000 elements using 23120000 bytes
```

Note: `<starp_root>` is the path where Star-P® is located.

The `ppback` and `ppfront` commands will emit a warning message if the array already resides on the destination (parallel server or client, respectively), so you will know if the movement is superfluous or if the array is not where you think it is.

These two commands, as well as the `ppchangedist` command, will also emit a warning message if the array being moved is bigger than a threshold data size (default size being 100MB). The messages can be disabled, or the threshold changed, by use of the **`ppsetoption`** command, documented in "Star-P® Functions".

Loading And Saving Data on the Parallel Server

Just as the `load` command reads data from a file into MATLAB variable(s), the `ppload` command reads data from a file into distributed Star-P® variable(s). Assume that you have a file created from a prior MATLAB or Star-P® run, called `imagedata.mat`, with variables **`App`** and **`Bpp`** in it. (MATLAB or Star-P® appends the `.mat` suffix.) You can then read that data into a distributed object in Star-P® as follows:

```
>> ppload imagedata App Bpp
>> ppwhos
Your variables are:
  Name       Size            Bytes        Class
  App        1000x1000p      8000000      ddense array
  Bpp        1000x1000p      8000000      ddense array
Grand total is 2000000 elements using 16000000 bytes
MATLAB has a total of 0 elements using 0 bytes
Star-P® server has a total of 2000000 elements using 16000000 bytes
```

While in some circumstances `ppload` can be replaced by a combination of `load` and `ppback`, in general distributed arrays in Star-P® will be larger than the memory of the client system running MATLAB, so it will be preferable to use `ppload`. For the same reason, users will probably want to use `ppsave` of distributed arrays rather than `ppfront/save`.

Note that the file to be loaded from must be available in a filesystem visible from the HPC server system, not just from the client system on which MATLAB itself is executing. Consequently, if your .mat file is initially located on your client system, then copy the file into a working directory on your server.

The `ppload` and `ppsave` Star-P® Commands

The distributed I/O commands `ppload` and `ppsave` store distributed matrices in the same uncompressed Level 5 `.mat`-File Format used by MATLAB.

Information about which dimension(s) of an array were distributed are not saved with the array, so `ddense` matrices retrieved by `ppload` will, by default, be distributed on the last dimension.

Note: The use of `*p` to make objects distributed and thereby make operators parallel can almost always be made backwards compatible with MATLAB by setting `p = 1`. The use of `ppload` does not have the same backward compatibility.

If you use `ppsave` to store distributed matrices into a file, you can later use `load` to retrieve the objects into the MATLAB client. Distributed matrices (`ddense` and `dsparse`) will be converted to local matrices (full and sparse), as if `ppfront` had been invoked on them. (The exception to this operation is that some very large matrices break `.mat`-File compatibility; if `ppsave` is applied to a distributed matrix with more than 2^{32} rows or columns, or `ppwhos` data requires more than 2^{31} bytes of storage, then `load` may not be able to read the file.)

To move data from the front-end to the back-end via a file, the MATLAB `save` command must use the `-v6` format, as in `save('foo','w','-v6')` for saving variable `w` in file `foo`. Then you can use `ppload` to read the resulting file to the server. This will convert local matrices to global matrices, just as if `ppback` had been invoked, except that the resulting matrices will be distributed only on the last dimension.

Star-P®'s `ppload` command cannot yet read the older Level 4 `.mat`-File, nor the compressed Level 5 format. Use the `-v6` flag in the MATLAB client to convert such files to uncompressed Level 5 format.

The `ppfopen` Star-P® Command

Another method of loading of data is through the use of the `ppfopen` command. By calling `ppfopen` with only a single string argument specifying a target file to open, the contents of the file are opened in a read-only mode. The following command opens `your_file` and returns a distributed file identifier of class `@dfid`.

```
fid = ppfopen('your_file');
```

Using a second input argument to `ppfopen`, further permissions for handling the contents of the target file can be specified.

```
fid = ppfopen('your_file',MODE);
```

The input MODE can take values that allow for various permissions for viewing or altering the file's contents.

MODE	Permission
'rb'	read
'wb'	write (create if necessary)
'ab'	append (create if necessary)

MODE	Permission
'rb+'	read and write (do not create)
'wb+'	truncate or create for read and write
'ab+'	read and append (create if necessary)

Note: Only native machine format is supported and the ppfopen interface will return an error if the caller tries to specify a different machine format or encoding parameter.

fopen, fread, fwrite, frewind, and fclose

The functions `fopen`, `fread`, `fwrite`, `frewind`, and `fclose` have been overloaded to work with distributed data, including distributed file identifiers.

For example, the `fread` function can be used in the following manner to assign a 1000 by 1000 matrix to a variable that has previously been associated with the distributed file identifier `fid`:

```
App = fread(fid,[1000 1000*p]);
```

You will notice that fread allows you to specify the distribution properties of the data assigned to the distributed variable **`App`**.

HDF5, Hierarchical Data Format Version 5

Star-P® supports import and export of datasets in the Hierarchical Data Format, Version 5 (HDF5). The HDF5 format

- is widely used in the high-performance computing community,
- is portable across platforms,
- provides built-in support for storing large scientific datasets (larger than 2GB) and
- permits lossless compression of data.

For more information about the HDF5 file format, please visit http://hdf.ncsa.uiuc.edu/HDF5.

The Star-P® interface to the HDF5 file format currently supports the import and export of distributed dense and sparse matrices with double precision and complex double precision elements. In addition, a utility function is provided to list meta-data information about all variables stored in a HDF5 file.

The next few sub-sections discuss the syntax of the individual HDF5 commands in more detail.

Writing variables to an HDF5 file

Distributed variables are written to a remote HDF5 file using the `pph5write` command. This command takes a filename, and a list of pairs consisting of a distributed variable and its corresponding fully-qualified dataset name within the HDF5 file. If the file already exists, an optional string argument can be passed to the command: `'clobber'` causes the file to be overwritten and `'append'` causes the variables to be appended to the file. The default mode is `'clobber'`. If the write mode is `'append'` and a variable already exists in the location specified, it is replaced.

Example 1

To write the distributed variables `matrix_a` to the dataset `/my_matrices/a and matrix_b` to the dataset `/my_matrices/workspaces/temp/matrix_b` to the HDF5 file `temp.h5` in the `/tmp` directory of the HPC server, you would use:

```
>> matrix_a = rand(1000*p);
>> matrix_b = sprand(1200*p,1200,0.05);
>>
pph5write('/tmp/temp.h5',matrix_a,'/my_matrices/a',matrix_b,'/my_matrices/work
space/tmp/b');
```

Example 2

To append a distributed variable **`matrix_c`** to the HDF5 file created in the previous example to the location `/my_matrices/workspace2/temp/matrix_c`, one would use:

```
>> matrix_c = rand(500*p);
>> pph5write('/tmp/temp.h5','append',matrix_c,'/my_matrices/workspace/tmp/c');
```

Reading variables from an HDF5 file

Datasets in a HDF5 file can be read into distributed variables using the pph5read command. It takes a file name and a list of fully-qualified dataset names to read.

Example 3

To read the dataset, `/my_matrices/workspaces/temp/matrix_b` into a distributed variable, **`matrix_d`**, from the file created in the first example, one would use:

```
>> matrix_d = pph5read('/tmp/temp.h5', '/my_matrices/workspace/tmp/b');
>> ppwhos
Your variables are:
  Name         Size             Bytes        Class
  matrix_d     1200px1200       1134528      dsparse array (sparse)
Grand total is 70304 elements using 1134528 bytes
MATLAB has a total of 0 elements using 0 bytes
Star-P® server has a total of 70304 elements using 1134528 bytes
```

Querying variables stored inside an HDF5 file

It is possible to obtain a list of variables stored in an HDF5 file and their associated types using the `pph5whos` command that takes in the name of the HDF5 file as its sole argument. With a single output argument, the command returns a structure array containing the variable name, dimensions and type information. With no output arguments, the command simply prints the output on the MATLAB console.

Example 4

Running `pph5whos` on the file after running Examples 1 and 2, the following is obtained:

```
>> pph5whos('/tmp/temp.h5')
```

Table 3-4

Name	Size	Bytes	Class
/my_matrices/a	1000x1000	8000000	double array
/my_matrices/workspace/tmp/b	1200x1200[70304 nnz]	562432	double array (sparse)
/my_matrices/workspace2/tmp/c	500x500	2000000	double array

Representation of data in the HDF5 file

This section describes the internal representation of HDF5 files used by the functions described previously. If the HDF5 file to be read is not generated using `pph5write`, it is important to read the following subsections carefully.

Multidimensional arrays

Distributed matrices are stored in column-major (or Fortran) ordering. Therefore, `pph5write` follows the same strategy used by Fortran programs that import or export data in the HDF5 format: multidimensional matrices are written to disk in the same order in which they are stored in memory, except that the dimensions are reversed. This implies that HDF5 files generated from a C program will have their dimensions permuted when read back in using `pph5read`, but the dimensions will not be permuted if the HDF5 file was generated either using a Fortran program or `pph5write`. In the former case, the data must be manually permuted using `ctranspose` for two-dimensional and permute for multidimensional matrices.

Complex data

An array of complex numbers is stored in the interleaved format consisting of a pairs of HDF5 native double-precision numbers representing the real and imaginary components.

Sparse matrices

A sparse matrix is stored in its own group, consisting of three attributes (a sparsity flag, `IS_SPARSE`, the number of rows, ROWS and the number of columns, COLS) and three datasets (`row_indices`, `col_indices` and `nonzero_vals`) containing the matrix data stored in the triplet form. All attributes and datasets are stored as double precision numbers, except `IS_SPARSE` which is stored as an integer and `nonzero_vals` which can either be double or double complex.

Limitations

The HDF5 interface in Star-P® currently has the following limitations:

1. Import and export of variables is restricted to types that can be represented in the Star-P® server. Currently, this is restricted to double and complex double elements.
2. It is not possible to import or export strings, structure arrays or cells.
3. It is not possible to attach attributes to datasets or groups.
4. Each dataset must be imported or exported explicitly; support for accessing files using wild cards or regular expressions is not yet supported.
5. Only the HDF5 file format is supported. Data files conforming to earlier versions of HDF or raw text files must be first converted to the HDF5 format.

Differences from MATLAB HDF5 support

The HDF5 import-export features in Star-P® currently differ from that provided in MATLAB in the following respects:

1. Permutations of dimensions for multidimensional arrays. MATLAB only permutes the first two dimensions even for multidimensional arrays; the permutation in Star-P® is consistent with that used for other Fortran programs
2. Handling of complex matrices. MATLAB does not support saving of complex matrices natively.
3. Handling of sparse matrices. MATLAB does not support saving of sparse matrices natively.
4. Handling of hdf5 objects. Star-P® currently does not support the loading and saving of datasets described using instances of the hdf5 class supported by MATLAB.
5. Direct access to the HDF5 library. Unlike MATLAB Star-P® does not provide direct access to the HDF5 library; all access must happen through the `pph5write`, `pph5read` and `pph5whos` commands.

Converting data from other formats to HDF5

1. Download and build the HDF5 library, version 1.6.5 library. The source files can be downloaded from http://hdf.ncsa.uiuc.edu/HDF5/release/obtain5.html.
2. Download and build the Steven Johnson's H5utils package available at http://ab-initio.mit.edu/wiki/index.php/H5utils.
3. The tool `h5fromtxt` can be used to convert a text file into the HDF5 format and the tool `h5fromh4` can be used to convert a data file in earlier HDF formats into HDF5.
4. Once converted, the resulting data files can be directly read in using the `pph5read` command.

INTERACTIVE supercomputing

Chapter 4

Task Parallelism with Star-P® and MATLAB

In the previous chapter, the operators used on distributed arrays operated on the entire array(s) in a fine-grained parallel approach. While this operation is easy to understand and easy to implement (in terms of changing only a few lines of code), there are other types of parallelism that don't fit this model. The `ppeval` function allows for coarse-grained parallel computation, otherwise known as MIMD (multiple instruction multiple data) or task parallelism, where operations are conducted on blocks, coarse-grains, of the data. This coarse-grained computation is distributed uniformly over the number of parallel processors. This mode of computation allows non-uniform parallelism to be expressed (e.g., the `sum` operator could be used on odd columns and the `max` operator on even columns).

This chapter contains information on performing operations in task parallel and includes the following:

- "The ppeval Function: The Mechanism for Task Parallelism"
- "Star-P® Naming Conventions"
- "Transforming a for Loop into a ppeval Call"
- "ppeval Syntax and Behavior"
- "ppevalsplit"
- "Choosing Your Task Parallel Engine (TPE)"
- "Per Process Execution"
- "Calling Non-”M” Functions from within ppeval"
- "Workarounds and Additional Information"

The ppeval Function: The Mechanism for Task Parallelism

`ppeval` allows you to execute built-in functions and user-defined functions in parallel on a High Performance Computer. `ppeval` handles the distribution of data and code over the processors in the HPC, as well as the execution and the gathering of computational results.

To define some of relevant terminology for `ppeval`, let's look again at the example from "Extending MATLAB with Star-P®".

```
Xpp = rand(1000,1000,100*p);
Ypp = ppeval('inv',Xpp);
```

In this example, the ppeval call splits up the variable `Xpp` into 100 individual slices (by default splitting is done along the last dimension). The slices are then divided over the available processors; so in the case of 100 slices and 10 processors, each processor would receive 10 slices. Each processor iterates over the slices it receive and applies the function `'inv'` to each of the slices. When each processor completed its job, the results of all processors are combined, preserving order, and returned as the output value.

You can view `ppeval` as a parallel loop. You cannot assume anything about the order in which the iterations occur or the processor(s) on which they occur. Since the computations of the individual iterations are performed in complete isolation of all the other iterations, `ppeval` requires that the computation being performed is independent over the iterations. Consequently, functions that contain recursive relations or that update variables based on sequentially previous iterations inside the function body are not applicable for task-parallel execution.

Any function passed to `ppeval` must be either a built-in MATLAB[1] function or a user-supplied MATLAB function in a `.m` file; a function named `foo` for example. As with any function called from MATLAB (or Star-P®), the function must exist in a file of the name `foo.m` located in one of the directories visible to the MATLAB directory search path on the system where the Star-P® client is running. As well as the particular function itself, files containing any functions that are called by `foo.m`, down to the level of the built-in operators, must be accessible in directories in the MATLAB search path. All of these identified files will be transferred to the HPC server for execution. For a discussion on calling a non-MATLAB function via the MATLAB system function, see "Calling non-MATLAB functions within a ppeval."

1. A subset of the MATLAB operators are supported. While you might want to extend this set with routines that are part of MATLAB or one of its toolboxes, The MathWorks software license prohibits this for the way the `ppeval` is implemented. To comply with this prohibition, `ppeval` will not move to the HPC server any routines that are generated by The MathWorks.

Star-P® Naming Conventions

Star-P® commands and data types generally use the following conventions, to distinguish them from standard MATLAB commands and data types:

- Most Star-P® commands begin with the letters **pp**, to indicate parallel. For example, the Star-P® `ppload` command loads a distributed matrix from local files. Exceptions to this rule include the split and bcast commands.
- Star-P® data types begin with the letter `d`, to indicate "distributed". For example, the Star-P® `dsparse` class implements distributed sparse matrices.

The following convention for displaying Star-P® related commands and classes is used throughout this chapter.

Command/Variable	Font
p & other dlayout variables	**bold green font**
Distributed variables	**bold blue font**
Star-P® functions	**bold black font**

Transforming a `for` Loop into a `ppeval` Call

The typical work-flow of introducing `ppeval` into a code that is currently serial takes the following steps:

1. Identify a `for` loop that is embarrassingly parallel.
2. Determine the input and output variables of the `for` loop.
3. Transform the body of the `for` loop into a function.
4. Call your newly defined function with `ppeval` using the correct input and output variables.

Here we will walk through an example of these steps. Below we will discuss ways in which the user can control the splitting and broadcasting behavior of the input variables to `ppeval`.

Step 1: Identify a for loop that is embarrassingly parallel.

```
x = rand(n,n,m);
y = rand(n,m);
z = zeros(n,m);

for i = 1:m
    [v d] = eig(x(:,:,i));
```

```
    a = v*y(:,i) + diag(d);
    z(:,i) = x(:,:,i)\a;
end
```

The `for` loop in this example is indeed embarrassingly parallel since it contains no recurrent relations and/or variable updates. In principle, if we had `m` computers, then we could compute one iteration of the `for` loop on every computer and obtain the correct result after recombining them.

Step 2: Determine the input and output variable of the loop

The input variables to the loop are `x` and `y` and the output variable is `z`. The variables `v`, `d`, and `a` are variables whose scope is limited to the `for` loop.

Step 3: Transform the body of the for loop into a function

The function `foo1`, defined below, contains the `for` loop body content with output variable `z` and input variables `x` and `y`.

```
function z = foo1(x,y)

[v d] = eig(x);
a = v*y + diag(d);
z = x\a;
```

Note that we removed all of the indexing operations that are present in the `for` loop body in Step 1. Since the `ppeval` process splits the variables `x` and `y` into individual slices along the last dimension (by default), `ppeval` does the indexing operations for you in the process of dividing of the input data and gathering the output data.

Step 4: Call function defined in Step 3 with ppeval

Now that we have the defined our function `foo1` and we know the input and the output arguments, we can perform the `ppeval` call:

```
X = rand(n,n,m*p);
Y = rand(n,m*p);
Zpp = ppeval('foo1',X,Y);
```

This completes the transformation of the serial `for` loop to a task-parallel execution of the same code with Star-P®.

Note: `X` and `Y` do not necessarily have to be distributed objects. See "Splitting" and "Broadcasting" for more details.

Note: It might seem natural that you could transform a `for` loop to run in parallel just by adding a `*p` to the loop bounds. Unfortunately, this does not have the desired effect. The simplicity of this approach has not been lost on the Star-P® developers, and some support for this method may appear in a future release.

ppeval Syntax and Behavior

This section covers the following topics:

- "ppeval Syntax Grammar"
- "Requirements of Functions Passed to ppeval"
- "Input Arguments"
- "Output Arguments"
- "Examples of ppeval Usage"
- "Star-P® M TPE"
- "Star-P® Octave Engine"
- "C/C++ Engine for Running Compiled C/C++ Package Functions"
- "String Arrays"
- "Splitting on a Scalar"
- "Global Variables"

ppeval Syntax Grammar

The syntax of `ppeval` is similar to that of `eval` or `feval`.

```
[o1 o2 ... oN] = ppeval('foo', In1, In2, ..., InN);
```

`foo` is the name of the function you would like to execute in task-parallel. `In1`, `In2`, ... are the input arguments to `func` and `o1`, `o2`, ... are the output arguments to `foo`. The supported input argument types are: strings, function handles, scalars, arrays, and matrices (see workaround section below for input arguments of type string-array and struct-array) and the supported output arguments are scalars, arrays, and matrices.

Requirements of Functions Passed to `ppeval`

`ppeval` puts a few requirements on a function `foo`:

- The dimensions of all input arguments that are split up over the processors all need to have the same size.
- At least one of the input arguments needs to be split up over the processors, or in other words, at least one of the input arguments needs to be an array or matrix.
- The function `foo` can have a maximum of 58 input arguments.
- The function `foo` must return at least 1 output argument.
- The output arguments of the function need to have the same size for each iteration of the function `foo` (see Output Arguments)
- `foo` must be an actual MATLAB function file, it cannot be a script file.
- The function `foo` cannot contain nested functions.

Input Arguments

The user has complete control over the splitting and broadcasting of input variables with the `split/ppsplit` and `bcast/ppbcast` commands. These commands can only be used in conjunction with the `ppeval` command.

Default Behavior

By default all scalars, strings, and function handles are broadcast to every processor on the HPC. Every processor receives an identical copy. Arrays and matrices are split up into slices along the last dimension and divided over the processors. The default behavior of splitting and broadcasting input arguments can be overridden by the user.

Splitting

To split an array or matrix in a dimension other than the last dimension, use the `split` command in conjunction with `ppeval`. The syntax of the `split/ppsplit` commands are

```
split(A,DIM)
ppsplit(A,DIM)
```

where `A` is the input argument and `DIM` is the dimension along which you want to split the variable `A`. The possible arguments to `split` and `ppsplit` are:

```
split(A,DIM) or ppsplit(A,DIM): Split variable A along dimension DIM
split(A,0)   or ppsplit(A,0)  : Split over individual elements of variable A
split(A)     or ppsplit(A): Split variable A along final dimension (default)
```

As stated above the `split` and `ppsplit` command can only be used in conjunction with `ppeval`. For example:

```
Xpp = rand(100*p,1000,1000);
Ypp = ppeval('inv',split(Xpp,1));
```

performs 100 matrix inversions. Each inversion is performed on a matrix 1000-by-1000 in size, because the variable `Xpp` is split along the first dimension.

Broadcasting

To broadcast an array or matrix to every processor, include the argument in a `bcast` command. You would use the `bcast` command if you want to send an array or matrix variable in its entirety to every processor. As with `ppsplit`, the `ppbcast` command needs to be used in conjunction with the `ppeval` command. For example, to add a 2D matrix, `Xpp`, to every slice of the 3D array `Ypp`, then issue the following command:

```
Xpp = rand(n*p);
Ypp = rand(n,n,m*p);
Zpp = ppeval('+',Ypp,bcast(Xpp));
```

This `ppeval` command is equivalent to the following `for` loop, apart from the fact it is performed in parallel as opposed to serial execution:

```
x = rand(n);
y = rand(n,n,m);
z = zeros(n,n,m);

for i = 1:m
    z(:,:,i) = y(:,:,i) + x;
end
```

Supported Input Argument Types

The supported input argument types are: strings and functions handles, as well as scalars, arrays, and matrices of type double and complex double. Scalars, arrays, and matrices of other types (for example single, ints, logical) are first converted to type double before being transferred to `ppeval`. By default, strings, function handles and scalars are broadcast.

Serial ppeval of Functions with Scalar Inputs

This section discusses how you can use `ppeval` in non-broadcast (serial mode) with a single scalar input argument.

Using Star-P® Octave TPE

Using `ppeval ('functionname',<scalar-value>)` with the Octave engine (`octave`) produces the error shown in the example.

Example

```
>> ppsetoption('TaskParallelEngine','octave')
```

```
>> ppeval('rand',1)
??? Error using ==> ppeval_octave at 208
At least one argument in a call to ppeval must be split
(either implicitly or explicitly)
```

The message indicates that at least one of the input arguments must be split. By default scalar input arguments are not split. They are broadcast. It is possible to split on a scalar by explicitly including the split command, such as:

```
>> ppeval('rand',split(1));
```

This results in one function evaluation of the function `rand` with the input argument 1.

Using Star-P® M TPE

Using `ppeval('myfunction',<scalar-value>)` with the Star-P® M (`starp_tpe`) task-parallel engine results in the execution of the function `myfunction`) on each Star-P® HPC server process with each using the input argument `<scalar-value>`.

Example

```
>> ppsetoption('TaskParallelEngine','starp_tpe')
>> ppeval('ones',1 )  % This example was run on a two processor install

ans =
    1x2 double
```

So, the execution of `ppeval` is effectively a per process evaluation of function `ones`. If you want to have just one function evaluation use the `split` function.

Example

```
>> ppsetoption('TaskParallelEngine','starp_tpe')
>> ppeval('rand',split(1))% This syntax works the same as w/ the octave engine.
ans =
    0.8147
```

Client vs. Server Variables

In the examples above, we used the `*p` construct to create the data that `ppeval` operates on. It is now a necessary requirement that ppeval operates on server variables only. In the case that `ppeval` receives a client variable, say a MATLAB variable, `ppeval` will first move the client variable to the server. Then the task parallel operation will be performed. Hence the result of operating on a client or server variable will be exactly the same. However, since the client variable must be moved from the client to the server, you will incur a performance penalty (moving large amounts of data of networks can be costly).

Distribution of input variables

The Star-P® server stores variables in a distributed matrix fashion. The information/memory contained by one variable is divided across the processors with each processor having access to part of the data. Star-P® supports several distributions. 2D matrices can be stored by rows or columns, where each processor has access to a single set of rows or columns respectively. ND arrays can be distributed only along one of the dimensions.

For the correctness of the ppeval execution, the dimensions of distributions for server variables are not important. However, the dimensions of distributions do have an effect on the performance characteristics of the `ppeval` execution. The best performance is achieved when the distributed dimension and the split dimension are the same; for example, splitting an input variable `Xpp`, defined by `Xpp = rand(10,10*p,10);`, as `push-pull(Xpp,2)`.

This superior performance occurs because all of the data is already distributed to the correct processors. As a counter-example, if an input variable is row-distributed (along the first dimension of a 2D matrix), and the `ppeval` splits the input along the columns (along the second dimension of a 2D matrix), then the first operation that must be performed is a distribution change of the input data. These operations do not come free, because they do cost communication time to perform. Consequently, the optimal performance of a `ppeval` operation occurs when all of the variables to be “split” are distributed along the same dimension as the dimension requested for the `split` or `ppsplit` operations.

Output Arguments

The supported output arguments to `ppeval` are scalars, arrays, and matrices of type double and complex-double. Scalars, arrays, and matrices of different types are converted to double before being handed from the task parallel engine to Star-P®. Additionally, each of the output arguments of the function called by `ppeval` need to have the same size for each iteration, or for each input slice for that function. For example, the following function `func2`, with input scalar variables `in1` and `in2`, always returns output variables of size 1-by-10, 3-by-5 and 13-by-1:

```
function [out1 out2 out3] = func2(in1, in2)

out1 = zeros(1,10);
out2 = zeros(3,5);
out3 = zeros(13,1);

out1(:) = in1 + in2;
out2(:) = in1 / in2;
out3(:) = in1 * in2;
```

As you can see, for every call to `func2`, the outputs will have exactly the same size. The requirement that the function called by `ppeval` returns arguments of the same size is important because of the way that `ppeval` returns the aggregate of task parallel computation. `ppeval` laminates the outputs for each iteration together along an additional dimension. The rules for laminating the output are the following:

1. If the output is a scalar, then laminate them in the column direction, and distribute by columns.
2. If the output is a row-vector, then laminate them in the row dimension, and distribute by rows.
3. If the output is a column-vector, then laminate them in the column dimension, and distribute by columns.
4. If the output is a 2D array or ND array, then laminate them in an additional dimension, and distribute along that dimension.

This means that if the size of the output argument of the function called by `ppeval` is `k-by-1` and the `ppeval` operation performs `r` iterations, then the output of `ppeval` is of size `k-by-1-by-rp`.

Examples of ppeval Usage

Let's first consider a simple example. Rather than using the built-in sum function on a `ddense` array, you could code it using `ppeval` and sum on a row or column.

```
>> n = 100
n =
   100
>> App = 1:n*p
app =
        ddense object: 1-by-100p
>> Bpp = repmat(App,n,1)
bpp =
        ddense object: 100-by-100p
>> ppfront(Bpp(1:6,1:6))
ans =
     1     2     3     4     5     6
     1     2     3     4     5     6
     1     2     3     4     5     6
     1     2     3     4     5     6
     1     2     3     4     5     6
     1     2     3     4     5     6
>> Cpp = ppeval('sum',Bpp)
Cpp =
        ddense object: 1-by-100p
>> ppfront(Cpp(1,1:6))
ans =
   100   200   300   400   500   600
>> Epp = ppeval('sum',ppsplit(Bpp,1))
Epp =
        ddense object: 1-by-100p
>> ppfront(Epp(1,1:6))
ans =
```

```
        5050        5050        5050        5050        5050        5050
```

- The first call in the previous ppeval example uses the default behavior to split its arguments along the last dimension (columns, in the case of 2D matrices).
- The variable b in the previous example did not need to be distributed, created with the `*p` construct, or transferred to the server using `ppback`, because `ppeval` automatically handled its distribution on the server.
- In the second call to `ppeval`, it was desired to split along rows, so the `ppsplit` function was used explicitly to obtain that result.

In this example, the function `'sum'` was called on each column of the input array. While useful for a simple example of functionality, you would not do this in practice because the sum operator on the whole array has the same behavior and is simpler to use. However, as shown in the next example, the function passed to `ppeval` does not have to perform the same computation for each input, and thus can be used to implement MIMD/task parallelism.

In this example, we will make use of the MATLAB function `quad`, which computes the definite integral of a function over an interval. The function being integrated could be highly nonlinear in its behavior, but `ppeval` supports that functionality.

If the file `func3.m` contains the following:

```
>> type func3
function b = func3(a)
b = (a^2-1) / (a+eps);
```

then `ppeval` can be called as:

```
>> n = 100;              % number of intervals
>> App = (0:(n-1)*p)/n  % lower bounds of intervals
app =
        ddense object: 1-by-100p
>> Bpp = (1:n*p)/n      % upper bounds of intervals
bpp =
        ddense object: 1-by-100p
>> Bpp = ppeval('quad',@func3,App,Bpp)
cpp =
        ddense object: 1-by-100p
>> ppfront(Cpp(1,1:6))
ans =
  -31.4384   -0.6930   -0.4052   -0.2873   -0.2227   -0.1818
```

This example also illustrates the use of a function handle (`@func3`).

ppevalsplit

In the case where the function returns a different size output for every iteration (see `func4` below) the `ppeval` procedure will fail since there is no logical way of laminating the output

values of the function. In this case, Star-P® provides the user with the ppevalsplit command, which returns the outputs of the individual iterations in a cell array to the client. Cell arrays are capable of holding variables of different sizes. Although the cell array returned from a ppevalsplit call is stored as a variable of type dcell on the Star-P® server, when indexing into a dcell array, the contents are automatically returned to the client. An example of a function that returns differently sized outputs and how to use it with ppevalsplit follows:

```
function out1 = func4(in1)
%
% func4 returns a random column vector of size in1-by-1
%
out1 = rand(in1,1);

end
```

Call the function func4 with ppevalsplit:

```
in = 1:10;
outpp = ppevalsplit('func4',in);
```

The output **outpp** will be of type dcell and contain:

```
outpp{1}  : 1-by-1 random vector
outpp{2}  : 2-by-1 random vector
...
outpp{10} : 10-by-1 random vector
```

Choosing Your Task Parallel Engine (TPE)

When performing task parallel operations in Star-P®, ppeval and ppevalsplit allow for you to make a choice as to the environment that will be utilized for performing task parallel operations.

In using ppeval or ppevalsplit, use one of the following as your task parallel engine:

- Star-P® M TPE
- Star-P® Octave Engine
- C/C++ Engine for Running Compiled C/C++ Package Functions

The Star-P® M TPE and Star-P® Octave TPE provide high performance computing compatible with MATLAB m-files.

Note! The Star-P® M TPE provides the fastest performance times for non-Altix users.

When choosing the option of using your own compiled C/C++ functions, packages must be loaded on to the server and used in accordance with the instructions specified in the "Star-P® Software Development Kit (SDK) Tutorial and Reference Guide".

Note! Choose an Octave task-parallel engine for task parallel applications that use sparse arrays or if you are running Star-P® on Altix. Star-P® TPE support for sparse task parallel operations will be implemented as a follow on to Star-P® Release 2.7.

Star-P® M TPE

The Star-P® M task-parallel engine (`starp_tpe`) is native to Star-P® and yields the best overall performance for non-Altix users when using `ppeval`. Select it by using the following call:

```
ppsetoption('TaskParallelEngine','starp_tpe')
```

Note! Choose an Octave task-parallel engine for task parallel applications that use sparse arrays or if you are running Star-P® on Altix. Star-P® TPE support for sparse task parallel operations will be implemented as a follow on to Star-P® Release 2.7.

Star-P® Octave Engine

You can choose Octave as the task parallel engine used when you call ppeval. If your task parallel codes require the use of MEX file functionality or include functionality that was not included in Octave 2.9.5, then you will want to set the task parallel engine to Octave 2.9.9 by calling:

```
ppsetoption('TaskParallelEngine','octave-2.9.9')
```

Within a given Star-P® session, only a single version of Octave can be set using `ppsetoption`.

For example, once there has been a call to `ppsetoption` using:

```
ppsetoption('TaskParallelEngine','octave-2.9.9')
```

you cannot switch to Octave 2.9.5 during a particular Star-P® session. By initially calling:

```
ppsetoption('TaskParallelEngine','octave-2.9.5')
```

then you are setting the Octave version to be 2.9.5 for the duration of a session. The first call to `ppsetoption('TaskParallelEngine','octave-2.9.x')` sets the available Octave engine for that particular Star-P® session.

C/C++ Engine for Running Compiled C/C++ Package Functions

In order to set the task parallel engine to use your own compiled C/C++ functions, `ppsetoption` can be configured in the following manner:

```
ppsetoption('TaskParallelEngine','C')
```

This allows you to write functions that run in parallel in C/C++ rather than Octave or Star-P® M with the Star-P® TPE. Use it if you have serial libraries in C and C++ you want to use in a task parallel manner. Using `ppeval` or `ppevalsplit` makes it easy to write wrapper functions in C/C++ using the task parallel Star-P® SDK API. When using `ppeval` or `ppevalsplit` in this manner, you need to build the function on the server and copy the module over to the HPC server machines.

The function `pploadpackage` can be used to load previously compiled shared object libraries whose contents can then be called using `ppeval`. Loading a compiled library for task parallel operations using `pploadpackage` requires calling syntax in one of two manners:

```
stringTP = pploadpackage('C','/path/to/package.so','TPname')
stringTP = pploadpackage('C','/path/to/package.so')
```

In either case, `pploadpackage` loads a package named `'package.so'` containing compiled functions for later use in `ppeval`. The initial string `'C'`, specifies the language in which the target package is written. Currently, only C or C++ libraries can be loaded on the server for task parallel operation, and both require the same initial string. The keyword argument “`name`” specifies a user-defined name that is used for identification of the task parallel package on the server. The string provided with the keyword argument `name` is returned in the function output `stringTP`. If the `name` keyword is not provided, then the naming convention utilized for assigning an output string to `stringTP` is to take the filename without path, extension, or underscores, converted to lowercase. This change ensures that the default name can always be used to prefix a function name, and is recognizable by the Star-P® client and server.

For more information about the task parallel API for using `ppeval` and `ppevalsplit` with compiled languages, see the “Star-P® Software Development Kit (SDK) Tutorial and Reference Guide”.

Per Process Execution

It is often useful to perform a certain operation only once per processor rather than performing the exact same operation within each iteration. Examples of such operations include opening and closing files or setting global variables. To enable a per process execution, one can use the Star-P® function named `np`, which returns the number of processors active in the current Star-P® session. For instance, the following sections of code

exemplify how to open a file for reading, read and process the data in the file, and close the file:

```
% Open the file in the variable filename on each ppeval process
fidpp = ppeval('open_file',filename,ppsplit(1:np));
% Apply algorithm to data samples contained in filename
resultspp = ppeval('process_file',1:num_data_samples);
% Close the file on each process
ppeval('fclose',fidpp);
```

where the functions `open_file` and `process_file` could look something like:

```
function fid = open_file(filename, pid)
% store file descriptor in global variable as well as return to client
global fid
%
fid = fopen(filename,'r');
gfid = fid;
```

and

```
function result = process_file(sample)
%
global gfid
% Read part of the data using fseek and fwrite and take an fft
sample_size = 8192;
% fseek to the correct location in the file
fseek(fid,sample_size*(sample-1),-1);
% Read the relevant section of the data
result = fread(gfid,sample_size);
% Apply the Fourier transform
result = fft(result);
```

Note that in the first line of the example above we used `ppsplit(1:np)` instead of `1:np`. This is because in the case that `np` happens to be equal to 1, the expression `1:np` returns a scalar. Normally, this input syntax is not valid due to the fact `ppeval` has no input argument over which it can iterate. In other words, `ppeval` received a string and a scalar, both of which will be broadcast by default. To override this behavior, use the `ppsplit` command on the `1:np` expression.

Calling Non-"M" Functions from within ppeval

The `ppeval` function can also be used to call a non-MATLAB program, via the `system` function and get results from that executable back into the Star-P® context. The simple example here illustrates a function `callapp2` that calls a pipeline of shell commands that returns the number of currently executing processors for a given user ID.

```
>> type callapp2
```

```
function z = callapp2(uid)
s = sprintf('ps -ael | grep %i | wc -l\n',uid)
[status, result] = system(s);
z = str2num(result);
```

Calling this from `ppeval` works as follows:

```
>> App = ppback([501 503 563 570])
App =
        ddense object: 1-by-4p
>> Bpp = ppeval('callapp2',split(App))
Bpp =
        ddense object: 1-by-4p
>> ppfront(Bpp)
ans =
    50      0      0      4
```

Note: In this case, the variable **`App`** has been split evenly to the available processors, which can be displayed by `np`. The default behavior of `split(App)` is to distribute along the last dimension, for example, `split(App,2)`.

You may want to note several things about this example.

1. It is not necessary for the number of calls made by `ppeval` to match the number of processors. `ppeval` uniformly allocates the number of calls over the number of processors available to the Star-P® session when the call is made.

2. Second, there is no built-in way for each column to know which column of the input it is. If that information is necessary for some reason, you will need to create such a vector and pass it in as another argument. For example, you can use the following statement:

```
>> Bpp = ppeval('callapp2',App,1:size(App,2));
```

3. Because `ppeval` is intended to take advantage of parallelism, each invocation of `callapp2` is done on a single processor of the HPC server. Star-P® takes care of the details of moving the function file (`callapp2` in this case) to the file system on the HPC server. Screen output from the called function will not appear on the Star-P® client. If you are reading or writing files in the called function, you will need to do those via paths relative to the file system structure on the server, not the current working directory of the MATLAB client. (Of course, if the file systems are the same between the client and the server, for example if they are NFS-mounted, then this is not an issue.)

By extension of this last example, almost any executable program could be called in parallel via `ppeval` using the `system` command, including end-user applications (written in C, C++ or Fortran) or third-party applications such as ANSYS, NASTRAN, FLUENT, or Gaussian. For further information on incorporating external applications in Star-P®, see the "Star-P® Software Development Kit (SDK) Tutorial and Reference Guide".

Workarounds and Additional Information

Below we discuss a couple of workarounds to some of the limitations that result from the requirement on the types of the input and output argument to `ppeval`. There are however several ways to work around these limitations. This section provides guidelines for the following:

- "String Arrays"
- "Splitting on a Scalar"
- "Global Variables"

String Arrays

To work with string arrays and `ppeval`, we use the fact that characters can be converted to variables of type double and doubles can be converted to variables of type character. So with minor modifications to a code, we can incorporate string arrays into applications that use `ppeval` to increase their performance.

```
str_arr = ['filename1';'filename2';'filename3'];
fnames = char(ppfront(ppeval('func',double(str_arr))));
```

with `func` defined below as follows:

```
function y = func(x)

y = x;
```

The `ppfront` command in this example is necessary since the `char` function has not been implemented in Star-P® and since Star-P® currently can only store objects of type double or double-complex.

Splitting on a Scalar

As mentioned in the section "Per Process Execution" above, it is possible to `ppsplit` on a scalar. This is a useful capability when one wants to do something only once, as in reading in an ASCII file on the server.

Global Variables

When running code inside of a `ppeval` command, several global variables that are specific to Star-P® are defined. These are:

1. `PP_COMM_SIZE` : The number of processors and `ppeval` engine processes.
2. `PP_MY_RANK` : The rank of this `ppeval` engine process, running from `0` to `PP_COMM_SIZE-1`.

3. `PP_TEMP_DIR` : The temporary work directory for the `ppeval` engine process.
4. `PP_CUR_ITER` : The value of the current iteration for each `ppeval` engine process. The `PP_CUR_ITER` counter runs from one to the number of slices for each `ppeval` engine process.

INTER*CTIVE supercomputing

Chapter 5

Tips and Tools for High Performance Star-P® Code

Star-P® enables MATLAB users to harness the computing power of HPC systems from within their familiar desktop environment. But as with any other software development environment or tool, there are advantageous and disadvantageous methods of using Star-P®.

This chapter provides tips for structuring your MATLAB codes for optimal performance using Star-P® and describes tools that can be utilized for monitoring and profiling the performance of your MATLAB applications using Star-P®. The tips and tools contained in this chapter are organized into the following sections:

- "Performance and Productivity"
- "Tips for Data Parallel Code"
- "Tips for Task Parallel Code"
- "Using External Libraries"
- "Integer Arithmetic in Star-P® Compared with MATLAB®"
- "Accuracy of Star-P® Routines"
- "Configuring ppsetoption for High Performance"
- "Performance Tuning and Monitoring"
- "UNIX Commands to Monitor the Server"

Performance and Productivity

The two most common reasons for users moving off their desktops to parallel computers are:

- to solve larger problems
- to solve problems faster

By contrast, users solve problems with MATLAB to take advantage of:

- ease of use
- high level language constructs
- productivity gains

To make the most of Star-P®, you need to find your own “comfort level” in the trade-off between productivity and performance. This is not a new trade-off. In 1956, the first so-called high level computer language was invented: FORTRAN. At the time, the language was highly criticized because of its relatively poor performance compared to programs that were highly tuned for special machines. Of course, as the years passed, the higher-level language outlasted any code developed for any one machine. Libraries became available and compilers improved.

This lesson is valuable today. To take advantage of Star-P®, you will benefit from simply writing MATLAB code, and inserting the characters `*p` at just the right times. You can improve performance both in terms of problem sizes and speed by any of the following means:

- restructure the serial MATLAB program through vectorization (described in “Vectorization”)
- restructure the serial MATLAB program through uses of functionally equivalent commands that run faster
- restructure the serial MATLAB program through algorithmic changes

You may not wish to change your MATLAB programs. Programs are written in a certain style that expresses the job that needs to be done. Psychologically, a change to the code may feel risky or uncomfortable. Programmers who are willing to make small or even large changes to programs may find huge performance increases both in serial MATLAB and with Star-P®.

Typically, changes that speed up serial MATLAB also speed up Star-P®. In other words, the benefits of speeding up the serial code multiply when going parallel.

You may want to develop new applications rapidly that work on very large problems, but absolute performance may not be critically important. The MATLAB operators have proven to be very powerful for expressing typical scientific and engineering problems. Star-P® provides a simple way to use those operators on large data sets. Today, Star-P® is early in its product life, and will undoubtedly see significant optimizations of existing operators in future releases. Your programs will transparently see the benefit of those optimizations. You benefit from ease of use and portability of code today.

Tips for Data Parallel Code

Vectorization

Vectorization is the process of converting a code from explicit element-by-element calculations to higher level operators that operate on entire vectors or arrays at a time. Vectorization reduces the amount of time spent in MATLAB or Star-P® bookkeeping operations and increases the amount of time spent doing the mathematical operations that are the purpose of your program. Vectorization is a process well known to many experienced MATLAB programmers, as it often gives markedly better performance for MATLAB execution. In fact, The MathWorks provides an online tutorial about vectorization at http://www.mathworks.com/support/tech-notes/1100/1109.html. The process of vectorization for both MATLAB and Star-P® execution is the same.

Vectorization speeds up serial MATLAB programs and eases the path to parallelization in many instances.

Note: The following MATLAB timings were performed on a Dell Dimension 2350. The Star-P® timings were performed on an SGI Altix system. Note that small test cases are used so that the unvectorized versions will complete in reasonable time, so the speedups shown in these examples are modest.

Example 4-1: Sample summation of a vector

The following MATLAB® code is not vectorized:

```
>> v = 1:1e6;
>> s = 0;
>> tic;
>> for i=1:length(v), s = s+v(i); end
>> toc;
Elapsed time is 0.684787 seconds.
>> s
s =
   5.0000e+11
```

The following line is vectorized:

```
>> v = 1:1e6;
>> tic;
>> s = sum(v);
>> toc;
Elapsed time is 0.003273 seconds.
```

The two ways of summing the elements of `v` give the same answer, yet the vectorized version using the sum operator runs more than 100 times faster. This is an extreme case of the speed-up due to vectorization, but not rare. Expressing your algorithm in high level operators,

provides more opportunities for optimization by Star-P® (or MATLAB) developers within those operators, resulting in better performance.

The following MATLAB code is parallelized:

```
>> vpp = 1:1e6*p;
>> s = sum(vpp);
```

Based on the vectorized form, it is straightforward to move to a parallel version with Star-P®. Note that the unvectorized form, since it's calculating element-by-element, would be executing on only a single processor at a time, even though Star-P® would have multiple processors available to it!

Example 4-2: Simple polynomial evaluation

The following MATLAB code is not vectorized:

```
>> v = 1:1e7;
>> w = 0*v;
>> tic;
>> for i=1:length(v), w(i) = v(i)^3 + 2*v(i); end
>> toc;
Elapsed time is 19.815496 seconds.
% The following code is vectorized
>> tic;
>> w = v.^3 + 2*v;
>> toc;
Elapsed time is 2.137521 seconds.
% The following code is parallelized
>> vpp = 1:1e7*p;
>> tic;
>> wpp = vpp.^3 + 2*vpp;
>> toc;
Elapsed time is 0.118621 seconds.
```

This example shows exactly the value of vectorization: it creates simpler code, as you don't have to worry about getting subscripts right, and it allows the Star-P® system bigger chunks of work to operate on, which leads to better performance.

Example 4-3: BLAS-1 compared to BLAS-3 matrix multiplication

This example compares two methods of multiplying two matrices. One (partially vectorized) uses `dot` n^2 times to calculate the result. The vectorized version uses the simple * operator to multiply the two matrices; this results in a call to optimized libraries (PBLAS in the case of Star-P®) tuned for the specific machine you're using. These versions compare to the BLAS Level 1 `DDOT` and BLAS Level 3 `DGEMM` routines, where exactly the same effect holds. Higher-level operators allow more flexibility on the part of the library writer to achieve optimal performance for a given machine.

Contents of the script `mxm.m`:

```
>> type mxm
for i=1:n
  for j=1:n
    c(i,j) = dot(a(i,:),b(j,:));
  end
end
>> n = 1000;
>> a = rand(n); b = rand(n);
% Unvectorized
>> tic; mxm; toc;
Elapsed time is 70.821620 seconds.
% Vectorized on a single processor
>> tic; d = a*b; toc;
Elapsed time is 0.431142 seconds.
% Vectorized and parallel
>> npp = n*p;
>> app = rand(npp); bpp = rand(npp);
>> tic; dpp = app*bpp; toc;
Elapsed time is 0.118349 seconds.
```

Example 4-4: Recognizing a histogram

This example is a bit fancy. If you are going to restructure this construct, it requires you to recognize that two computations are the same; the first is not vectorized, while the second may be considered vectorized. Here the trick is to recognize that the code is computing a histogram and then cumulatively adding the numbers in the bins.

Form 1: Unvectorized and unrecognized:

```
>> v = rand(1,1e7);
>> w = [];
>> i = 0;
>> tic;
>> while (i<1), i=i+0.1; w = [w sum(v<i)]; end
>> toc
Elapsed time is 0.947873 seconds.
>> w.'
ans =
      997890
     1998324
     2996577
     3997599
     4999280
     6000307
     7000870
     8000829
     9000054
    10000000
    10000000
```

Form 2: Vectorized cumulative sum and histogram:

```
>> tic; w = cumsum(histc(v,0:.1:1)); toc
Elapsed time is 0.382109 seconds.
>> w.'
ans =
       997890
      1998324
      2996577
      3997599
      4999280
      6000307
      7000870
      8000829
      9000054
     10000000
     10000000
```

As one would expect, the vectorized version works best in Star-P® as well.

Star-P® Solves the Breakdown of Serial Vectorization

For all but the smallest of loops, vectorization can give enormous benefits to serial MATLAB code. However, as array sizes get larger, much of the benefit of serial vectorization can break down. The good news is that in Star-P® vectorization is nearly always a good thing. It is unlikely to break down.

The problem with serial MATLAB is that as variable sizes get larger, MATLAB swaps out the memory to disk. This is a very costly measure. It often slows down serial MATLAB programs immensely.

There is a serial approach that can partially remedy the situation. You may be able to rewrite the code with an outer loop that keeps the variable size small enough to remain in main memory while large enough to enjoy the benefit of vectorization. While for some problems this may solve the problem, users often find the solution ugly and not particularly scalable. The other remedy uses the Star-P® system. This example continues to use vectorized code, inserting the Star-P® at the correct points to mark the large data set.

As an example, consider the case of FFTs performed on matrices that are near the memory capacity of the system MATLAB is running on.

```
>> n = 1.2*10^4;
>> a   = rand(n);
>> app = rand(n*p);
>> tic; b   = fft( a); toc;
Elapsed time is 92.685374 seconds.
>> tic; bpp = fft(app); toc;
Elapsed time is 6.916634 seconds.
```

While you would expect Star-P® to be faster due to running on multiple processors, Star-P® is also benefiting from larger physical memory. The serial MATLAB execution is hampered by a lack of physical memory and hence runs inordinately slow. A recurring requirement for efficient Star-P® programs is keeping large datasets off the front end.

The code below shows what happens upon computing 2^26 random real numbers with decreasing vector sizes. When k=0, there is no loop, just one big vectorized command. On the other extreme, when k=25, the code loops 2^25 times computing a small vector of length 2.

Notice that in the beginning, the vectorized code is not efficient. This turns out to be due to paging overhead, as the matrix exceeds the physical memory of the system on which MATLAB is running. Later on, the code is inefficient due to loop overhead. Star-P® overcomes the problem of insufficient memory by enabling you to run on larger-memory HPC systems. The simple command `app = randn(2^26*p,1)` parallelizes this computation.

Serial:

```
>> for k=0:25, tic; for i=1:2^k, a = randn(2^(26-k),1); end; toc; end;
Elapsed time is 1.865770 seconds.
Elapsed time is 1.600310 seconds.
Elapsed time is 1.581707 seconds.
Elapsed time is 1.590823 seconds.
Elapsed time is 1.597639 seconds.
Elapsed time is 1.577038 seconds.
Elapsed time is 1.579628 seconds.
Elapsed time is 1.578954 seconds.
Elapsed time is 1.581229 seconds.
Elapsed time is 1.163945 seconds.
Elapsed time is 1.059308 seconds.
Elapsed time is 1.165907 seconds.
Elapsed time is 1.079797 seconds.
Elapsed time is 1.069463 seconds.
Elapsed time is 1.090218 seconds.
Elapsed time is 1.145205 seconds.
Elapsed time is 1.235547 seconds.
Elapsed time is 1.453363 seconds.
Elapsed time is 1.883642 seconds.
Elapsed time is 2.731986 seconds.
Elapsed time is 4.467244 seconds.
Elapsed time is 7.057231 seconds.
Elapsed time is 13.076593 seconds.
Elapsed time is 25.143928 seconds.
Elapsed time is 44.867566 seconds.
Elapsed time is 88.540178 seconds.
```

Solving Large Problems: Memory Issues

The ability of MATLAB and Star-P® to create and manipulate large matrices easily sometimes conflicts with the desire to run a problem that consumes a large percentage of the physical memory on the system in question. Many operators require a copy of the input, or sometimes temporary array(s) that are the same size as the input, and the memory consumed by those temporary arrays is not always obvious. Both MATLAB[1] and Star-P® will run much more slowly when their working set exceeds the size of physical memory, though Star-P® has the advantage that the size of physical memory will be bigger.

If you are running into memory capacity issues, as evidenced by server exceptions being logged in the `~/.starp/log/latest/starpserver.log`, then there may be one or a few places that are using the most memory. In those places, manually inserting `clear` statements for arrays no longer in use, allows the Star-P® garbage collector to free up as much memory as possible.

As a means of determining where in your application you are requesting a larger amount of memory than is available for use, then you may consider enabling the `STARP_SOFT_MEM_LIMITS` environment variable in the `env.sh` file on the server, located in the `<path/to/starp/install>/config` directory, or placing this environment variable in the user's `.bashrc` file, also on the server. `STARP_SOFT_MEM_LIMITS` controls whether "soft-limits" will be enforced (`= true`) or not (`= false`). By default, the value of `STARP_SOFT_MEM_LIMITS` is set to be `false`.

When `STARP_SOFT_MEM_LIMITS` is set to `true`, server "out-of-memory" exceptions are returned to the client when one or more server processes exceed their "soft limit" for memory allocation and a subsequent large array allocation is attempted. The "soft-limits" approach utilizes UNIX "`setrlimit`" functionality to limit the user virtual memory to 1/Nth of the available physical memory on a system's cache coherent domain that is running N Star-P® server processes. This artificial limit for memory allocation allows exception handling to be focused at specific points in the Star-P® server code and should allow users to code for these exceptions. This limit helps eliminate unexpected slow downs due to oversubscription of memory.

With `STARP_SOFT_MEM_LIMITS` set to `false`, an exception will be thrown on the server and an error message returned to the client, upon the first call to `malloc()`, the call for memory allocation in C, that exceeds the available memory on the server. If `STARP_SOFT_MEM_LIMITS` is set to `true`, then at the first call to `malloc()` where a request for memory exceeds this soft-limit, a null pointer is returned, and this event is recorded for later exception handling on the server. The soft-limit is then disabled, and the operation is repeated. If at this point, the call to `malloc()` does not exceed the free memory available on the system, then your application will continue. If the subsequent call to

1. The MathWorks has a help page devoted to handling memory issues at http://www.mathworks.com/support/tech-notes/1100/1106.html.

`malloc()` exceeds the memory available on the system, then an exception is thrown on the server, and an error message is returned to the client.

When examining the performance of your code using "soft-limits", you should also be aware of the Star-P® `mallochooks` setting on the server. `mallochooks` is set using ppsetoption. It provides a thin wrapper around the `malloc()` operation in C that records the user request in the case of a failed `malloc()`. This wrapper also provides that record at a later point to the Star-P® server for logging purposes. If you choose to set `mallochooks` to be off, then you are turning off Star-P®'s mechanism for tracking memory usage on the server. Any out of memory errors that occur with `mallochooks` turned off are subject to the memory limits and error handling provided by your server's operating system.

Further user control for the actual memory "soft" limit is available via the `STARP_MBYTES_PER_PEER` environment variable. If defined, this environment variable will override the default limit which is calculated to be $1/N^{th}$ of the host's actual physical memory. `STARP_MBYTES_PER_PEER` can be exceed the default value, but should be used with caution since it will allow oversubscription of memory, and could thereby cause application slow down due to swapping.

Tips for Task Parallel Code

Use of Structs and Cell Arrays

MATLAB codes allow for the use of structs and cell arrays as a convenient method of collecting and organizing related data sets. Within MATLAB, the contents of these containers can be any valid MATLAB data type, including matrices, strings, and other structures or cell arrays. Depending on the code being developed, these arrays may be gigantic arrays of structures or cells.

Star-P® currently allows the use of structures locally inside of functions called by "`ppeval`". Structs and cell arrays on the client side continue to work within the MATLAB environment. You can assign distributed data to members of a struct or cell array, as well as manipulate distributed data that is a member of a struct or cell array. However, current versions of Star-P® lack the ability to pass entire structures or cell arrays from client to server. This means that you cannot pass a top-level struct or cell array name as an argument to "`ppeval`", nor can you distribute an entire struct using `ppback`. Here are some examples:

```
%Legal Star-P® operations on structs

a.scalar = 57.36;
a.foo     = ppback([1:100]);
a.left    = rand(100*p,100);
a.right   = rand(100,100*p);
myprod    = a.left*a.right;
bar       = ppeval('somefunc', split(a.foo), split(a.left,1),split(a.right,2));
```

```
%Illegal Star-P® operations -- can't pass struct using top-level name
b.this    = rand(10,10);
b.that    = rand(10,10);
baz       = ppeval('someotherfunc',b);
bpp       = ppback(b);
```

MATLAB structs or cell arrays can be arbitrarily more complex than shown here (for example, structs containing cell arrays containing structs among other possibilities). As a general rule, if your data is held in a struct or cell array, and you need to pass a part of that data to the server, then pass only the structure members or contents of a cell array element that contain distributable matrix data or string variables.

When creating replacement variables for passing this data into or out of a `ppeval` call, 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 for Loops Inside of ppeval Calls

For similar reasons that vectorization is key to achieving optimal performance with data parallel codes, vectorization is also extremely important for good performance of task parallel codes. Each iteration of a function in task parallel takes place on an individual processor on the server and still involves the use of an interpreter. Consequently, the benefits of vectorization that can be achieved in serial MATLAB code are also available with task parallel MATLAB code with Star-P®.

The following example shows the effort needed and gains achieved by vectorization inside a "`ppeval`" call:

```
%Top.m -- Top level fcn invokes two different versions of sum to check speeds.

%Main function. Assume computation involves processing 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),split(zarr,3));
toc

function x = fcn_looping(n,y,z)
%===== Unvectorized version -- Bad! =====
for i = 1:n
if z(1,i) >= 0.5
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

function x = fcn_vectorized(n,y,z)
%===== Vectorized version -- Good! =====
indx = z(1,:) >= 0.5;       %Replace if with a logical expression
x(indx) = sum(y(:,indx).*z(:,indx),1);
indx = indx == 0;   %use complement of indx for else case
x(indx) = sum(y(:,indx)./z(:,indx),1);
```

The performance gains achieved by this vectorization inside of a `ppeval` are shown in the following table as a function for the main loop index.

Loop Iterator	Looping Execution Time (sec)	Vectorized Execution Time (sec)	Speed-up Factor
100	0.798259	0.751317	1.06
1000	1.444807	0.815917	1.77
10000	9.126191	1.231624	7.41
100000	351.754691	5.797857	60

Performance Note on Iteration Timing

Each `ppeval` iteration has overhead cost associated with it on the order of 10s of microseconds. This means that if iterations of your `for` loop take less time than this overhead cost no performance gains will be achieved by using `ppeval` directly over the entire set of iterations. By blocking iterations within a function call, you can:

1. reduce the number of iterations performed by `ppeval`
2. increase the time per `ppeval` iteration
3. reduce the overall time necessary to perform all target function iterations.

To illustrate this point, let us consider a function `foo` contained in a file `foo.m`. Let us also assume that evaluating a single iteration of `foo` takes less than 10-20 microseconds.

Performing N iterations of `foo` serially would take the following form:

```
% Performing all iterations serially
% on a function foo that takes two scalars
m = 6;
N = 2^m;
x = rand(N,1);
y = randn(N,1);
z = zeros(N,1);
```

```
for i=1:N
    z(i) = foo(x(i),y(i));
end
% file foo.m
function z = foo(x,y);
z = x+y;
```

Performing all N iterations in a single `ppeval` would then be:

```
% Performing all iterations in a single ppeval
xpp = ppback(x);
ypp = ppback(y);
zpp = ppeval('foo',xpp,ypp);
```

Now in the assumed case where `foo` takes less than 10-20 microseconds to execute, it is recommended that you rearrange the loop body to create a wrapper function, say `foo_wrapper`, that executes only a portion of your iterations serially. Then, this function `foo_wrapper` would be passed to `ppeval`.

```
% from file foo_wrapper.m
function z = foo_wrapper(x,y,N);
z = zeros(N,1);
for i=1:N
    z(i) = foo(x(i),y(i));
end
%Transformed Serial operations
m = 6;
N = (2^m/np);
x = rand(N,np);
y = randn(N,np);
z = zeros(N,np);
for j = 1:np
    z(:,j) = foo_wrapper(x(:,j),y(:,j));
end
%Implementation in parallel with effective starp.ppeval operations
xpp = ppback(x);
ypp = ppback(y);
zpp = ppeval(foo_wrapper,xpp,ypp,N);
z = ppfront(zpp(:));
```

This example assumes that the total number of iterations desired for `foo` is a multiple of the number of processors. When this is not the case, the logic of how you choose to break up your for loops needs to be changed. In addition, the optimal method for determining the number of iterations that should be performed inside `foo_wrapper`, which is called by `ppeval`, is something that you will need to determine through experimentation based on the following quantities:

- The amount of time necessary to call a single iteration of `foo`.
- The total number of iterations of `foo` needed.
- The number of processors available.

Using External Libraries

In addition to the functions available within MATLAB®, Star-P® allows for the integration of external functions for your own libraries or third party vendor libraries through the use of the `ppinvoke`, `ppeval`, and `ppevalsplit`, `pploadpackage`, and `ppunloadpackage` functions that are part of Star-P® SDK interface. More information on the Star-P® SDK can be found in the "Star-P® Software Development Kit Reference and Tutorial". External functions can also be run in task parallel within `ppeval` through the use of the MATLAB `system` command. For more information on calling external functions using the `system` command, see "Calling Non-"M" Functions from within ppeval".

Although most Star-P® functions are insensitive to matrix distribution, many (or most) third party libraries are not. Consequently, if your MATLAB® program interfaces to external programs or libraries that are sensitive to distribution through the Star-P® SDK, then you must carefully consider how you distribute your matrices. In this situation you may ask, "how do I know whether to call the function with row distributed or column distributed input matrices?" Unless the third party programs explicitly state their desired distribution, then the answer is: experiment. Surround the function with "`tic/toc`", "`pptic/pptoc`" and send it random matrices distributed in all ways possible. Then scale the matrix sizes up and see which distributions (if any) offer faster execution time, or which distributions break first when the matrix size becomes gigantic.

Integer Arithmetic in Star-P® Compared with MATLAB®

In MATLAB®, all operations on integer types "saturate." This means values greater than `intmax` of that integer class are set to `intmax` and values below `intmin` of that integer class are set to intmin.

In Star-P® M, all operations on integers "overflow." This behavior is more common in languages and environments that support integer operations such as C, C++, C#, Java and NumPy. This means when a value is greater than `intmax` for a particular integer class the result will cycle back from `intmin` of that class. Similarly, when a value is less than `intmin` of an integer class, the result will cycle starting from `intmax`.

Accuracy of Star-P® Routines

The underlying numerical libraries in Star-P® such as ScaLAPACK, FFTW and SPRNG are of high accuracy and comply with the IEEE standards. However, in many cases the results from Star-P® may differ from that reported by MATLAB® for a number of reasons:

1. In the most common case, the answers may simply be non-unique. For example, the eigenvalues from the single-return form of `eig` might be returned in a different order

from MATLAB® or the eigenvectors might be scaled differently. Similarly, the outputs from `svd` (singular value decomposition) and its derivatives such as `null` and `orth`, `hess` (reduction to the upper-Hessenberg form) and `schur` (reduction to the Schur form) are non-unique and therefore not guaranteed to match the corresponding outputs from MATLAB®. Instead, you must verify that the results satisfy the properties of the underlying decomposition. For instance, if you were to run `[Upp,Spp,Vpp]=svd(App)` for `App` being a `ddense` object, the outputs `Upp`, `Spp`, and `Vpp` are valid if `Upp` and `Vpp` are unitary and `norm(Upp*Spp*Vpp - App)` is small.

2. Another reason numerical results from Star-P® might not correspond to those from MATLAB has to do with the influence of small round-off errors. For example, it is well-known that even addition is not associative in the presence of rounding errors: the result of `(a + b) + c` can differ from `a + (b + c)`.

3. When the underlying problem is ill-conditioned or singular, it is very likely that the results from Star-P® will not match MATLAB. For instance, when a matrix A is singular to working precision, `inv(A)` returns `inf(size(A))` in MATLAB, but not in Star-P®. When such cases are encountered, Star-P® does its best to return a descriptive warning message.

4. Differences between MATLAB and Star-P® numerical results might arise for extremely large or extremely small input values.

5. Finally, differences between the numerical results in Star-P® and MATLAB might result from software issues. If you suspect that the Star-P® result is incorrect, please contact us at support@interactivesupercomputing.com.

Configuring `ppsetoption` for High Performance

The following items should be considered when the performance of your Star-P® application is critical.

1. 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, via the following command at the MATLAB® prompt:

```
ppsetoption('log','off')
```

Through increasing the frequency of calls, by setting `ppgcFreq` to a number smaller than 30, the server can use less memory. This could be useful when executing server calls which allocate a lot of temporaries over a few server calls.

2. On SGI Altix systems, the Star-P® server will yield CPU usage after each command completes. Significantly improved performance is available at the cost of continuing to use CPU time in a loop even when Star-P® is idle. This increased performance can be obtained via the following command at the MATLAB prompt:

```
ppsetoption('YieldCPU','off')
```

3. By default, the Star-P® server maintains a count of how much memory it is consuming and how much memory is being used on the system. This enables it to more gracefully handle situations that arise when the server machine is running low on available memory. There is a minor performance cost associated with this functionality, because it requires a small amount of extra work to be done with each call to `malloc()` inside the server. This feature can be disabled, providing improved performance, via the following command:

```
ppsetoption('mallochooks','off')
```

Star-P® TPE provides the option of using various versions of Octave or compiled C codes as task parallel computational engines using the `'TaskParallelEngine'` option as an argument in `ppsetoption`.

For information on using `ppsetoption` to change your task parallel engine, see "Choosing Your Task Parallel Engine (TPE)".

Performance Tuning and Monitoring

Diagnostics and Performance

Star-P® provides several diagnostic commands that help determine the following:

- Which variables are distributed,
- How much time is spent on communication between the client and the server.
- How much time is spent on each function call inside the server.

Each of these diagnostics can help identify bottlenecks in the code and improve performance. The diagnostic arguments are `ppwhos`, `pptic/toc`, `ppeval_tic/toc`, and `ppprofile`.

Client/Server Performance Monitoring

Coarse Timing with pptic and pptoc

Communication between the client and server can be measured by use of the `pptic` and `pptoc` commands, which are modeled after the MATLAB® `tic` and `toc` commands, but instead of providing wall-clock time between the two calls, they provide the number of client-server messages and bytes sent during the interval.

```
>> app = randn(1000*p);
>> tic; bpp = fft(app), toc;
bpp =
        ddense object: 1000-by-1000p
```

```
Elapsed time is 0.022127 seconds.
>> pptic; bpp = fft(app), pptoc;
bpp =
        ddense object: 1000-by-1000p
Client/server communication report:
  Sent by server: 1 messages, 7.200e+01 bytes
  Received by server: 1 messages, 8.800e+01 bytes
  Total communication time: 5.341e-05 seconds
Server processing report:
  Duration of calculation on server (wall clock time): 1.802e-02s
  #ppchangedist calls: 0
----------------------------------------------------------------------
Total time: 3.410e-02 seconds
```

And of course the two can be combined to provide information about transfer rates.

```
>> tic; pptic; bpp = fft(app), pptoc; toc;
bpp =
        ddense object: 1000-by-1000p
Client/server communication report:
  Sent by server: 1 messages, 7.200e+01 bytes
  Received by server: 1 messages, 8.800e+01 bytes
  Total communication time: 5.198e-05 seconds
Server processing report:
  Duration of calculation on server (wall clock time): 1.881e-02s
  #ppchangedist calls: 0
----------------------------------------------------------------------
Total time: 3.241e-02 seconds
Elapsed time is 0.032588 seconds.
```

The `pptic` and `pptoc` commands can be used on various amounts of code, to focus on the source of a suspected performance problem involving communications between the client and the server. For instance, when you explicitly move data between the client and server via `ppfront` or `ppback`, you will expect to see a large number of bytes moved.

```
>> app = rand(1000*p);
>> pptic; ma = ppfront(app); pptoc;
Client/server communication report:
  Sent by server: 1 messages, 8.000e+06 bytes
  Received by server: 1 messages, 2.400e+01 bytes
  Total communication time: 7.060e-01 seconds
Server processing report:
  Duration of calculation on server (wall clock time): 1.566e-02s
  #ppchangedist calls: 0
----------------------------------------------------------------------
Total time: 7.415e-01 seconds
>> ppwhos
Your variables are:
  Name        Size            Bytes        Class
  app         1000x1000p      8000000      ddense array
  ma          1000x1000       8000000      double array
```

```
Grand total is 2000000 elements using 16000000 bytes
MATLAB has a total of 1000000 elements using 8000000 bytes
Star-P® server has a total of 1000000 elements using 8000000 bytes
```

But there might be places where implicit data movement occurs. For example, below we see an example of a distributed matrix being multiplied by a local, client-side matrix. In performing this operation, the matrix `b` must be shipped to the server to perform this operation.

```
>> app = rand(1000*p);
>> b = rand(1000);
>> pptic; cpp = app * b; pptoc;
Client/server communication report:
  Sent by server: 2 messages, 1.480e+02 bytes
  Received by server: 2 messages, 8.000e+06 bytes
  Total communication time: 6.799e-01 seconds
Server processing report:
  Duration of calculation on server (wall clock time): 1.403e-01s
  #ppchangedist calls: 0
----------------------------------------------------------------------
Total time: 8.587e-01 seconds
```

Other operations may produce different amounts of communication depending upon how they are called. For example, the single-return case of the `find` function may move only a few hundreds or thousands of bytes between the client and the server, but when calling the find operation on a distributed variable with three returns, the row indices, column indices and array values are all moved from the server to the client. Depending on the size of the distributed input, this could be a very large amount of data that is transferred.

An excessive number of client-server messages (as opposed to bytes transferred) can also hurt performance. For instance, the values of an array could be created element-by-element, as in the `for` loop below, or it could be created by a single array-level construct as below. The first construct calls the Star-P® server for each element of the array, meaning almost all the time will be spent communicating between the client and the server, rather than letting the server spend time working on its large data.

```
>> app = rand(100*p,1);
>> bpp = rand(100*p,1);
>> tic; pptic;
>> for i = 1:double(size(app,1)), cpp(i,1) = app(i,1)*2 + bpp(i,1)*7.4; end
>> pptoc; toc;
Client/server communication report:
  Sent by server: 200 messages, 8.800e+03 bytes
  Received by server: 200 messages, 1.281e+04 bytes
  Total communication time: 3.047e-03 seconds
Server processing report:
  Duration of calculation on server (wall clock time): 5.492e-01s
  #ppchangedist calls: 0
----------------------------------------------------------------------
Total time: 8.171e-01 seconds
Elapsed time is 0.817286 seconds.
```

The second construct is drastically better because it allows the Star-P® server to be called only a few times to operate on the same amount of data.

```
>> app = rand(100*p,1);
>> bpp = rand(100*p,1);
>> tic; pptic;
>> cpp(:,1) = app(:,1)*2 + bpp(:,1)*7.4;
>> pptoc; toc;
Client/server communication report:
  Sent by server: 7 messages, 5.080e+02 bytes
  Received by server: 7 messages, 5.680e+02 bytes
  Total communication time: 1.159e-04 seconds
Server processing report:
  Duration of calculation on server (wall clock time): 4.857e-02s
  #ppchangedist calls: 0
----------------------------------------------------------------------
Total time: 9.490e-02 seconds
Elapsed time is 0.095106 seconds.
```

The execution of this script bears out the differences in messages sent/received, with the first method sending 200 times more messages than the second. What is even worse for the element-wise approach, the performance difference will grow as the size of the data grows.

Summary and Per-Server-Call Timings with `ppprofile`

The different subfunctions of the `ppprofile` command can be combined to give you lots of information about where the time is being spent in your Star-P® program. There are different types of information that are available.

Perhaps the most common usage of `ppprofile` is to get a report on a section of code, as follows.

```
>> app = rand(1000*p);
>> ppprofile on
>> doffts(app)
>> ppprofile report
function                  calls       time   avg time     %calls      %time
ppbase_setoption              1   0.079922   0.079922    11.1111    45.9766
starp_fft1                    2   0.046222   0.023111    22.2222      26.59
ppdense_max                   2   0.016304   0.008152    22.2222     9.3792
ppdense_unary_op              1    0.01081    0.01081    11.1111     6.2186
ppdense_binary_op             1   0.010652   0.010652    11.1111     6.1278
ppdense_viewelement           2   0.009922   0.004961    22.2222     5.7078
Total                         9    0.17383   0.019315
```

The report prints out all server functions that are used between the calls to `ppprofile on` and `ppprofile report`, sorted by the percentage of the execution time spent in that function. For this example, it shows you that 34% of the time is spent executing in the server routine `ppfftw_fft`, which calls the FFT routine in the FFTW parallel library. This report

also tells you how many calls were made to each server routine, and the average time per call.

Information from this report can be used to identify routines that your program is calling more often than necessary, or that are not yet implemented optimally. An example of the former is given below, by a script which does a matrix multiplication in a non-vectorized manner, compared to a vectorized routine. The script has the following contents:

```
>> type domxmp
function c = domxmp(a,b)
%  Do matrix multiply by various methods (bad to good perf)
%
%  First, do it in an unvectorized style.
[ma, na] = size(a);
[mb, nb] = size(b);
ppprofile clear, ppprofile on;
tic;
c = zeros(ma,nb)
for i = 1:double(ma)
  for j = 1:double(nb)
    c(i,j) = dot(a(i,:),b(:,j));
  end
end
fprintf('MxM via dot takes ');
toc;
ppprofile report
%  Second, do it in a vectorized style.
ppprofile clear, ppprofile on;
tic;
c = a*b;
fprintf('\nMxM via dgemm takes ');
toc;
ppprofile report
```

With two input arrays sized as 20-by-20p, you get the following output:

```
>> app = rand(20*p);
>> bpp = rand(20*p);
>> cpp = domxmp(app,bpp);
cpp =
        ddense object: 20-by-20p
MxM via dot takes Elapsed time is 13.243921 seconds.
function                  calls       time  avg time    %calls     %time
ppdense_setelement          400     3.1333 0.0078333   24.9688   37.7747
ppdense_subsref_col         400     1.7981 0.0044952   24.9688    21.677
ppdense_subsref_row         400     1.7178 0.0042944   24.9688    20.709
ppdense_dotv                400     1.6218 0.0040545   24.9688   19.5519
ppbase_setoption              1   0.019089  0.019089  0.062422   0.23013
ppdense_zeros                 1   0.004753  0.004753  0.062422  0.057301
Total                      1602     8.2948 0.0051778
MxM via dgemm takes Elapsed time is 0.007292 seconds.
```

```
function                     calls       time   avg time    %calls     %time
pppblas_gemm                     1   0.003829   0.003829        50   54.4588
ppbase_setoption                 1   0.003202   0.003202        50   45.5412
Total                            2   0.007031  0.0035155
```

You can see that the first report requires over 2,000 server calls, while the second requires only one. This accounts for the drastic performance distance between the two styles of accomplishing this same computational task.

If you want to delve more deeply and understand the sequential order of system calls, or get more detailed info about each server call, you can use the `ppprofile display` option.

```
>> app = rand(1000*p);
>> ppprofile off
>> ppprofile on, ppprofile display
>> doffts(app)
starp_fft1  time=0.019604
starp_fft1  time=0.014379
ppdense_binary_op  time=0.011343
ppdense_unary_op  time=0.01186
ppdense_max  time=0.009945
ppdense_max  ppdense_viewelement  time=0.003043
ppdense_viewelement  time=0.006743
time=0.00759
>> ppprofile off
```

With this option, the information comes out interspersed with the usual MATLAB console output, so you can see which MATLAB or Star-P® commands are invoking which server calls. This can help you identify situations where Star-P® is doing something you didn't expect, and possibly creating a performance issue.

Another level of information is available with the `ppprofile on -detail full` option coupled with the `ppprofile display` option.

```
>> ppprofile off
>> ppprofile on -detail full
>> ppprofile display
>> doffts
echo on
bpp = fft(app);
ppfftw_fft          time=0.11616 stime=0        chdist=0
ppbase_gc_many      time=3.1397  stime=0.078125 chdist=0
cpp = ifft(bpp);
ppfftw_fft          time=0.33294 stime=0.10938  chdist=0
diff = max(max(abs(cpp-app)))
ppdense_elminus     time=0.16202 stime=0.046875 chdist=0
ppdense_abs         time=0.20006 stime=0.09375  chdist=0
ppdense_max         time=0.12719 stime=0.015625 chdist=0
ppbase_removeMatrix time=0.11105 stime=0        chdist=0
ppdense_max         time=0.12176 stime=0        chdist=2
ppdense_viewelement time=0.11217 stime=0        chdist=0
```

```
ppbase_gc_many       time=0.1127  stime=0        chdist=0
ppdense_viewelement time=0.11225 stime=0        chdist=0
diff =
7.7963e-16
```

As you can see, the per-server-call information now includes not only the time spent executing on the server ("`stime`") but also the number of times that the distribution of an object was changed in the execution of a function ("`chdist`"). Changes of distribution are necessary to provide good usability (think of the instance where you might do element-wise addition on 2 arrays, one of which is row-distributed and one of which is column-distributed), but changing the distribution also involves communication among the processors of the Star-P® server, which can be a bottleneck if done too often. In this example, the `max` function is doing 2 changes of distribution.

ppeval_tic/toc:

Star-P® also provides a set of timer functions specific to the `ppeval` command: `ppeval_tic/ppeval_toc`. They provide information on the complete `ppeval` process by breaking down the time spent in each step necessary to perform a `ppeval` call:

```
>> ppeval_tic();
>> ypp = ppeval('inv',rand(10,10,1000*p));
>> ppeval_toc(0)
ans =
             TotalCalls: 1
             ServerInit: 6.1989e-06
           ServerUnpack: 5.0068e-06
      ServerFunctionGen: 0.0019
        ServerCallSetup: 1.9908e-04
       ServerOctaveExec: 0.0493
      ServerDataCollect: 2.0599e-04
        ServerTotalTime: 0.0516
          ClientArgScan: 0.0050
           ClientDepFun: 0.0028
            ClientEmode: 0.0549
     ClientReturnValues: 0.0096
        ClientTotalTime: 0.0723
       TPELogFileLength: 84
        InputElementsPP: [12503 0]
       OutputElementsPP: [12500 0]
           TPEInnerExec: 0
           TPEOuterExec: 0.0478
          TPESliceCount: 125
```

`ppeval_tic/toc` is useful to determine how much time is spent on actual calculation (`ServerOctaveExec`) and how much on server (`ServerTotalTime - ServerTPEExec)` and client (`ClientTotalTime`) overhead. The argument to `ppeval_toc` determines is the maximum time of all processors (0), the minimum time of all processors (1), or the mean time of all processors (2) is returned.

Maximizing Performance

Maximizing performance of Star-P® breaks down to the following guidelines:
1. minimize client/server communication,
2. keep data movement between the client and server to a minimum, and
3. keep distributions of server variables aligned.

The first point is most important for data parallel computation and can be achieved by vectorizing your code, meaning, that instead of using looping and control structures, you use higher level functions to perform your calculations. Vectorization takes control of the execution away from MATLAB (e.g., MATLAB is no longer executing the `for` loop line by line) and hands it over to optimized parallel libraries on the server. Not only will vectorized code run faster with Star-P®, it will also run faster with MATLAB.

The second point simply reflects the fact that transferring data from the client to the server is the slowest link in the Star-P® system. Any operation that involves a distributed variable and a normal MATLAB variable will be executed on the server, and hence, includes transferring the MATLAB variable to the server so that the server has access to it. When the MATLAB variables are scalars, this does not impact the execution time, but when the variables become large it does impact the time it takes to perform the operation.

Note that when combining a distributed and MATLAB variable inside a loop, the MATLAB variable will be sent over to the server for each iteration of the loop.

The third point reflects the fact that changes in the distribution type, say from row to column distributed, costs a small amount of time. This time is a function of the interconnect between the processors and will be larger for slower interconnects. In general, avoiding distribution changes is straightforward and is easily achieved by aligning the distribution types of all variables, i.e. all row distributed or all column distributed.

Maintaining Awareness of Communication Dependencies

Communication between the Star-P® Client and Server

Distributed objects in Star-P® reside on the server system, which is usually a different physical machine from the system where the MATLAB client is running. Thus, whenever data is moved between the client and the server, it involves interprocessor communication, usually across a typical TCP/IP network (Gigabit Ethernet, for instance). While this connection enables the power of the Star-P® server, excessive data transfer between the client and server can cause performance degradation, and thus the default behavior for Star-P® is to leave large data on the server. One typical programming style is to move any needed data to the server at the beginning of the program (via `ppback`, `ppload`, etc.), operate on it repeatedly on the server, then save any necessary results at the end of the program (via `ppfront`, `ppsave`, etc.).

However, there are times when you want to move the data between the client and the server. This communication can be explicit.

```
>> load imagedata a
>> app = ppback(a);
>> bpp = app.*app;
>> b = ppfront(bpp);
>> ppwhos
Your variables are:
  Name       Size             Bytes        Class
  a          1000x1000        8000000      double array
  app        1000x1000p       8000000      ddense array
  b          1000x1000        8000000      double array
  bpp        1000x1000p       8000000      ddense array
Grand total is 4000000 elements using 32000000 bytes
MATLAB has a total of 2000000 elements using 16000000 bytes
Star-P® server has a total of 2000000 elements using 16000000 bytes
```

The `load` command loads data from a file into MATLAB variable(s). The `ppback` command moves the data from the client working space to the Star-P® working space, in this case as a ddense array. Similarly, the `ppfront` command moves data from the Star-P® server working space back to the MATLAB client working space.

```
>> bpp = rand(1000*p);
>> d = max(max(bpp));
>> e = norm(bpp);
>> ppwhos
Your variables are:
  Name       Size             Bytes        Class
  bpp        1000x1000p       8000000      ddense array
  d          1x1              8            double array
  e          1x1              8            double array
Grand total is 1000002 elements using 8000016 bytes
MATLAB has a total of 2 elements using 16 bytes
Star-P® server has a total of 1000000 elements using 8000000 bytes
```

When accessing data from disk, it may be faster to load it directly as distributed array(s) rather than loading it into the client and then moving it via `ppback` (and similarly to save it directly as distributed arrays). The `ppload/ppsave` commands are the distributed versions of the `load/save` commands. For information on `ppload` and `ppsave`, see "The ppload and ppsave Star-P® Commands".

Implicit Communication

The communication between the client and the server can also be implicit. The most frequent cases of this communication pattern are the call(s) that are made to the Star-P® server for operations on distributed data. While attention has been paid to optimizing these calls, making too many of them will slow down your program. The best approach to minimizing the

number of calls is to operate on whole arrays and minimize the use of control structures such as `for` and `while`, with operators that match what you want to achieve.

Another type of implicit communication is done via reduction operations, which reduce the dimensionality of arrays, often to a single data element, or other operators which produce only a scalar.

```
>> d = max(max(bpp));
>> e = norm(bpp);
>> ppwhos
Your variables are:
   Name        Size           Bytes      Class
   bpp         100x100p       80000      ddense array
   d           1x1            8          double array
   e           1x1            8          double array
```

One of the motivations behind the design of Star-P® was to allow larger problems to be tackled than was possible on a single-processor MATLAB session. Because these problems often involve large data (i.e., too big to fit on the MATLAB client), and because of the possibility of performance issues mentioned above, Star-P®'s default behavior is to avoid moving data between the client and the server. Indeed, given the memory sizes of parallel servers compared to client systems (usually desktops or laptops), in general it will be impractical to move large arrays from the server to the client. The exception to this rule arises when operations on the server result in scalar output, in which case the scalar value will automatically be brought to the client.

Because of this bias against unnecessary client-server communication, some Star-P® behavior is different from MATLAB. For instance, if a command ends without a final semicolon, MATLAB will print out the resulting array.

```
>> f = rand(8,8)
f =
    0.4838    0.1520    0.1996    0.7267    0.4563    0.7669    0.3624    0.7185
    0.5923    0.5584    0.1937    0.4047    0.2911    0.2298    0.2460    0.8987
    0.7036    0.2819    0.4815    0.3219    0.0787    0.4983    0.9179    0.8907
    0.8828    0.1345    0.1551    0.3135    0.4714    0.7376    0.1811    0.8055
    0.1802    0.1512    0.2509    0.2147    0.9806    0.0915    0.6026    0.8420
    0.6950    0.4017    0.5268    0.0104    0.9427    0.0030    0.1507    0.3435
    0.9811    0.0213    0.4433    0.7595    0.8324    0.7831    0.4493    0.2497
    0.1848    0.7306    0.0034    0.5078    0.7174    0.1684    0.6500    0.8098
    0.0904    0.5250    0.2795    0.5770    0.5986    0.0795    0.3651    0.4867
    0.4757    0.5727    0.9461    0.6291    0.4177    0.8044    0.2065    0.3597
```

While this makes good sense for small data sizes, printing out the data sizes possible with Star-P® distributed objects, which often contain hundreds of millions to trillions of elements, would not be useful. Thus the Star-P® behavior for a command lacking a trailing semicolon is to print out the size of the resulting object.

```
>> fpp = rand(8*p,8)
```

```
fpp =
        ddense object: 8p-by-8
```

If you want to see the contents of an array, or a portion of an array, you can display a single element in the obvious way, as follows:

```
>> fpp(1,4)
ans =
    0.2433
```

Alternately, you can move a portion of the array to the client:

```
>> fsub = ppfront(fpp(1:4,:));
>> ppwhos
Your variables are:
  Name        Size       Bytes    Class
  ans         1x1        8        double array
  fpp         8px8       512      ddense array
  fsub        4x8        256      double array

Grand total is 97 elements using 776 bytes
MATLAB has a total of 33 elements using 264 bytes
Star-P server has a total of 64 elements using 512 bytes
```

Note: When you call `ppfront` and leave off the final semicolon, MATLAB will print out the whole contents of the array.

Note: Communication can happen implicitly as described in "Mixing Local and Distributed Data".

Communication Among the Processors in the Parallel Server

During operations on the parallel server, communication among processors can happen for a variety of reasons. Users who are focused on fast application development time can probably ignore distribution and communication of data, but those wanting the best performance will want to pay attention to them.

Some operations can be accomplished with no interprocessor communication on the server. For instance, if two arrays are created with the same layout (see details of layouts in "Types of Distributions"), element-wise operators can be done with no communication, as shown in the following example.

```
>> app = rand(100*p,100);
>> bpp = rand(100*p,100);
>> cpp = app +  bpp;
>> dpp = app .* bpp;
>> ppwhos
Your variables are:
  Name        Size           Bytes       Class
  app         100px100       80000       ddense array
  bpp         100px100       80000       ddense array
```

```
  cpp        100px100      80000      ddense array
  dpp        100px100      80000      ddense array
Grand total is 40000 elements using 320000 bytes
MATLAB has a total of 0 elements using 0 bytes
Star-P® server has a total of 40000 elements using 320000 bytes
```

These element-wise operators operate on just one element from each array, and if those elements happen to be on the same processor, no communication occurs. If the elements happen not to be on the same processor, the element-wise operators can cause communication. In the example below, **app** and **epp** are distributed differently, so internally Star-P® redistributes **epp** to the same distribution as **app** before doing the element-wise operations.

```
>> app = rand(100*p,100);
>> epp = rand(100,100*p);
>> fpp = app .* epp;
>> ppwhos
Your variables are:
  Name       Size          Bytes      Class
  app        100px100      80000      ddense array
  epp        100x100p      80000      ddense array
  fpp        100px100      80000      ddense array
Grand total is 30000 elements using 240000 bytes
MATLAB has a total of 0 elements using 0 bytes
Star-P® server has a total of 30000 elements using 240000 bytes
```

Often redistribution cannot be avoided, but for arrays which will be operated on together, it is usually best to give them the same distribution.

Any operator that rearranges data (for example, `sort`, `transpose`, `reshape`, `permute`, `horzcat`, `circshift`, extraction of a submatrix) will typically involve communication on a parallel system. Other operators by definition include communication when executed on distributed arrays. For example, multiplication of two matrices requires, for each row and column, multiplication of each element of the row by the corresponding element of the column and then taking the summation of those results. Similarly, a multi-dimensional FFT is often implemented by executing the FFT in one dimension, transposing the data, and then executing the FFT in another dimension. Some operators require communication, in the general case, because of the layout of data in Star-P®. For instance, the find operator returns a distributed dense array (column vector) of the nonzero elements in a distributed array. Column vectors in Star-P® contain an equal number of elements per processor for as many processors as can be fully filled, with any remainder in the high-numbered processors. Thus the `find` operator must take the result values and densely pack them into the result array. In general, this requires interprocessor communication. For the same reason, creating a submatrix by indexing into a distributed array also requires communication.

As a programmer, you may want to be aware of the communication implicit in various operators, but only in rare cases would the communication patterns of a well-vectorized code make you choose one operator over another. The performance cost of interprocessor communication will be heavily application dependent, and also dependent on the strength of

the interconnect of the parallel server. For high communication problems, a tightly integrated system, such as an SGI Altix system, will provide the best performance.

Enhanced Performance Profiling in Star-P®

If you are using Star-P®, it is probably because you want to achieve maximum performance from your MATLAB program. That is, you are interested in making your program run as quickly as possible, while still returning correct results. Since Star-P® is a client/server program, correctly distributing your processing tasks between client and server is instrumental in obtaining best performance. Also, calling the right functions to achieve your goals is important to obtaining good run times.

Knowing which functions are invoked, where they are running, how many times they are called, and how long they run before completion is information you can use while optimizing your program for best performance. Therefore, Star-P® provides several profiling facilities to help you wring maximum performance out of your program. These facilities include:

- MATLAB's `profile` and Star-P®'s `ppprofile`, which provide run-time profiling of your program's execution. This information includes statistics about execution time, number and name of sub-function calls, and other execution tracing information. “Profile” tracks program activity on the client, and “ppprofile” tracks activity on the server.
- MATLAB's `tic/toc` and Star-P®'s `pptic/pptoc`, which report the time elapsed on the client (`tic/to`) and the server (`pptic/pptoc`) between the `tic/pptic` call and the `toc/pptoc` call.
- Star-P®'s unique `ppperf` function. `ppperf` provides fine-grained profiling of compute activity on both the client and the server together. It pays close attention to the time required to perform computational tasks, and it also tracks communication between client and server over the network. The vision behind `ppperf` is to provide you a top-level view of what your program is doing as it runs your calculation. Using the information provided by `ppperf`, you can identify program choke points, isolate excessive client/server communication, see what functions are invoked on both client and server, and determine how long each function takes to finish. This information can be invaluable when debugging or optimizing a Star-P® application.

The rest of this section describes the use of `ppperf` in investigating your code.

Using `ppperf`

Usage of Star-P®'s profiling tool is loosely based upon MATLAB's `profile` functionality. If you are used to code profiling using MATLAB's `profile`, then Star-P®'s `ppperf` will feel comfortable to you.

To use `ppperf`, you should first have a mental model of what `ppperf` is doing. Figure 5-1 is a simplified diagram showing the major software components at work when you execute a function called `myfunc` on the parallel supercomputer. You, the user, interacts with MATLAB on your local PC. When you execute `myfunc`:

- MATLAB passes control to the Star-P® client software, which in turn sends it to the Star-P® server software, which evaluates your function using the appropriate numerical library.
- Then, the Star-P® server passes the result to the Star-P® client, which sends it up to MATLAB, which displays the result to you in your MATLAB session. Meanwhile, performance data is recorded at several points within the system.

Figure 5-1 Major Software Components At Work

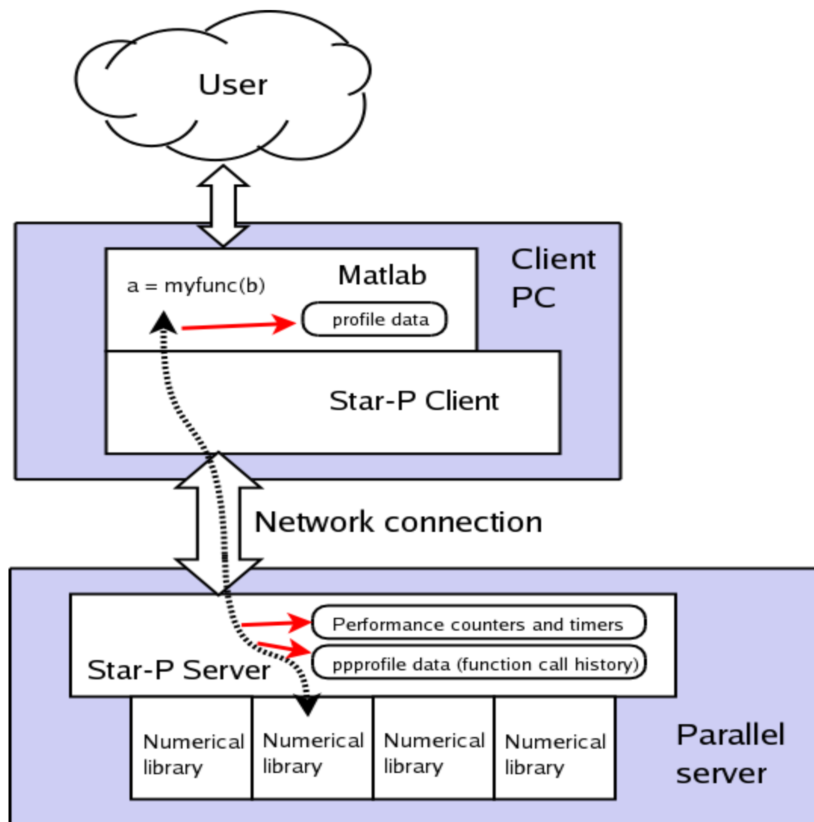


Figure 5-1 is a conceptual picture of what happens when you invoke `myfunc()` in the MATLAB client. Star-P® software passes the function call down to the appropriate numerical library on the server, and passes the returned results back to the MATLAB session (thick dotted black line in Figure 5-1).

Three different sets of performance data are gathered: One set in MATLAB on the client, and two sets in the Star-P® server (red arrows). Later, when the user types "`ppperf report`", the performance data is gathered into a table living on the client (blue arrows).

With this picture in mind, performance monitoring occurs in three places:

1. MATLAB records performance data on the client PC. It makes this data available to you using its `profile` facility. This data is also accessible using `ppperf` with the appropriate flag set.
2. Star-P® records performance data related to the sub-functions called by your function. Much of this data is related to the calling history of sub-functions you invoke. This data is available to you using Star-P®'s `ppprofile` facility, as well as using `ppperf` with the appropriate flag set.
3. Star-P® also keeps timers and counters related to supercomputer resource utilization. This data is uniquely accessible using Star-P®'s `ppperf` feature.

Gathering these three categories of performance parameters is suggested in the figure by the red arrows, which indicate collection of performance data into the associated data structures and storage tables.

After your program has run, you may request a report showing all recorded performance data by issuing the command `ppperf report`. This command brings all the performance data scattered around the client and server into a table living on the client. This is suggested by the blue arrows shown in the figure. Once the data is gathered into a table on the client, the table is then displayed to the user.

Another important picture to visualize when profiling your code is to understand how the client and the server interact. Under Star-P®, the client and the server process your computation using a "ping-pong" mode. That is, while Star-P® is performing your calculation, the client does some work while the server sits idle, and then the server does some work while the client is idle, then the client does work and the server sits idle, and so on. Each time a work hand-off occurs, a burst of network activity occurs as data is exchanged between client and server. Keep this work flow in mind as you examine the data generated by `ppperf`.

Using `ppperf` is simple. First, you initiate performance monitoring. This means that you tell Star-P® to clear any old performance data stored on the server, and start the performance counters and timers afresh. Next you run your program. When your program is done, you fetch the performance data from the server and display it. Finally, assuming you are done with performance monitoring, you turn off the monitoring facility. Here's an example sequence of commands you could enter in your Star-P® session:

```
ppperf o2                         % Initiate performance monitoring
baz  = my_function1(foo, bar);    % Function running on server
woof = my_function2(foo, baz);    % Another function running on server
my_function3(woof);               % A third function running on server
ppperf report                     % Get and display performance data
ppperf clear                      % Clear statistics table
```

`ppperf` will accumulate performance statistics until you turn it off using one of `ppperf report`, `ppperf off`, or `ppperf clear`. The distinction between these commands lies in whether they erase the statistics table. As a general rule, `ppperf`'s subcommands will behave similarly to the analogous subcommands of MATLAB's `profile` function.

Gathering performance statistics

- **`ppperf`** `on | o1`
  `ppperf` `o2 | o3 <seconds> -` This command starts the performance monitoring process and initializes the results table. It is always the first command issued when you want to profile your code's execution. You may select one of three levels of profiling. The profiling level (1, 2, 3) is prefixed with the lower case letter 'o'. Consult Figure 5-1 to see the three different types of profiling data measured, and where they are logged. The different profiling levels are invoked using these flags:
- `on` - This flag is a synonym for `o1`. That is, `ppperf on` is equivalent to `ppperf o1`.
- `o1` - Returns the data stored in the performance counters and timers on the server.
- `o2` - Returns the data in the server's performance counters and timers. It also returns the server-side function call history data typically returned by `ppprofile`.
- `o3` - Returns the data stored in the server's performance counters and timers. It also returns the server-side function call history data. Finally, it also returns the client-side profiling data which is gathered by MATLAB's `profile` function.

- After the profiling level, you may optionally specify the update interval (in seconds) for statistics gathering (denoted by `<seconds>` above). The update interval must be an integer. The update interval is optional; if you don't specify this parameter, update interval sampling is not done.
- **`ppperf`** `report` - This command stops performance statistics gathering on the server, brings the performance data to the client, and displays them. Use this command when you are done gathering statistics on your program and want to see the results.
- **`ppperf`** `clear` - Turns off profiling and clears the results table.
- **`ppperf`** `off` - Turns off the performance monitoring process, but leaves the results table alone. Use this command if you want to perform some work without gathering statistics.

Displaying performance statistics

Star-P®'s `ppperf` facility supports two methods to display performance statistics: textual and graphical. Text reports are covered in this section and graphical output is covered in "Using ppperf's graphical mode".

To see profiling results:

- First turn on profiling using `ppperf on`, then run your program.
- Then issue the command `ppperf report`, or `ppperf report detail`.
    - **ppperf** `report` will emit a long text report detailing the resources used by your program while it ran.
    - **ppperf** `report detail` will return a more extensive report. Specifically, `ppperf report detail` will display run statistics gathered on each compute node on the server (`ppperf report` displays statistics for the server as a whole).

Here's an example `ppperf` run. First, we'll look at the program being profiled. It is called `SumDifferences_loop.m`. It calculates the RMS deviation from one point to the next in a 1000 element random vector.

```
% This example will calculate the RMS point-to-point
% deviation of one random point to the next.

xpp = rand(1000*p, 1);
n=length(xpp)-1;

tic
   dx_sum = 0;
   for i = 1:n
      dx = xpp(i+1)-xpp(i);
      dx_sum = dx_sum + dx^2;
   end
   dx_sum = sqrt(dx_sum);
toc
fprintf('dx_sum = %f\n', dx_sum);
```

This is a particularly bad program for Star-P®, since it involves using a `for` loop to perform a simple sum. It is easy to create a vectorized version of this program whose run time is perhaps 100 times faster. Nonetheless, this code provides a very interesting example for `ppperf` profiling since it demonstrates many of the things you can learn by running `ppperf` on your code.

Here's the `ppperf` run:

```
>> ppperf on
Start MATLAB/Star-P® Performance Metrics
>> SumDifferences_loop
Elapsed time is 12.254049 seconds.
dx_sum = 12.304663
>> ppperf report
==============================================================================
MATLAB/Star-P® Performance Metrics
Date:    17-May-2007 18:11:01
Client:  my_client_machine_address.com
```

```
Server:  my_server
Elapsed: 32 seconds
------------------------------------------------------------------------------

Performance Time Measurement
------------------------------------------------------------------------------
  count        min        max       mean       time  metric
   2001     0.0029     5.5610     0.0082    16.4061  Client Time
   2001     0.0003     0.0061     0.0004     0.8222  Network Time
   2001     0.0000     0.0003     0.0000     0.0432  Client2Server
   2001     0.0000     0.0001     0.0000     0.0334  Server2Client
   2001     0.0073     0.0226     0.0074    14.7372  Server Time
   2001     0.0073     0.0226     0.0074    14.4858  Command
   2001     0.0000     0.0004     0.0001     0.2128  Command Execute
   1998     0.0000     0.0001     0.0000     0.0384  Command Execute Move
   2001     0.0000     0.0000     0.0000     0.0176  Command EStatus
      1     0.0001     0.0001     0.0001     0.0001  Command Execute Create
      3     0.0000     0.0000     0.0000     0.0000  Command Execute Misc

Performance Process Measurement
------------------------------------------------------------------------------
                                                       value   metric
                                                     34.0630   Starp Real Time
                                                     36.4111   Starp Sys Time
                                                     78.6709   Starp User Time

Star-P® Profiling
------------------------------------------------------------------------------
function                  calls       time  avg time    %calls     %time
ppdense_viewelement        1998    16.9636 0.0083551   99.8501   99.6233
ppbase_setoption              1   0.030958  0.030958  0.049975   0.18475
ppdense_rand                  1   0.023654  0.023654  0.049975   0.14116
ppbase_profile_onoff          1   0.008517  0.008517  0.049975  0.050827
Total                      2001    16.7567 0.0083742
>>
```

Interpretation of `ppperf`'s output

Now that we've seen the output generated by a typical `ppperf` run, the question is: What does all that data mean? Let's look at each section generated by `ppperf`.

The preamble

The preamble provides basic information about the performance run just completed. Here's the preamble from the above run:

```
MATLAB/Star-P® Performance Metrics
Date:    17-May-2007 18:11:01
Client:  my_client_machine_address.com
Server:  my_server
Elapsed: 32 seconds
```

The preamble provides the following information:

- The date of the performance run,
- the names of the client and server computers, and
- the total elapsed time.

If you are running `ppperf` interactively (for example, typing each command into the Star-P® prompt), then the elapsed time is the time duration between when you typed in `ppperf o2` and when you typed in `ppperf report`. If you wait for 20 seconds before running your function, the additional 20 seconds of idle time will be incorporated into the reported elapsed time.

Performance Time Measurement

This section provides information about how the two computers (client and server) spent their time on your calculation.

Remember that Star-P® operates in a ping-pong mode. The client does some work while the server idles, then the server does some work while the client idles, and so on, until your computation is done.

Also, every time there is a hand-off of work between client and server, a burst of network activity occurs. The performance time measurement shows you how many times each component was active in your program, the min, max, and mean time it was active (in seconds), and the total time required by each component to do its job. Here's the "Performance Time Measurement" section of the report shown earlier:

```
Performance Time Measurement
-----------------------------------------------------------------------------
  count        min        max       mean       time  metric
   2001     0.0029     5.5610     0.0082    16.4061  Client Time
   2001     0.0003     0.0061     0.0004     0.8222  Network Time
   2001     0.0000     0.0003     0.0000     0.0432  Client2Server
   2001     0.0000     0.0001     0.0000     0.0334  Server2Client
   2001     0.0073     0.0226     0.0074    14.7372  Server Time
   2001     0.0073     0.0226     0.0074    14.4858  Command
   2001     0.0000     0.0004     0.0001     0.2128  Command Execute
   1998     0.0000     0.0001     0.0000     0.0384  Command Execute Move
   2001     0.0000     0.0000     0.0000     0.0176  Command EStatus
      1     0.0001     0.0001     0.0001     0.0001  Command Execute Create
      3     0.0000     0.0000     0.0000     0.0000  Command Execute Misc
```

As you can see, the client performed a chunk of work 2001 times. Also, it consumed by far the majority of the elapsed time. Since the program `SumDifferences_loop.m` involves a `for` loop iterated 1000 times, it appears that the client performed two tasks for each loop iteration -- plus one task at the end of the loop -- giving rise to the activity count of 2001.

The network was active 2001 times, which reflects the fact that upon each iteration of the `for` loop the client was required to perform two tasks. Also, the large amount of time communicating on the network signals that this program spent too much time communicating.

Finally, the server was active 2001 times, but each burst of activity lasted at most a few milliseconds. This indicates that the server was very underutilized by this program. Since the whole point of using Star-P® is to effectively harness and use the power of the supercomputer server, this program, `SumDifferences_loop.m`, clearly does an inefficient job of exploiting the potential of Star-P®. Of course, this is expected, since the program is essentially a big `for` loop, which is known to be a slow and inefficient way to implement this computation. Later, we'll look at a vectorized version of the same computation.

Performance Process Measurement

One of the most interesting things you can learn from `ppperf` is the amount of time spent in the various software subcomponents (daemons or libraries) running on the server. The “process measurement” results provide this information.

To visualize the meaning of the “process measurement” data, imagine that the server runs a Star-P® server daemon. The daemon manages a set of numerical libraries that are used to evaluate your function. This is shown schematically in Figure 5-1.

When a server call is made, the Star-P® server daemon must spend some time figuring out how to handle your function call. Having done that, it then hands your data to the appropriate function in one of the numerical libraries. Once the function is done executing, it hands the returned data back to the Star-P® server daemon, which in turn sends it to the client. Meanwhile, performance timers and counters are running, measuring the amount of time spent in the Star-P® server daemon, as well as the time spent executing the library function.

Here's a report returned in the default mode, for example, `ppperf report`:

```
Performance Process Measurement
------------------------------------------------------------------------------
                                                     value   metric
                                                   34.0630   Starp Real Time
                                                   36.4111   Starp Sys Time
                                                   78.6709   Starp User Time
```

There are several things to note here:

- In this example the only process that ran was the “Starp” process. The “Starp” process is the Star-P® server daemon, which manages your computation on the server side. Depending upon the details of your calculation -- and specifically which numerical engines it invokes on the server -- you may see other processes listed in this section alongside “Starp”. Rerun the example to get Octave times.
- Three times are listed for each process:

1. Real Time - The wall clock time spent by this process during the execution of your function. This can be greater or less than User Time depending on the application.
2. Sys Time - This is the CPU time spent executing “kernel space” code on the server at the behest of your program. This will likely be smaller than the real, wall clock time since the server is multitasking many jobs at once. This can be greater or less than User Time depending on the application.
3. User Time - This is the CPU time spent executing “user space” code on the server at the request of your program. This will likely be smaller than the real, wall clock time since the server is multitasking many jobs at once.

- The elapsed time reported is the time since the `ppperf` command was invoked, not since the process started.

If you issue the command `ppperf report detail`, then `ppperf` will return the time spent broken down by compute node. The “Performance Process Measurement” section is the only one in which `ppperf report detail` will provide additional detailed information about your run. Here are the results of a new run of `SumDifferences_loop.m` under `ppperf` showing the difference between the default and the detailed report:

<table>
<tr><td>Default</td><td><pre>Performance Process Measurement
----
                                      value metric
                                    34.0630 Starp Real Time
                                    36.4111 Starp Sys Time
                                    78.6709 Starp User Time</pre></td></tr>
<tr><td>Detailed</td><td><pre>Performance Process Measurement
----
                                      value metric
                                    34.0630 Starp Real Time
                                    34.0630 Starp Real Time(0)
                                    33.1740 Starp Real Time(1)
                                    33.2720 Starp Real Time(2)
                                    32.2670 Starp Real Time(3)
                                    36.4111 Starp Sys Time
                                     6.6670 Starp Sys Time(0)
                                     9.9482 Starp Sys Time(1)
                                    10.0010 Starp Sys Time(2)
                                     9.7949 Starp Sys Time(3)
                                    78.6709 Starp User Time
                                     9.8037 Starp User Time(0)
                                    23.2090 Starp User Time(1)
                                    23.2539 Starp User Time(2)
                                    22.4043 Starp User Time(3)</pre></td></tr>
</table>

As you can see, using `ppperf report detail` shows a breakdown of time spent on each compute node. You can use this information to help locate a node which might be particularly slow, either due to an excessively long task-parallel computation, or perhaps because it has

other activity running on it. You can use this information to help balance the load across all compute nodes on your machine.

Star-P® Profiling

This section lists all functions called on the server while running your program. It tabulates the number of invocations as well as the average time spent in each function, along with some other performance metrics.

The functions listed in the Star-P® Profiling section are exclusively built-in Star-P® functions. The built-in Star-P® functions are typically named something like `ppdense_foo` or `ppbase_bar` to distinguish them from the names you might give to your functions. In general, these Star-P® built-in functions are not available to you to run, and you cannot use `help ppdense_foo` at the command line to get more information about the function. However, the functions are usually named in a logical way so that you can make an educated guess about what the functions are doing.

The information provided in the Star-P® Profiling section is particularly useful when tweaking your code for best performance, since it allows you to identify which functions consume the majority of your compute time. You can focus your optimization efforts on improving the functions which consume the most time, or at least optimizing the number of times each function is invoked.

Here's the Star-P® Profiling section copied from the above run of `SumDifferences_loop.m`:

```
Star-P® Profiling
-----------------------------------------------------------------------------
function                   calls      time   avg time    %calls     %time
ppdense_viewelement         1998   16.9636 0.0083551   99.8501   99.6233
ppbase_setoption               1  0.030958  0.030958  0.049975   0.18475
ppdense_rand                   1  0.023654  0.023654  0.049975   0.14116
ppbase_profile_onoff           1  0.008517  0.008517  0.049975  0.050827
Total                       2001   16.7567 0.0083742
```

As you can see, the Star-P® built-in function `ppdense_viewelement` consumed the majority of the compute time during the run. But what is `ppdense_viewelement`? This function is invoked each time an array element (for example, a scalar) must be returned from the server to the client for processing. Our program `SumDifferences_loop.m` iterates over all elements in the vector and sums them, as follows:

```
n = length(xpp)-1;
for i = 1:n
   dx = xpp(i+1)-xpp(i);
   dx_sum = dx_sum + dx^2;
end
```

Recall that under Star-P®, scalar variables always live on the client. Therefore, when this program requests the values **xpp**(i+1) and **xpp**(i), Star-P® goes to the server, gets the individual elements out of the vector, and brings them to the client where they are added. That's why `ppdense_viewelement` is invoked 1998 times.

Lessons Learned

When using `ppperf` on your code, look for the following things:

- **Time spent on client vs. server.** Is the program running on either client and server according to your expectations? That is, if you think you have exported a calculation to the server, but you see significant activity on the client, this is a signal that your program isn't fully optimized.
- **Excessive network time.** Monitor your program's network time. Keep in mind that a 100mb or 1Gig network link between your client and server can transport many millions of bytes in well under a second. Therefore, if your network time is not commensurate with transferring a small number of bytes (depending upon your program's structure), you may be paying a communication time penalty due to `for` loops or an unexpected data transfer.
- **Number of times the client runs a task.** If your client runs a short task many times, or there is significant “ping-ponging” between the client and the server, your program is causing too much client/server communication. Find a way to keep **all** the computation on the server. Perhaps you need to vectorize more?
- **Excessive time spent on one or two functions.** If your program spends most of its time running one particular function, you should probably focus attention on why that is the case. If the one function was written by you, then it is a good candidate for further optimization.
- **Many calls to `ppdense_viewelement`.** `ppdense_viewelement` transfers scalar data from server to client. If this function dominates your function usage, it is a signal that you need to vectorize your code.

Using `ppperf`'s graphical mode

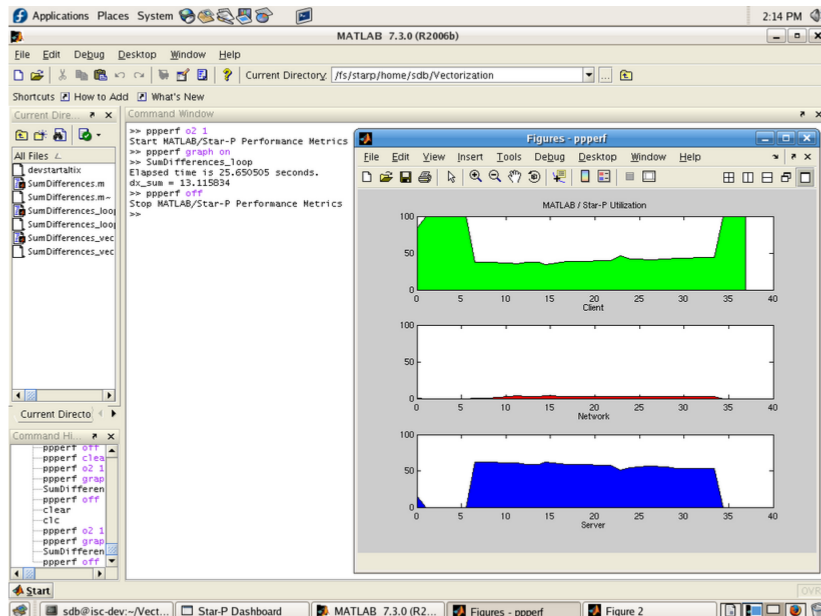
Besides providing you a text report, `ppperf` can also show a graph of client and server activity. This information can be useful if you want to see exactly when your computation was passed from client to server and back again.

You invoke `ppperf` graphical mode in much the same way as you get a text report. The particular command sequence looks like this:

```
ppperf o2 1
ppperf graph on
my_function
ppperf off
```

A screenshot showing the results of a graphical profiling session using `ppperf` is presented in Figure 5-2.

Figure 5-2 Graphical output of `ppperf`[2]



Graphical output of `ppperf`, showing activity on client (top), network (middle), and server (bottom). The units are percent of activity, where 100% means that the particular Star-P® subcomponent was active 100% of the time over the last measurement interval. Remember that this is not a measure of CPU utilization! Rather, the graph shows which Star-P® subcomponent is active (or has control over) performing your calculation.

Here are some things to keep in mind when using `ppperf`'s graphical mode:

- The graphical display is only available in conjunction with the `o2` and `o3` statistics gathering levels.
- When you turn on performance logging for graphical display, you must specify the logging interval. The default value (1 second for a text report) does not apply to graphical output. If you neglect to specify a logging interval, Star-P® will give you a "No samples" error. Again, the logging interval must be an integer; the units are seconds. Accordingly, in the above example the logging interval is explicitly set to one second.

2. Illustration of the MATLAB® Desktop IDE by The MathWorks.

- You can get a real-time running update of activity on the client and server by turning on graphics mode before invoking your function, as in the example shown in Figure 5-2. Your graphic display will be updated whenever the client has control over your computation, and is available to update the graphics window. Note that if you kick off a long, server-side calculation (for example, using `ppeval`), the `ppperf` graphic won't update until the server is done working, and passes control to the client again. The same is true for the client side when it is in a CPU-bound execution.
- After the run is over, turn off the performance logging using `ppperf off`, as shown in Figure 5-2. If you do not turn off logging, then the performance graphic will continue to update, and the region of interest - the portion of the graph showing your computation running - will compress within the available space, making it hard to read.
- If `ppperf` data gathering is not active when you invoke `ppperf graph on` (for example, you have issued a `ppperf off` command), then `ppperf` will plot the static results contained in the performance results table. If you do not have any data in the performance results table, Star-P® will give you a “No samples” error.
- Since performance samples are made at regular, but large intervals, the client and server utilization graphs - like that shown in Figure 5-2 - represent averages over the sample interval. As you know, Star-P® ping-pongs control between client and server; while one machine is busy processing, the other is essentially idle. In the program `SumDifferences_loop.m`, control is passed between the client and the server about 2000 times. However, the update interval is `q` seconds. Therefore, `ppperf` cannot graph each and every time control is passed between client and server. Rather, it plots the average time spent on each, over each 1 second measurement intervals.
- When both client and server are idle, `ppperf`'s graphing function will indicate 100% client utilization. This is because the graph itself is not a graph of CPU loading. Rather, it is simply an indication of which computer is currently in control of your computation. When both computers are idle, waiting for you to type something, then the client is the computer who is in control. Therefore, it's graph will indicate 100% utilization.

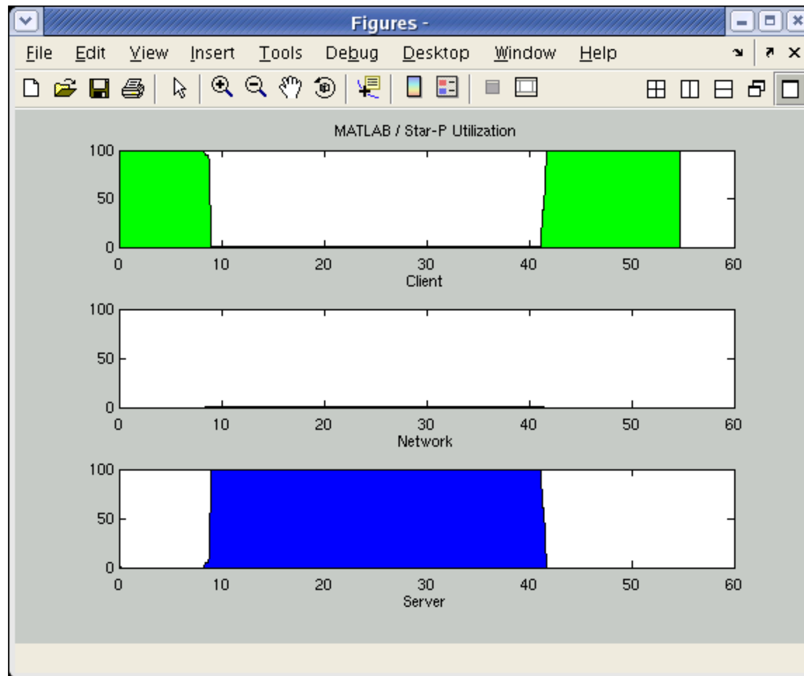
You might wonder, “For what is `ppperf`'s graphics facility useful?” It can be used for quick, visual identification of situations where there is too much communication between client and server during the course of a computation.

This is signaled by graphs showing compute activity on both the client and the server at the same time. A better computation is shown in the graph in Figure 5-3. This particular computation involved computing the Mandelbrot set using a task-parallel algorithm. In this case, the client initialized some variables, and then passed control to the server. The server performed the bulk of the computation over a period of about 30 seconds, and then returned control to the client.

The resulting graph shows that 100% of the computation takes place on either the client or the server, depending upon time. At no time does it appear that the computation is shared

between client and server. Therefore, this computation is not bogged down with client/server communication.

Figure 5-3 Graphical output from `ppperf` showing a different Star-P® session.



In this example, a `ppeval` call was used to compute the Mandelbrot set. The `ppeval` call lasted about 30 seconds, during which time the server was working on the computation 100% of the time, while the client idled.

Any successful computation performed using Star-P® should show a similar graph: 100% of the computation should take place on either the client or the server for long periods of time (seconds or longer). Control of the computation may bounce back and forth between client and server, but certainly not frequently, and at no time should your program appear to share the computation between client and server.

This highlights another use of `ppperf`'s graphing facility: It can alert you to situations where you think a portion of a calculation is taking place on the server, but it is actually running on the client. That is, since `ppperf` shows you where the computation is happening, it can help you verify that your program is actually doing what you think it should be doing.

Finally, `ppperf`'s graphic mode can quickly show you if you are spending too much time running on the client. Since you likely purchased Star-P® to help export computations to the server, if you find that a lot of your compute time is spent on the client, then you probably need to modify your program so that more of the computation is performed on the server.

Lessons Learned

- Use `ppperf`'s graphic facility as a quick way to see if your program requires too much client/server communication. This scenario is signaled by performance graphs showing computation is shared between client and server.
- `ppperf`'s graphic facility can also verify that your code is actually doing what you think it should be doing.
- The graphic facility can also show you if you are not getting enough use from your server. In the best case, you should see client activity at the beginning and end of your program run, and server activity in the middle for the bulk of its run.

Using `ppperf` to Eliminate Performance Bottlenecks

To illustrate the utility of `ppperf` when optimizing your code's performance, let's look at an example finite element method calculation (FEM). FEM problems typically involve manipulating large matrices, and are computation intensive.

Therefore, FEM problems are well-adapted to solution using Star-P®. In particular, data-parallel matrix manipulation is a logical use of Star-P® for FEM problems. This means that we want to structure the program so that all large matrices live on, and are processed on the server HPC.

The example code shown below was originally written solely in MATLAB, with no parallel extensions. The program consists of four parts:

1. Data read-in and initialization,
2. Building the stiffness matrix,
3. Solving the set of linear equations, and
4. Post-processing and solution visualization.

```
%  Initial, non-parallelized version.
clear;

disp('load grid file')
tic;
    load('/home/FEM/fem/mediumgrid.mat');
toc

% set up vertices for looping:
pi = points(connec(:,1),:);
pj = points(connec(:,2),:);
pm = points(connec(:,3),:);

% Each element (triangle) results into a [6x6]
% submatrix of K
nelem = size(connec,1);      % number of elements in the mesh
npoints = size(points,1);    % number of point that make up the mesh
nK = npoints*2;              % size of the stiffness matrix

% Set up the global variables used in the calculation of the stiffness
```

```
matrix.
tocnp = 1;
disp('Building stiffnessmatrix');
tic
    % Set up global variables
    set_globals(1);
    % allocate row, column, value arrays
    II = zeros(6,6,nelem);
    JJ = II;
    KK = II;
    % Get stiffnessmatrix contribution for each mesh element
    for i = 1:nelem
        [II(:,:,i), JJ(:,:,i), KK(:,:,i)] = ...
            get_k_matrix(pi(i,:),pj(i,:),pm(i,:),connec(i,:));
    end
toc;

disp('Create sparse matrix');
tic;
    % Setup stiffness matrix as a sparse matrix:
    K = sparse(II(:),JJ(:),KK(:),nK,nK);
toc;

%
% Now we need to set the boundary conditions.
% For the boundary condititions we require that the
% vertices on the bottom stay fixed.
%
% find vertices on the bottom
disp('Apply boundary conditions');
tic;
    nbase = length(ipoints_base);
    K(2*ipoints_base-1,:) = 0.0;
    K(2*ipoints_base,:) = 0.0;
    K(:,2*ipoints_base-1) = 0.0;
    K(:,2*ipoints_base) = 0.0;
    K(2*ipoints_base-1,2*ipoints_base-1) = speye(nbase);
    K(2*ipoints_base,2*ipoints_base)     = speye(nbase);
toc;

% Make a force vector. Apply force along the top only.
% and only in the +x direction
disp('Make force vector');
tic;
    ntop = length(ipoints_top);
    force = 2;
    F = sparse(ipoints_top,ones(1,ntop),force,nK,1);

    % Just to make sure we have no forces excerted on the bottom
    F(2*ipoints_base-1) = 0.0;
    F(2*ipoints_base) = 0.0;
toc;
```

```
% Solve the system of linear equations.
disp('Solve system');
tic;
    displacement = K\F;
toc;

% Calculate new point positions based on old ones and the displacement
% from the FEM analysis
tic;
    i= 1:npoints;
    new_points(i,1) = points(i,1)+displacement(2*i-1);
    new_points(i,2) = points(i,2)+displacement(2*i);
toc

% Plot the results
clf;
subplot(1,2,1)
h=trimesh(connec,points(:,1),points(:,2),zeros(size(points,1),1));
set(h,'EdgeColor','k');
view(2),axis equal,axis off,drawnow;

subplot(1,2,2)
h=trimesh(connec,new_points(:,1),new_points(:,2),zeros(size(new_points,1),1));
set(h,'EdgeColor','k');
view(2),axis equal,axis off,drawnow;
```

It might be tempting to run this program under `ppperf` to see what happens. However, since it is a pure MATLAB program, it runs exclusively on the client. Therefore, it doesn't generate any server-side performance data, so profiling with `ppperf` does not provide any useful statistics. (Running this program under `ppperf o3` would indeed show performance data, specifically the performance data gathered by MATLAB. Since this is not relevant to Star-P® performance tweaking, we will skip that step here.)

Parallelizing this code under Star-P® involves several, simple steps.

First, since FEM modeling is a logical candidate for data-parallel processing, we will simply read the matrix data into the server (instead of the client) by replacing the `load` statement with `ppload`. This tells Star-P® to read the data from a disk on the server directly into the server's memory. (This implies that you previously copied the data onto the server machine using a separate step, for example, using FTP.) This change is highlighted in blue in the listing below.

Once the data is read into the server using `ppload`, a couple of other changes become necessary. First, since **`points`**, **`connec`**, **`pi`**, **`pj`**, and **`pm`** are now all server-side variables, the return from `get_k_matrix(`**`pi`**`(i,:),`**`pj`**`(i,:),`**`pm`**`(i,:),`**`connec`**`(i,:))` will also be a server variable. Therefore, we must initialize **`II`**, **`JJ`**, and **`KK`** on the server, instead of the client. Second, since the matrices all live on the server, we must bring them to the client using `ppfront` before plotting.

With these changes, the parallelized FEM program takes the following form:

```
%  Partially parallelized version. -- fem_ppload.m.
clear;

disp('load grid file')
tic;
    ppload('/home/FEM/fem/mediumgrid.mat');   %  --- Star-P®! ---
toc

% set up vertices for looping:
pi = points(connec(:,1),:);
pj = points(connec(:,2),:);
pm = points(connec(:,3),:);

% Each element (triangle) results into a [6x6]
% submatrix of K
nelem = size(connec,1);     % number of elements in the mesh
npoints = size(points,1);   % number of points that make up the mesh
nK = npoints*2;             % size of the stiffness matrix

% Set up the global variables used in the calculation of the stiffness
matrix.
tocnp = 1;
disp('Building stiffnessmatrix');
tic
    % Set up global variables
    set_globals(1);
    % allocate row, column, value arrays
    II = zeros(6,6,nelem*p);                          %  --- Star-P®! ---
    JJ = II;                                          %  lives on server since
    KK = II;                                          %  derived from II
    % Get stiffnessmatrix contribution for each mesh element
    for i = 1:nelem
        [II(:,:,i), JJ(:,:,i), KK(:,:,i)] = ...
            get_k_matrix(pi(i,:),pj(i,:),pm(i,:),connec(i,:));
    end
toc;

disp('Create sparse matrix');
tic;
    % Setup stiffness matrix as a sparse matrix:
    K = sparse(II(:),JJ(:),KK(:),nK,nK);
toc;

%
% Now we need to set the boundary conditions.
%
% For the boundary condititions we require that the
% vertices on the bottom stay fixed.
%
% a) find vertices on the bottom
```

```
%
disp('Apply boundary conditions');
tic;
    nbase = length(ipoints_base);
    K(2*ipoints_base-1,:) = 0.0;
    K(2*ipoints_base,:)   = 0.0;
    K(:,2*ipoints_base-1) = 0.0;
    K(:,2*ipoints_base)   = 0.0;
    K(2*ipoints_base-1,2*ipoints_base-1) = speye(nbase);
    K(2*ipoints_base,2*ipoints_base)     = speye(nbase);
toc;

% Make a force vector. Apply force along the top only.
% and only in the +x direction
disp('Make force vector');
tic;
    ntop = length(ipoints_top);
    force = 2;
    F = sparse(ipoints_top,ones(1,ntop),force,nK,1);

    % Just to make sure we have no forces exerted on the bottom
    F(2*ipoints_base-1) = 0.0;
    F(2*ipoints_base) = 0.0;
toc;

% Solve the system of linear equations.
disp('Solve system');
tic;
    displacement = K\F;
toc;

% Calculate new point positions based on old ones and the displacement
% from the FEM analysis
tic;
    i= 1:npoints;
    new_points(i,1) = points(i,1)+displacement(2*i-1);
    new_points(i,2) = points(i,2)+displacement(2*i);
toc

% Move results to client for plotting
disp('Move results to client and plot them')   %  --- Star-P®! ---
tic;
    new_points = ppfront(new_points);          %  --- Star-P®! ---
    points = ppfront(points);                  %  --- Star-P®! ---
    connec = ppfront(connec);                  %  --- Star-P®! ---
toc;


% Plot the results
clf;
subplot(1,2,1)
h=trimesh(connec,points(:,1),points(:,2),zeros(size(points,1),1));
```

```
set(h,'EdgeColor','k');
view(2),axis equal,axis off,drawnow;

subplot(1,2,2)
h=trimesh(connec,new_points(:,1),new_points(:,2),zeros(size(new_points,1),1));
set(h,'EdgeColor','k');
view(2),axis equal,axis off,drawnow;
```

With these changes, this program is now parallelized, and will execute on the server.

Unfortunately, with the above changes, the FEM program is now extremely slow. Watching the output of `disp` as the program executes, it is clear that the above program gets stuck somehow when it tries to build the stiffness matrix. But what is wrong? To investigate this question, you can use `ppperf` in your Star-P® session as follows:

1. Turn on profiling: `ppperf o2`.
2. Run the program: `fem_ppload`.
3. Let the program run for a while. Then, when you are tired of waiting for it to complete, hit <control>-C.
4. Stop profiling: `ppperf off`.
5. Bring up the `ppperf` graph: `ppperf graph on`.

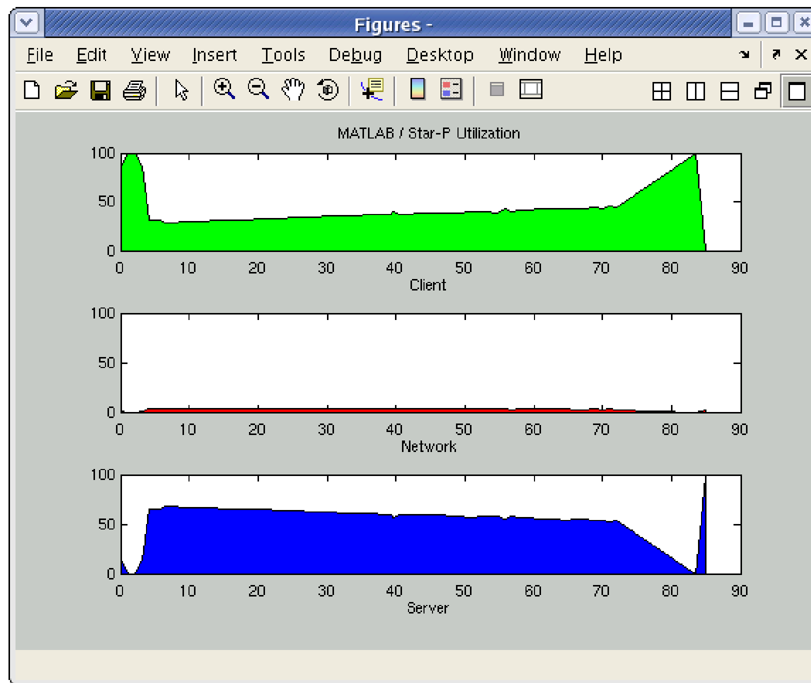
Here's a log showing this sequence of events in a Star-P® session:

```
>> pp_perf o2
Start MATLAB/Star-P® Performance Metrics
>> fem_ppload
load grid file
Elapsed time is 0.100384 seconds.
Building stiffnessmatrix
Error in ==> datenum at 92
n = datenummx(arg1);

Error in ==> now at 16
t = datenum(clock);

Error in ==>
/usr/local/starp-versions/6718/matlab/pp_perfupdate.p>pp_perfupdate at 87

Error in ==>
/usr/local/starp-versions/6718/matlab/cppclient/private/ppprofileupdate.p>pppr
ofileupdate at 63

Error in ==>
/usr/local/starp-versions/6718/matlab/@ddense/ctranspose.p>ctranspose at 4


Error in ==> get_k_matrix at 66
i = j';
Error in ==> fem_ppload at 32
        [II(:,:,i), JJ(:,:,i), KK(:,:,i)] = ...
```

```
>> pp_perf off
Stop MATLAB/Star-P® Performance Metrics
>> pp_perf graph on
```

(The error beginning `Error in ==> datenum at 92` was generated as a consequence of pressing <control>-C while Star-P® was running.) The graph generated by this `ppperf` run is shown in Figure 5-4. The salient feature to note is that both client and server show about 50% utilization for well over 60 seconds (the amount of time this calculation was allowed to run before it was killed). This is a strong signal that too much client-server communication is taking place. Every time computational control is handed off between client and server, a time penalty must be paid since communication between the two machines can last for several milliseconds.

Figure 5-4 Graph generated when running `fem_ppload.m`



Since both client and server show compute activity occurring at the same time, it is likely that control of the program is 'ping-ponging' rapidly back and forth between client and server. This implies a severe performance penalty, since each transfer of control involves a communications delay.

The hypothesis of too much client-server activity is further evidenced by the result of running `ppperf` report, as shown here:

```
>> ppperf report
=============================================================================
MATLAB/Star-P® Performance Metrics
Date:    17-May-2007 18:18:23
Client:  my_client_machine_address.com
```

```
Server:  my_server
Elapsed: 72 seconds
-----------------------------------------------------------------------------
Performance Time Measurement
-----------------------------------------------------------------------------
  count        min     max       mean        time   metric
   6205     0.0014  3.9374     0.0077     29.6009   Client Time
   6205     0.0002  0.1298     0.0004      2.6080   Network Time
   6206     0.0000  0.0025     0.0000      0.1488   Client2Server
   6206     0.0000  0.0043     0.0000      0.1171   Server2Client
   6205     0.0073  0.1317     0.0093     39.9593   Server Time
   6206     0.0073  0.1317     0.0093     39.9717   Command
   6206     0.0072  0.1239     0.0073     27.3850   Command Distribute
   5555     0.0000  0.0528     0.0004      2.1035   Command Execute
    978     0.0000  0.0137     0.0016      1.5488   Command Execute MathOp
   6206     0.0000  0.0014     0.0001      0.4486   Command EStatus
   3423     0.0000  0.0001     0.0001      0.2308   Command Execute Move
    327     0.0002  0.0527     0.0005      0.1577   Command Execute LibOp
    658     0.0000  0.0164     0.0001      0.0470   Command Execute SubsRef
   1312     0.0000  0.0001     0.0000      0.0327   Command Execute Misc
    163     0.0000  0.0002     0.0000      0.0055   Command Execute Redist
Performance Process Measurement
-----------------------------------------------------------------------------
                                                  value metric
                                               109.1050 Starp Real Time
                                               122.6221 Starp Sys Time
                                               258.5273 Starp User Time
Star-P® Profiling
-----------------------------------------------------------------------------
function                 calls     time    avg time     %calls    %time
ppdense_viewelement       3423  29.0718   0.0084931    55.1652  43.2014
pp_dense_ppback            651  11.4339    0.017564    10.4915   16.991
ppdensend_subsasgn_slice   487   8.1989    0.016835     7.8485  12.1838
ppdense_subsref_row        652   7.5395    0.011564    10.5077  11.2039
ppdense_kron               326   3.2293   0.0099058     5.2538   4.7988
ppdense_scalar_op          326   3.1701   0.0097243     5.2538   4.7109
ppdense_transpose          163   2.7824     0.01707     2.6269   4.1348
ppdense_binary_op          163   1.5718   0.0096428     2.6269   2.3357
ppio_loadallvar              1 0.073072    0.073072   0.016116  0.10859
ppdense_subsref_col          3  0.06607    0.022023   0.048348 0.098182
ppdensend_add                1 0.042937    0.042937   0.016116 0.063806
ppdense_subsref_drow         3 0.033902    0.011301   0.048348 0.050379
ppbase_id2ddata              4 0.033217   0.0083042   0.064464 0.049361
ppbase_setoption             1 0.030537    0.030537   0.016116 0.045379
ppbase_profile_onoff         1 0.016158    0.016158   0.016116 0.024011
Total                     6205  67.2935    0.010845
>>
```

There are several important things to note about this report:

- In the “Performance Time Measurement” section, the network time, 2.6 seconds, is quite large. Although 2.6 seconds may seem small as a wall clock time, a high-speed Ethernet connection can transfer tens or hundreds of millions of bytes in one second. Therefore, 2.6 seconds of network time suggests that this program is burning up the client-server network connection by transferring scores of megabytes of data back and forth.
- In the “Performance Bytes Measurement” section, the number of times control was passed from the server to the client was 6025. This is a strong signal that a `for` loop is at work. A `for` loop will cause control of the computation to ping-pong between client and server with each loop iteration.
- In the “Star-P® Profiling” section, the function `ppdense_viewelement` was invoked 3423 times. Recall that this function is called every time a matrix element is moved from the server to the client. Again, since it is invoked so frequently, we can see the `for` loop at work. Also, `ppdense_viewelement` was running almost 43% of the time, suggesting that it is called repeatedly, as if from a `for` loop.

At this point, it is clear that the code suffers from having a `for` loop, which drags down Star-P® performance. Inspecting the code shows that there is indeed a `for` loop involved in initializing the elements of the stiffness matrix, **`II`**, **`JJ`**, and **`KK`**. Optimizing this code obviously requires eliminating the `for` loop. Since the loop involves no dependencies, it can become a task-parallel operation, and can be replaced by a `ppeval` call to perform the job of initializing **`II`**, **`JJ`**, and **`KK`**.

A new version of the program - in which the II, JJ, and KK are initialized in a `ppeval` call - is shown below.

```
%  Fully parallelized version -- fem_ppload_ppeval.m.
clear;

disp('load grid file')
tic;
    ppload('/home/FEM/fem/mediumgrid.mat');  %  --- Star-P®! ---
toc

% set up vertices for looping:
pi = points(connec(:,1),:);
pj = points(connec(:,2),:);
pm = points(connec(:,3),:);

% Each element (triangle) results into a [6x6] submatrix of K
nelem   = size(connec,1);   % number of elements in the mesh
npoints = size(points,1);   % number of point that make up the mesh
nK = npoints*2;             % size of the stiffness matrix

% Set up the global variables used in the calculation of the stiffness matrix.
tocnp = 1;
disp('Building stiffnessmatrix');
```

```
tic
    % Set up global variables.
    % We need to use ppeval in this case since
    % we need to set the global variables on ALL processors.
    out = ppeval('set_globals',1:np);             %  --- Star-P®! ---
    %
    % Get stiffnessmatrix contribution for each mesh element
    % Note that we need to split the arguments along the rows
    [II,JJ,KK] = ppeval('get_k_matrix',split(pi,1),split(pj,1),...
                                       split(pm,1),split(connec,1));
                                                  %  --- Star-P®! ---
toc;

disp('Create sparse matrix');
tic;
    % Setup stiffness matrix as a sparse matrix:
    K = sparse(II(:),JJ(:),KK(:),nK,nK);
toc;

%
% Now we need to set the boundary conditions.
%
% For the boundary conditions we require that the
% vertices on the bottom stay fixed.
%
% a) find vertices on the bottom
%
disp('Apply boundary conditions');
tic;
    nbase = length(ipoints_base);
    K(2*ipoints_base-1,:) = 0.0;
    K(2*ipoints_base,:) = 0.0;
    K(:,2*ipoints_base-1) = 0.0;
    K(:,2*ipoints_base) = 0.0;
    K(2*ipoints_base-1,2*ipoints_base-1) = speye(nbase);
    K(2*ipoints_base,2*ipoints_base)     = speye(nbase);
toc;

% Make a force vector. Apply force along the top only
% and only in the +x direction
disp('Make force vector');
tic;
    ntop = length(ipoints_top);
    force = 2;
    F = sparse(ipoints_top,ones(1,ntop),force,nK,1);

    % Just to make sure we have no forces exerted on the bottom
    F(2*ipoints_base-1) = 0.0;
    F(2*ipoints_base)   = 0.0;
toc;

% Solve the system of linear equations.
```

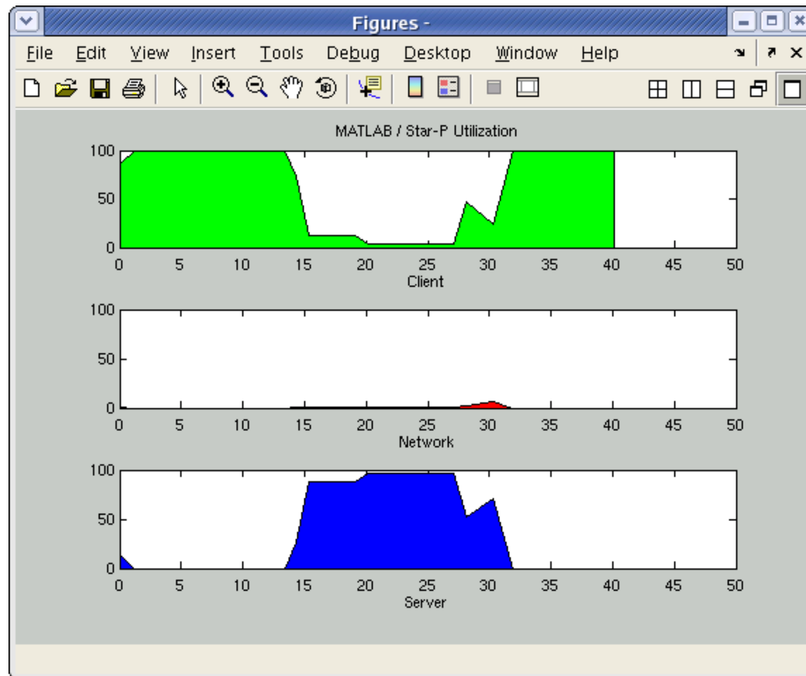
```
disp('Solve system');
tic;
    displacement = K\F;
toc;

% Calculate new point positions based on old ones and the displacement
% from the FEM analysis
tic;
    i= 1:npoints;
    new_points(i,1) = points(i,1)+displacement(2*i-1);
    new_points(i,2) = points(i,2)+displacement(2*i);
toc

% Move results to client for plotting
disp('Move results to client and plot them')   %  --- Star-P®! ---
tic;
    new_points = ppfront(new_points);
    points = ppfront(points);
    connec = ppfront(connec);
toc;

% Plot the results
clf;
subplot(1,2,1)
h=trimesh(connec,points(:,1),points(:,2),zeros(size(points,1),1));
set(h,'EdgeColor','k');
view(2),axis equal,axis off,drawnow;

subplot(1,2,2)
h=trimesh(connec,new_points(:,1),new_points(:,2),zeros(size(new_points,1),1));
set(h,'EdgeColor','k');
view(2),axis equal,axis off,drawnow;
```

In this version of the program, initializing the stiffness matrix is performed almost totally as a parallel operation on the back-end HPC. The entire program takes under 15 seconds to complete.

Running this program under `ppperf` reveals the reasons for the performance improvement: Control of the computation stays with the server for almost the entire computation. Because the `for` loop in the initialization section has been replaced with `ppeval`, the computation does not need to ping-pong rapidly and repeatedly between client and server. This is shown quite clearly in the graphical result from `ppperf`, shown in Figure 5-5. In that figure, transfer of computational control started with the client, but quickly passed to the server. The computation stayed on the server until the end of the run, when the computation was passed back to the client for results visualization.

It's interesting to note that both client and server seem to have been active towards the end (starting at around 27 seconds); this reflects the fact that `ppfront` was invoked to move the results back to the client after they were generated on the server. As such, this behavior is unavoidable since the results must live on the client in order to graph them.

Figure 5-5 Graph generated when running `fem_ppload_ppeval.m`



At the beginning of the graph, there is a dead time for about 12 seconds. This represents the time elapsed between typing `ppperf o2` and typing the name of the function to run, `fem_ppload_ppeval`. Then, once the program started up, transfer of control passed quickly from client to server, and stayed with the server for most of the execution time.

At the end, client and server were both active (starting at about 27 seconds) as evidenced by the rise in client utilization and the accompanying fall in server utilization. This is likely due to data being transferred back to the client via `ppfront`.

Finally, the difference between this optimized run, and the previous, slow run can be seen in the results returned by `ppperf report`. The report generated by this successful run is shown below:

```
>> ppperf report
=============================================================================
MATLAB/Star-P® Performance Metrics
Date: 17-May-2007 18:21:08
Client:  my_client_machine_address.com
Server:  my_server
Elapsed: 31 seconds
-----------------------------------------------------------------------------
Performance Time Measurement
-----------------------------------------------------------------------------
   count      min       max      mean       time  metric
      90   0.0015   10.9759   0.1944   17.4998  Client Time
      90   0.0003    0.0314   0.0016    0.1471  Network Time
```

```
      90   0.0000   0.0231   0.0004    0.0395  Server2Client
      90   0.0000   0.0059   0.0004    0.0389  Client2Server
      90   0.0073   6.1153   0.1487   13.3871  Server Time
      90   0.0073   6.1153   0.1487   13.3870  Command
      86   0.0000   6.1080   0.1313   11.2893  Command Execute
      10   0.0002   6.1079   1.0281   10.2808  Command Execute Octave
       2   3.9016   3.9017   1.9508    3.9017  Command Octave Init**
       3   0.0001   0.8286   0.2802    0.8406  Command Execute LibOp
      90   0.0072   0.0075   0.0072    0.6504  Command Distribute
       2   0.2233   0.2401   0.1138    0.2275  Command Octave Unpack**
       2   0.0738   0.1567   0.0548    0.1096  Command Octave Xfer2Octave**
       8   0.0002   0.0928   0.0120    0.0957  Command Execute Move
      10   0.0001   0.0109   0.0031    0.0311  Command Execute Redist
       4   0.0055   0.0107   0.0072    0.0290  Command Octave PartGlue
      10   0.0007   0.0027   0.0017    0.0171  Command Execute SubsRef
      90   0.0000   0.0045   0.0002    0.0140  Command EStatus
       2   0.0006   0.0173   0.0065    0.0131  Command Octave WrapperGen**
       2   0.0090   0.0138   0.0057    0.0114  Command Octave Xfer2Starp**
      10   0.0000   0.0023   0.0010    0.0096  Command Execute SubsAsgn
      47   0.0000   0.0018   0.0002    0.0078  Command Execute Misc
      22   0.0001   0.0014   0.0003    0.0062  Command Execute MathOp
       3   0.0001   0.0001   0.0001    0.0002  Command Execute Create
       2   0.0000   0.0000   0.0000    0.0000  Command Octave Execute
Performance Process Measurement
-----------------------------------------------------------------------------
                                                    value   metric
                                                  17.5000   Octave Real Time
                                                   4.9678   Octave Sys Time
                                                  24.4189   Octave User Time
                                                  32.0560   Starp Real Time
                                                  13.7227   Starp Sys Time
                                                  39.4756   Starp User Time
Star-P® Profiling
----------------------------------------------------------------------------
function                  calls       time       avg time    %calls      %time
ppemode_emodecall             2     10.336          5.168    2.2222    75.1343
ppio_loadallvar               1      1.187          1.187    1.1111     8.6285
ppsuperlu_gssvx               1    0.83727        0.83727    1.1111     6.0863
ppdense_scalar_op            15    0.20813       0.013876   16.6667      1.513
ppsparse_construct_rowcol     3    0.16121       0.053737    3.3333     1.1719
ppdense_ppfront               3    0.11116       0.037052    3.3333    0.80802
ppbase_id2ddata               6   0.065029       0.010838    6.6667    0.47271
ppemode_part2densend          3   0.060169       0.020056    3.3333    0.43738
pp_dense_ppback               4   0.059271       0.014818    4.4444    0.43085
ppemode_get_common_sizes      4   0.056725       0.014181    4.4444    0.41235
ppdense_subsref_col           3   0.052731       0.017577    3.3333    0.38331
ppdense_ones                  3   0.050913       0.016971    3.3333     0.3701
ppdense_subsref_rowcol        2   0.043143       0.021572    2.2222    0.31362
ppbase_getoption              3    0.04006       0.013353    3.3333     0.2912
ppdensend_reshape             3   0.037715       0.012572    3.3333    0.27416
ppdense_transpose             4   0.037607      0.0094018    4.4444    0.27337
ppdense_makeRange             2   0.034688       0.017344    2.2222    0.25215
```

```
ppdense_binary_op           2   0.034093      0.017046    2.2222    0.24783
ppdense_subsref_drow        3   0.032474      0.010825    3.3333    0.23606
ppdense_subsasgn_rowcol     2   0.032009      0.016004    2.2222    0.23268
ppdensend_clobber_singlet   3     0.0294        0.0098    3.3333    0.21371
ppsparse_subsasgn_col_s     2   0.028776      0.014388    2.2222    0.20918
ppsparse_subsasgn_rowcol    2   0.026117      0.013059    2.2222    0.18985
ppsparse_reshape            2   0.026024      0.013012    2.2222    0.18917
ppbase_setoption            1   0.025033      0.025033    1.1111    0.18197
ppsparse_subsasgn_row_s     2   0.024614      0.012307    2.2222    0.17892
ppsparse_subsasgn_idx_s     2   0.024431      0.012215    2.2222    0.17759
ppemode_part2dense          1   0.023146      0.023146    1.1111    0.16825
ppdense_subsref_idx         2   0.020242      0.010121    2.2222    0.14714
ppsparse_nnz                1   0.015974      0.015974    1.1111    0.11612
ppbase_profile_onoff        1   0.015961      0.015961    1.1111    0.11602
ppsparse_construct_rowcol   1   0.009912      0.009912    1.1111   0.072052
ppsparse_sparse2full        1   0.009668      0.009668    1.1111   0.070279
Total                      90    13.7567       0.15285
>>
```

The important features to observe in this report are:

- In the “Performance Time Measurement” section, the network time used is 0.15 seconds, which is consistent with reduced client-server communication.
- Under “Performance Process Measurement” a new process has appeared: Octave. This signals that the Octave engine has been invoked on the back-end server during processing. Since the only change made in the code involves the `ppeval` call, the presence of Octave amongst the called processes signals that `ppeval` has executed Octave code while setting up the stiffness matrix -- most likely while executing `get_k_matrix`.
- In the “Star-P® Profiling” section, no function is called more times than any other. This is in contrast to the report generated for `fem_ppload.m` above, in which `ppdense_viewelement` was invoked 3423 times.
- A new function, `ppemode_emodecall`, was invoked only twice, but soaked up 75% of the compute time. This function is the Star-P® function which handles `ppeval` calls on the server side. Since the `ppeval` call which initialized the stiffness matrix soaked up the majority of the wall clock time during this run, it makes sense that `ppemode_emodecall` uses most of the server processing time.

Lessons Learned

- Use `ppperf` in graphics mode to identify sections of code with excessive client-server communication.
- Use `ppperf report` to provide detailed analysis of what resources your program uses while executing.
- For best Star-P® performance, make sure your program is thoroughly vectorized! Avoid using `for` loops over multi-dimensional data whenever you can. Looping over multi-dimensional data necessitates transfer of scalar data between client and server, causing a significant time penalty due to communication overhead.

`ppperf` command summary

<table>
<tr><th>Command</th><th>Explanation</th></tr>
<tr><td>ppperf on | o1
ppperf o2 | o2 <seconds></td><td>This command starts the performance monitoring process and initializes the results table. It is always the first command you issue when you want to profile your code's execution.

The profiling level (1, 2, 3) is prefixed with the lower case letter 'o'. The different profiling levels are invoked using these flags:

• on -- The flag is a synonym for o1. That is, ppperf on is equivalent to ppperf o1.

• o1 -- This returns the data stored in the performance counters and timers on the server.

• o2 -- This returns the data in the server's performance counters and timers. It also returns the server-side function call history data typically returned by ppprofile.

• o3 -- This returns the data stored in the server's performance counters and timers. It also returns the server-side function call history data. Finally, it returns the client-side profiling data which is gathered by MATLAB's "profile" function.

After profiling levels o2 or o3, you may optionally specify the update interval (in seconds) for statistics gathering (denoted by <seconds>). The update interval must be an integer. If you don't specify this parameter, update interval sampling is not done.</td></tr>
</table>

ppperf off	This command turns off the performance monitoring process, but leaves the results table alone. Use this command if you want to perform some work without gathering statistics.
ppperf clear	This command clears the results table, and turns off performance data logging. Use this command if you want to end your performance monitoring session.
ppperf report	This command prints out a large text report providing information about compute resources utilized by your program while it ran.
ppperf report detail	This command prints out a large text report providing information about compute resources utilized by your program while it ran. It provides more detail than “ppperf report”. In particular, it breaks down the process measurement results for each compute node on your parallel server.
ppperf graph on	This command displays a graph showing compute resource utilization on the client, network, and server. If you invoke this command before running your program, it will show you a real-time graph of your computation's activity (as long as control passes to the client). If you invoke this command after executing “`ppperf off`”, it will show you the static graph of compute activity recorded between “`ppperf on`” and “`ppperf off`”. Also, you must specify the update sample when using the `graph` option. • Note that the update interval must be specified to enable the `graph` option.
ppperf graph off	This command closes the performance graph window. It does not affect the compiled performance data table.

UNIX Commands to Monitor the Server

While Star-P® is designed to allow you to program at the level of the MATLAB command language and ignore the details of how your program runs on the HPC server, there are times when you may want to monitor the execution of your program directly on the server.

The following commands will often be useful for monitoring server processes. Execute these commands on the HPC server in a terminal window.

- `top`: This command is often the most useful. See `man top` for details. It displays the most active processes on the system over the previous time interval, and can display all processes or just those of a specific user. It can help you understand if your Star-P® server processes are being executed, if they're using the processors, if they're competing with other processes for the processors, etc. `top` also gives information about the amount of memory your processes are using, and the total amount of memory in use by all processes in the system.
- `ps`: The `ps` command will tell you about your active processes, giving a snapshot similar to the information available via `top`. Since the Star-P® server processes are initiated from an ssh or rsh session, you may find that `ps -lu <yourlogin>` will give you the information you want about your Star-P® processes. In the event that Star-P® processes hang or get disconnected from the client, this can give you the process IDs you need to kill the processes.

INTERACTIVE supercomputing

Chapter 6

Star-P® Functions

This chapter summarizes the Star-P® functions that are not part of standard MATLAB and describes their implementation. It also describes the syntax of the Star-P® functions.

Basic Server Functions Summary

The following table lists the types of functions available.

Function	Description
General Functions	
fseek	The return value FID is a distributed file identifier. Passing this value to the following MATLAB functions: fopen(), fread(), fwrite(), fseek(), frewind() and fclose() will operate on distributed matrices on the server with the same semantics as with regular file id on the client.
np	Returns the number of processes in the server.
p	Creates an instance of a `dlayout` object.
pp	Is useful for users who wish to use the variable `p` for another purpose.
ppbench	Collects basic information about the hardware and software characteristics of your server, and runs low-level performance tests on your server.

Function	Description
ppclear	Clears distributed variables from the server memory.
ppgetoption	Returns the value of Star-P® properties.
ppsetoption	Sets the value of Star-P® properties.
ppgetlog	Get the Star-P® server log file.
ppgetlogpath	Get starpserver log file path.
ppinvoke	Invoke a function contained in a previously loaded user library via the Star-P® Software Development Kit (SDK).
pploadpackage	Load a compiled user library on the server.
ppunloadpackage	Unload a user library from the server.
ppfopen	Open a distributed server-side file descriptor. The syntax is similar to that of the regular fopen() but the file is accessed on the server. You control data distribution when reading data from a file on the server as column distributed only.
ppquit	Disconnects from the server and causes the server to terminate.
ppwhos	Gives information about distributed variables and their sizes (similar to `whos`).
pph5whos	Print information about variables in a HDF5 file.
`Data Movement Functions`	
ppback	Transfers a local matrix to the server and stores the handle to the server matrix.
ppfront	Retrieves a distributed matrix from the server and stores it in local matrix.

Function	Description
ppchangedist	Allows you to explicitly change the distribution of a matrix in order to avoid implicit changes in subsequent operations.
pph5write	Writes variables to an HDF5 file on the server.
pph5read	Reads distributed variables from an HDF5 file on the server
ppload	Loads a data set from the server filesystem to the back-end.
ppsave	Saves backend data to the server filesystem.
Task Parallel Functions	
bcast, ppbcast	Broadcasts an array section where the entire argument is passed along to each invocation.of a function called by `ppeval`.
split, ppsplit	Splits an array for each iteration of a `ppeval` function.
ppeval	Executes a specified function in parallel on sections of input array(s)
	When using ppeval or ppevalsplit to call a compiled C++ library function, use the format `PACKAGENAME:FNAME`, where `PACKAGENAME` is the module name as returned by an earlier call to `ppevalcloadmodule`, and, `FNAME` is the function name registered in that module. For example, the call: `ppeval('C://solverlib:polyfit', arg1, arg2);` invokes the `polyfit` function in the `imsl` ppevalc module with input arguments `arg1` and `arg2`.
ppevalsplit	Returns a `dcell` object, a cell array of return values from each iteration

Function	Description
ppevalcloadmodule	Loads a C++ module for task parallel operation on the server. This function is deprecated as of Release 2.6.0. Loading compiled C++ libraries can now also be performed using `pploadpackage`.
ppevalcunloadmodule	Removes a previously loaded C++ module.from the server. This function is deprecated as of Release 2.6.0. Unloading compiled C++ libraries can now also be performed using `ppunloadpackage`.
Performance Functions	
ppperf	Star-P®'s performance monitoring function.
ppprofile	Collects and display performance information on Star-P®
pptic/pptoc	Provides information complementary to the MATLAB® `tic/toc` command

General Functions

fseek

```
ST = fseek(fid, offset, origin)
```

Repositions the file position indicator in the file with the given distributed file identifier FID to the byte specified with the `offset`.

The return value FID is a distributed file identifier. Passing this value to the following MATLAB functions:

- `fopen()`
- `fread()`
- `fwrite()`
- `fseek()`
- `frewind()` and
- `fclose()`

np

`n = np`

Function Syntax Description

- `n` (`double`) - number of processes
- `np` returns the number of processes in the server. This is the argument that was passed to the `-p` switch to `starp`.

Note: This number should be less than or equal to the number of processors.

p

`z = p`

Function Syntax Description

- `z` (`dlayout`) - a `dlayout` object
- `p` creates an instance of a `dlayout` object. `p` by itself is a 'symbolic variable'. Variables of type `dlayout` are used to tag dimensions as being distributed.

pp

`z = pp`

Function Syntax Description

- `z` (`dlayout` object) - a `dlayout` object
- `pp` is an alias to `p`. `pp` is useful for users who wish to use the variable `p` for another purpose.

Reference

- See `p`.

ppbench

`ppbench` collects information about the basic hardware and software characteristics of your server. When utilizing multiple CPUs in a cluster configuration, the output of this test should be examined for consistency; for example, the amount of memory per node should be the same and the reported CPU information is similar.

If `ppbench` is invoked with an output argument, then it will return a data structure that can be stored using the `save` function and later displayed (see example 1 below).

If `ppbench` is invoked with no input arguments, then it acts as if it were invoked with the `-levels [0,1]` switch.

If the `-levels` switch is used, the additional argument is either a scalar or a list of levels to be run. (see example 2 below)

`ppbench('-levels',0)` will print the lowest level system information which is extracted from `/proc/cpuinfo` and `/proc/meminfo`. In addition, when the Star-P® server utilizes more than 1 CPU, the generated report will include MPI latency and bandwidth data.

`ppbench('-levels',1)` will print the results of a single CPU HPC Streams benchmark. This provides an interesting data point that represents an important class of simple operations that turn up frequently in HPC applications. See http://www.cs.virginia.edu/stream/ to see how your results compare with a range of commodity and special purpose CPUs.

If the `'-display'` switch is used (example 3), the additional argument identifies the data structure saved from a previous invocation of `ppbench`, which is then displayed.

Example 1:

```
X = ppbench
```

Example 2:

```
ppbench('-levels',[0,1])
```

Example 3:

```
ppbench('-display',X)
```

---

ppclear

`ppclear` eliminates distributed variables from the caller's Star-P® workspace, and immediately frees the memory allocated for them on the server. If no argument is provided, then `ppclear` removes all distributed variables in the workspace.

`ppclear('var1','var2')` or **`ppclear var1 var2`** removes the listed variables only.

> **Important:** Invoking `bpp = app; ppclear app;` will leave the symbol `bpp` in your workspace, but the distributed object accessed through `bpp` will no longer exist. When you desire a hard copy of a variable, as opposed to a soft copy, use assignment statements such as `bpp = +app;` or `bpp = app(:,:);`.

ppgetoption

Returns the value of the Star-P® properties.

ppsetoption

Sets the value of the Star-P® properties.

ppsetoption('option','value')

Function Syntax Description

- **ppsetoption**('SparseDirectSolver', 'value') where value can be SuperLU or MUMPS
- **ppsetoption**('log', 'value') where value can be one of on (default) or off. This controls whether information about the steps executed by the server is written to the log.
- **ppsetoption**('ppfront_msg','value') where value can be one of on (default) or off. This controls whether or not the warning message from ppfront and ppback about large transfers and from ppchangedist about large redistributions is emitted.
- **ppsetoption**('ppfront_size',size) where size is the threshold above which the ppfront/ppback/ppchangedist warning message will be emitted. The default is 100 megabytes.
- **ppsetoption**('TaskParallelEngine',<'engine'>) where 'engine' is a sting containing the task parallel engine you wish to use when calling ppeval or ppevalsplit. Options for the task parallel engine setting include 'octave-2.9.5' (default), 'octave-2.9.9', or 'C'. Choosing 'C' as your task parallel engine allows you to call functions from compiled task parallel packages that are loaded using pploadpackage and called with ppeval or ppevalsplit.

ppgetlog

Get the Star-P® server log file.

Function Syntax Description

- f = **ppgetlog**

Returns the filename of a local temporary file containing the Star-P® server log. The temporary file is deleted when MATLAB exits.

- **ppgetlog**(`FILENAME`)

  Stores a copy of the Star-P® server log file in `FILENAME`.

- **ppgetlog** `-all`

  Copies ALL files from the server log directory into the client log directory and creates an `all_logs.zip` archive with all files in the client log directory.

- **ppgetlog** `-all <filename>`

  Creates a <filename> zip archive with all files from the client and server log directories.

- `f =` **ppgetlog** `-all`

  Creates an `all_logs.zip` archive with all files from the client and server log directories and returns the full filename of the archive.

- **ppgetlog** `-all -nozip`

  Copies all files from the server log directory into the client log directory.

The `'-nozip'` option is ignored if `'-all'` is not specified, <filename> is ignored if `'-all -nozip'` is specified, and the output is an empty string if `'-all -nozip'` is specified.

Note: `ppgetlog` will make an SSH connection to the Star-P® server machine to fetch the log file, so if your ssh client is not configured for passwordless SSH, then you may be prompted for your server password again.

---

ppgetlogpath

Get starpserver log file path.

Function Syntax Description

- `F =` **ppgetlogpath** returns the filename of the starpserver log on the server.
- `F =` **ppgetlogpath**`('server')` returns the filename of the starpserver log on the server.
- `F =` **ppgetlogpath**`('client')` returns the filename of the starpclient log on the client.

The naming format for individual session log directories is `YYYY_MM_DD_HHMM_SS`. Hours are represented in the 24-hour format.

The server log and configuration files will be named as follows:

```
<log>/workgroup_manager.log
<log>/starp_server.log
<log>/octave_$MPI_RANK.log
<log>/machine_file
<config>/machine_file.user_default
<log>/starp_session_id.*
<config>/user_env.sh
```

The Client log files will be named as follows:

```
<log>/starpmatlab.log
<log>/starpclient.log
```

---

ppinvoke

Invoke a function contained in a previously loaded user library via the Star-P® SDK.

Function Syntax Description

```
[varargout] = ppinvoke(function, varargin)
```

Note: See the “Star-P® Software Development Kit (SDK) Tutorial and Reference Guide” for more information on this function.

---

ppsloadpackage

Loads a compiled task parallel or data parallel user library on the server using positional arguments.

Function Syntax Description

```
stringTP = pploadpackage('C','/path/to/package.so','TPname')
stringTP = pploadpackage('C','/path/to/package.so')
```

Loads a package named `'package.so'` containing compiled functions for later use in `ppeval`. The first argument, specifies the language in which the target package is written. Currently, only C or C++ libraries can be loaded on the server for task parallel operation, and require the initial argument to be the string `'C'`. The second string argument specifies a user-defined name that is used for identification of the task parallel package on the server. The string provided with the keyword argument `name` is returned in the function output `stringTP`. If the third argument `'TPname'` is not provided, then the naming convention utilized for assigning an output string to `stringTP` is to take the filename without path, extension, or underscores, converted to lowercase. This change ensures that the default name can always be used to prefix a function name, and is recognizable by the Star-P® client and server.

```
stringDP = pploadpackage('/path/to/package.so','DPname')
```

```
stringDP = **pploadpackage**('/path/to/package.so')
```

When the initial engine string argument is omitted, the package specified will be loaded as a data parallel package. Currently, only C or C++ libraries can be loaded on the server for task parallel operation. The keyword argument “name” specifies a user-defined name that is used for identification of the data parallel package on the server. The string provided with the keyword argument `name` is returned in the function output `stringDP`. If the `name` keyword is not provided, then the naming convention utilized for assigning an output string to `stringDP` is to take the filename without path, extension, or underscores, converted to lowercase. This change ensures that the default name can always be used to prefix a function name, and is recognizable by the Star-P® client and server.

Note: See the “Star-P® Software Development Kit (SDK) Tutorial and Reference Guide” for more information on this function.

---

ppunloadpackage

Unload a user task parallel or data parallel library from the server.

Function Syntax Description

```
ppunloadpackage('C','TPname')
ppunloadpackage('C',stringTP)
```

By passing the initial engine string argument, `'C'`, along with a string containing the name of a compiled language task parallel package that has previously been loaded on the server, `ppunloadpackage` will unload the package from the Star-P® server’s current compiled language task parallel engine.

```
ppunloadpackage('DPname')
ppunloadpackage(stringDP)
```

By passing only a single string argument, containing the name of a compiled language data parallel package that has previously been loaded to the server, `ppunloadpackage` will unload the package from the Star-P® server.

In the case of unloading either a task parallel or data parallel package, if the name given for the package does not match the name of a package already loaded on the server, then an error will be thrown.

Note: See the “Star-P® Software Development Kit (SDK) Tutorial and Reference Guide” for more information on this function.

---

ppfopen

Open a distributed server-side file descriptor. The syntax is similar to that of the regular `fopen()` but the file is accessed on the server. You control data distribution when reading data from a file on the server as column distributed only.

Function Syntax Description

`FID = `**`ppfopen`**`('F')`

Opens file 'F' in read-only mode.

`FID = `**`ppfopen`**`('F', MODE)`

Opens file F in the mode specified by MODE. MODE can be: '

MODE	DESCRIPTION
rb	read
wb	write (create if necessary)
ab	append (create if necessary)
rb+	read and write (do not create)
wb+	truncate or create for read and write
ab+	read and append (create if necessary)

Return Values

The return value FID is a distributed file identifier. Passing this value to the following MATLAB functions: `fopen()`, `fread()`, `fwrite()`, `frewind()` and `fclose()` will operate on distributed matrices on the server with the same semantics as with regular file id on the client.

Note: For `fread()`, you control data distribution when reading data from a file on the server as column distributed only.

ppquit

Disconnects from the server and causes the server to terminate.

ppwhos

`ppwhos` lists the variables in the caller's Star-P® workspace. `ppwhos` is aware of distributed matrices that exist on the server so it will return the correct dimensions and sizes for those matrices, as well as returning the distribution information.

`ppwhos` is the Star-P® equivalent of the MATLAB `whos` command. It provides detailed information about the distribution of the server side variables (2nd column), their size (3rd column), and their types (4th column).

All distributed variables will also show up in the MATLAB `whos` command, but the information displayed for these variables does not accurately represent their size and distribution properties. The `ppwhos` output helps align the distributions of the variables; in general having similar distributions for all variables provides the best performance. It also allows identifying variables that should be distributed, since they are large, which variables are not, and variables that should not be distributed, since they are small, but are distributed. A typical `ppwhos` output looks something like this:

```
>> app = rand(1000,1000*p);
>> bpp = rand(1000*p,1000);
>> c = rand(1000,1000);
>> ppwhos
Your variables are:
  Name        Size             Bytes        Class
  app         1000x1000p       8000000      ddense array
  bpp         1000px1000       8000000      ddense array
  c           1000x1000        8000000      double array

Grand total is 3000000 elements using 24000000 bytes
MATLAB has a total of 1000000 elements using 8000000 bytes
Star-P® server has a total of 2000000 elements using 16000000 bytes
```

pph5whos

Print information about variables in a HDF5 file.

```
pph5whos('FILE')
```

Prints size and type information of variables in an HDF5 FILE on the server. The format is similar to the MATLAB `whos` function.

```
S = pph5whos('FILE')
```

Returns the dataset names in an HDF5 FILE along with the corresponding size and type information in a structure array, S.

> **Note:** `pph5whos` is able to parse an arbitrary HDF5 file, but will return accurate size and type information only for datasets that consist of double or double complex dense and sparse data. In all other cases, the type field is marked 'unknown'.

Reference

See also: `pph5write`, `pph5read`.

Data Movement Functions

ppback

```
Bpp = ppback(A)
Bpp = ppback(A, d)
```

Transfer the MATLAB matrix `A` to the backend server and stores the result in **`Bpp`**. `A` can be dense or sparse.

Function Syntax Description

- Input:
  - `A` (dense/sparse matrix) - MATLAB matrix to be transferred
  - `d` (optional) - distribution
- Output

  `Bpp` (`ddense/ddensend/dsparse` matrix) - distributed matrix

  Transfer the MATLAB matrix `A` to the backend server and store the result in `B`.

  If `A` is dense and two-dimensional:

  - If `d` is not specified, then **`Bpp`** is column distributed unless it is a column vector of length > 1, in which case it is row distributed.
  - If `d` is 1, then **`Bpp`** is row distributed.
  - If `d` is 2, then **`Bpp`** is column distributed.

  If `A` is dense and greater than two-dimensional:

  - If `d` is not specified, then **`Bpp`** is distributed along the last dimension, else.
  - **`Bpp`** is distributed along the dimension specified.

  If `A` is sparse:

  - **`Bpp`** is row distributed.

Important: A warning message is displayed if the transfer is over a threshold size (currently 100MB), to avoid silent performance losses. Emission of the message or the value of the threshold can be changed by use of the `ppsetoption` command.

Reference:

See also **`ppfront`**, **`ppsetoption`**.

ppfront

Transfers the distributed matrix **App** from the server to the MATLAB client.

```
B = ppfront(App)
```

Function Syntax Description

- Input: **App** - distributed matrix
- Output: `B` (dense/sparse MATLAB matrix) - local copy of **App**

`ppfront` transfers the distributed matrix `A` from the server to the MATLAB client.

- If **App** is a distributed dense matrix, then `B` is a dense MATLAB matrix.
- If **App** is a distributed sparse matrix, then `B` is a sparse matrix.

`dlayout` objects are converted to double and other non-distributed objects are preserved.

Important: A warning message is emitted if the transfer is over a threshold size (currently 100MB), to avoid silent performance losses. Displays the warning message or the value of the threshold can be changed by use of the `ppsetoption` command. Currently, there is also a 2GB limit for the size of data that can be transferred from the server to the client using `ppfront`.

Reference

See also **ppback, ppsetoption**.

ppchangedist

The `ppchangedist` command allows you to explicitly change the distribution of a matrix in order to avoid implicit changes in subsequent operations. This is especially important to do when performing operations within loops. In order to maximize performance, operands should have conformant distributions. `ppchangedist` can be used before and/or after the loop to prepare for subsequent operations.

Function Syntax Description

```
ppchangedist(App,dist)
```

- **App** is input `ddense`
- `dist` is the desired distribution
  - 1 for ROW
  - 2 for COLUMN

Important: A warning message is emitted if the transfer is over a threshold size (currently 100MB), to avoid silent performance losses. Emission of the message or the value of the threshold can be changed by use of the `ppsetoption` command.

---

pph5write

Write variables to a HDF5 file on the server.

Function Syntax Description

```
pph5write('FILE', VARIABLE1, 'DATASET1', VARIABLE2, 'DATASET2', ...)
```

Writes `VARIABLE1` to `DATASET1` in the `FILE` specified on the server in the HDF5 format.

- If the `FILE` already exists, it is overwritten.
- Similarly if one of the dataset variables already exists, it is also overwritten with the new variable.

```
pph5write('FILE', 'MODE', ...)
```

Specifies the output mode which can either be '`clobber`' or '`append`'.

- If the mode is '`clobber`' and `FILE` already exists, it is overwritten.
- If the mode is '`append`' and `FILE` already exists, the variables specified in the `PPH5WRITE` call are appended to the `FILE`. If `FILE` does not exist, it is newly created.

Example 1

```
% Write matrix_a to the dataset /my_matrices/a and matrix_b to the
% dataset /my_matrices/workspaces/temp/matrix_b to the file
% /tmp/temp.h5 on the server, overwriting it if it already exists
pph5write('/tmp/temp.h5', matrix_a, '/my_matrices/a', matrix_b, '/
          my_matrices/workspace/temp/matrix_b');
```

Example 2

```
% Append matrix_c to the existing file in the location /
% my_matrices/workspace2/temp/matrix_c
pph5write('/tmp/temp.h5', 'append', matrix_c,'/my_matrices/
          workspace2/temp/matrix_c');
```

Note: Currently, only writing double and double complex dense and sparse matrices is supported.

Reference

See also: **pph5read**, **pph5whos**.

---

pph5read

```
[VARIABLE1, VARIABLE2, ...] = pph5read('FILE', 'DATASET1', 'DATASET2', ...)
```

Read distributed variables from a HDF5 file on the server.

Reads from `FILE`, the contents of `DATASET1` into `VARIABLE1`, `DATASET2` into `VARIABLE2`, etc.

- If any of the datasets is missing or invalid, or the `FILE` is not a valid HDF5 file, the function returns an error.

Example

```
% Read the contents of the dataset /my_matrices/workspace/temp/matrix_b from
the file /tmp/temp.h5 into the distributed variable matrix_d
matrix_d = pph5read('/tmp/temp.h5', '/my_matrices/workspace/temp/matrix_b');
```

Only the contents of datasets which contain double or double complex dense or sparse data can currently be read. In the latter case, the sparse matrix must be stored in a specific format outlined in "How Star-P® Represents Sparse Matrices".

Reference

See also: **pph5write**, **pph5whos**.

---

ppload

```
ppload('f', 'v1', 'v2', ..., dist)
```

Loads the distributed objects named `v1`, `v2`, ... from the file `f` into variables of the same names. Specify the distribution to use with `dist`.

Function Syntax Description

- **ppload**`('f', dist)`

  Loads all variables out of mat file `f`, retaining their original names. All loaded matrices will be distributed the same way, given by `dist`. A `dist` value of 1 denotes a row-distributed object, and a value of 2 denotes a column-distributed object.

- **ppload**`('f','v1', 'v2', ...)`

- **ppload**`('f')`

If `dist` is omitted, the `ddense` objects will be column-distributed by default.

- `S = **ppload**('f', 'v1', 'v2', ..., dist)`

  Defines `S` to be a `struct` containing fields that match the returned variables.

- `**ppload** f v1, v2, ...`

  Alternate syntax description

---

ppsave

```
ppsave('f', 'v1', 'v2', ...)
```

Saves the distributed objects `v1`, `v2`, ... directly to the server file `f`, each under its own name.

Function Syntax Description

- `**ppsave**('f')`

  If no variables are listed, saves all distributed objects currently assigned to variable names in the workspace.

- `**ppsave**('f', 'v1', 'v2', ..., -append)`

  Appends the variables to the end of file `f` instead of overwriting.

- `**ppsave**('f', 'v1', 'v2', ...,)`

  Splits the variable data into one file per processor, each containing the local data for that processor.

- `**ppsave** f v1, v2, ...`

  Alternate syntax description

Important:`ppsave` will not save the contents of any local (client) objects.

Task Parallel Functions

---

bcast, ppbcast

Tag distributed object `x` as being broadcast to all of the **`ppeval`** calls.

```
y = bcast(x)
y = ppbcast(x)
```

References

Also, see **ppeval**, and **split**/**ppsplit**.

---

split, ppsplit

Split a distributed object **Xpp** along dimension `dim`. Used as input to **ppeval**.

```
y = split(Xpp,dim)
y = ppsplit(Xpp,dim)
```

Function Syntax Description

If `dim` == 0, then **Xpp** is split into its elements

Example

split`(`**Xpp**`,1)` splits **Xpp** by rows
ppsplit`(`**Xpp**`,2)` splits **Xpp** by columns

Each row is then an input to the function specified in the **ppeval** call.

References

Also, see **ppeval** and **bcast**/**ppbcast**.

---

ppeval

Execute a function in parallel on distributed data. `ppeval` is just another way of specifying iteration.

```
[o1,o2,...,oN] = ppeval('foo',in1,in2,...,inl)
```

Function Syntax Description

Two pieces of information are required for the call:

- The function to be executed. This is the `foo` argument. It is a string containing the function name.
- The specification of the set of inputs to `foo`. These are the `in1` arguments. If `foo` is a function of `k` arguments then `k in1` arguments are needed. Each of these arguments are split into inputs to `foo` by the following rules:
  - If `class(`**in1**`) = 'ddense'`, then it is split by columns.
    If `class(in1) = 'double'`, then it is broadcast (each invocation of foo gets the entire argument)
    - `in1 =` **split**`(ddense,d)`, then it is split along dimension `d`

- `in1 = ppsplit(ddense,d)`, then it is split along dimension `d`
- `in1 = split(ddense,0)`, then it is split into its constituent elements
- `in1 = ppsplit(ddense,0)`, then it is split into its constituent elements
- `in1 = bcast(a)`, then `a` is broadcast
- `in1 = ppbcast(a)`, then a is broadcast.

Note: The arguments must “conform” in the sense that the size of each split (excluding broadcasts, of course) must be the same for all the arguments that are split. In this way we can determine the total number of calls to `foo` that will be made.

The output arguments, `o1, o2, ..., oN` are `ddense` or `ddensend` arrays representing the results of calling '`foo`'. Each output argument is created by concatenating the result of each iteration along the next highest dimension; for example, if `K` iterations of `foo` are performed and the output of each iteration is a matrix of size `MxN`, then the corresponding output after the `ppeval` invocation will be a `MxNxK` matrix.

Note: Note that prior versions of Star-P® had a version of `ppeval` that did not reshape the output arguments to `ddense` objects. For backward compatibility, this function is now as `ppevalsplit`.

`ppeval` is only defined for arguments that are dense.

If `foo` returns `n` output arguments then there will be `n` output arguments. See also `split` and `bcast`.

When using `ppeval` or to call a compiled C++ library function, use the format `MODULENAME:FNAME`, where `MODULENAME` is the module name as returned by an earlier call to `ppevalcloadmodule`, and, `FNAME` is the function name registered in that module. For example, the call:

```
ppeval('C://solverlib:polyfit', arg1, arg2);
```

invokes the `polyfit` function in the `imsl` C++ module with input arguments `arg1` and `arg2`.

Known Differences Between MATLAB and Octave Functions

This section lists the known differences between MATLAB and Octave, which is useful to know when Octave is set as your task parallel engine (the default setting).

- If an `inf` value is present in a matrix that is used as an argument to `eig` in ppeval, Star-P® may hang, while MATLAB returns an error.
- When using the Star-P® Octave TPE, the evaluation of the '++' and '--' auto-increment/decrement operators differs between ppeval and MATLAB. For example, x=7;++x returns 8 in `ppeval`, but returns 7 in MATLAB.

ppevalsplit

ppevalsplit()

The dcell is analogous to MATLAB cells. The dcell type is different from the other distributed matrix or array types, as it may not have the same number of data elements per dcell iteration and hence doesn't have the same degree of regularity as the other distributions. This enables dcells to be used as return arguments for ppevalsplit().

Because of this potential irregularity, a dcell object cannot be used for much of anything until it is converted into a “normal” distributed object via the reshape operator. The only operators that will work on a dcell are those that help you figure out what to convert it into, e.g., size, numel, length, and reshape, which converts it, in addition to ppwhos. Luckily, you will almost never need to be aware of dcell arrays or manipulate them.

When using ppevalsplit to call a compiled C++ library function, use the format PACKAGENAME:FNAME, where PACKAGENAME is the module name as returned by an earlier call to ppevalcloadmodule (deprecated) or pploadpackage, and, FNAME is the function name registered in that package. For example, the call:

```
ppevalsplit('C://solverlib:polyfit', arg1, arg2);
```

invokes the polyfit function in the imsl C++ module with input arguments arg1 and arg2.

ppevalcloadmodule

NAME = **ppevalcloadmodule**(FNAME, NAME)

Loads a task parallel C++ module on the server.

This function is deprecated as of release 2.6.0. Compiled C and C++ task parallel libraries can now be loaded on the server with pploadpackage.

ppevalcunloadmodule

ppevalcunloadmodule(NAME)

Remove a previously loaded task parallel C++ module.

This function is deprecated as of release 2.6.0. Compiled C and C++ task parallel libraries can now be unloaded from the server with ppunloadpackage.

Performance Functions

ppperf

ppperf

Provides fine-grained profiling of compute activity on both the client and the server together. It pays close attention to the time required to perform computational tasks. It also tracks communication between the client and server over the network. The vision behind `ppperf` is to provide you a top-level view of what your program is doing as it runs your calculation. Using the information provided by `ppperf`, you can

- identify program choke points,
- identify excessive client/server communication,
- see what functions are invoked on both client and server, and
- see how long each function takes to finish.

This information can be invaluable when debugging or optimizing a Star-P® application.

Function Syntax Description

Command	Explanation
ppperf [01 | 02 | 03 | on] <number>	This command starts the performance monitoring process and initializes the results table. It is always the first command you issue when you want to profile your code's execution. `<number>` is the update interval (in seconds) for statistics gathering. The interval must be an integer. If you don't specify this parameter, it is set to 1 second.
ppperf off	This command turns off the performance monitoring process, but leaves the results table alone. Use this command if you want to perform some work without gathering statistics. You may later resume statistics gathering by entering `ppperf resume`.
ppperf clear	This command clears the results table, and turns off performance data logging. Use this command if you want to end your performance monitoring session.
ppperf resume	This command restarts the performance monitoring process (in other words, after you have paused it using `ppperf off`). It will not affect the results table.

ppperf report	This command prints out a large text report providing information about compute resources utilized by your program while it ran.
ppperf report detail	This command prints out a large text report providing information about compute resources utilized by your program while it ran. It provides more detail than `ppperf report`. In particular, it breaks the process measurement results down for each compute node on your parallel server.
ppperf graph on	This command displays a graph showing compute resource utilization on the client, network, and server. If you invoke this command before running your program, it will show you a real-time graph of your computation's activity (as long as control passes to the client). If you invoke this command after executing `ppperf off`, it will show you the static graph of compute activity recorded between `ppperf on` and `ppperf off`.
ppperf graph off	This command closes the performance graph window. It does not affect the compiled performance data table.

ppprofile

The **ppprofile** command collects and displays performance information for Star-P®. `ppprofile` is a profiler for the Star-P® server. It allows you to examine which function calls the Star-P® server makes and how much time is spent in each call.

Function Syntax Description

- **ppprofile on** starts the collection of performance data about each call from the Star-P® client to the Star-P® server.
- **ppprofile on -detail basic** has the same effect as **ppprofile on**.
- **ppprofile on -detail full** also collects information about the number of changes of distribution that occur on the server, and the amount of time spent executing in the server.
- **ppprofile off** stops gathering data without clearing the data that's already been collected.
- **ppprofile clear** clears the collected data.
- **ppprofile display** displays the data about each server call as it occurs.
- **ppprofile nodisplay** delays the immediate display of data about each server call.
- **ppprofile report** generates a report of the data collected so far.

See "Summary and Per-Server-Call Timings with ppprofile" for examples of the usage of `ppprofile`.

Example

To turn profiling on, issue the following command:

```
>> ppprofile on
```

Then follow with the commands or scripts of interest and end with `ppprofile` report:

```
>> ppprofile on
>> app = rand(1000*p);
>> bpp = inv(app);
>> dpp = inv(app);
>> cpp = eig(bpp);
>> ppprofile report
function                 calls      time   avg time    %calls     %time
ppscalapack_eig              1    6.3244     6.3244        10    90.082
ppscalapack_inv              2   0.62992    0.31496        20    8.9723
ppdense_scalar_op            1  0.014254   0.014254        10   0.20303
ppdense_binary_op            1  0.012628   0.012628        10   0.17987
ppdense_sumv                 1  0.009353   0.009353        10   0.13322
ppdense_rand                 1  0.009036   0.009036        10    0.1287
ppbase_setoption             1   0.00856    0.00856        10   0.12192
ppdense_transpose            1  0.006571   0.006571        10  0.093594
ppdense_sum                  1  0.005996   0.005996        10  0.085404
Total                       10    7.0208    0.70208
```

The `ppprofile` information is ordered in columns and displays, from left to right, the server function called, the number of function calls, the time spent inside the function, the average time spent inside the function per function call, the percentage of function calls, and the percentage of time spend inside the function. For the full range of functionality of `ppprofile` please consult the Command Reference Guide or type `help ppprofile` in Star-P®.

pptic/pptoc

`pptic/pptoc` provides information complementary to the MATLAB `tic/toc` command. The latter provides the wall-clock time of the instructions enclosed by the `tic/toc` statement and the former provides information on the communication between the client and the server.

The `pptic/pptoc` output displays:

- the number of messages and the number of bytes received by the server from the client and
- the number of messages and the number of bytes sent from the server to the client.

```
>> app = rand(1000*p);
>> pptic; dpp = inv(app); pptoc;
Client/server communication report:
  Sent by server: 1 messages, 1.120e+02 bytes
  Received by server: 1 messages, 2.400e+01 bytes
  Total communication time: 4.840e-05 seconds
Server processing report:
  Duration of calculation on server (wall clock time): 3.124e-01s
  #ppchangedist calls: 0
----------------------------------------------------------------------
Total time: 3.219e-01 seconds
```

In addition to the number of messages and bytes received and sent, `pptic/pptoc` shows the time spent on communication and calculation as well as the number of distribution changes needed to accomplish the instructions enclosed by the `pptic/pptoc` statement.

The two important pieces of information contained in `pptic/pptoc` that affect performance are the bytes received or sent and the number of distribution changes.

Combining client variables and server variables in the expression will result in the movement of the client variable to the server, which will show up in the bytes received field. Since data movement is expensive, this is a possible place to enhance performance, especially if the expression happens to be located inside a looping construct. For example, compare the following two calculations:

Example 1

```
% Multiply client matrix and server matrix
>> A = rand(1000);
>> Bpp = rand(1000*p);
>> tic; pptic; Cpp = A * Bpp; pptoc; toc;
Client/server communication report:
  Sent by server: 2 messages, 1.840e+02 bytes
  Received by server: 2 messages, 8.000e+06 bytes
  Total communication time: 6.799e-01 seconds
Server processing report:
  Duration of calculation on server (wall clock time): 1.427e-01s
  #ppchangedist calls: 0
----------------------------------------------------------------------
Total time: 8.771e-01 seconds
Elapsed time is 0.877322 seconds.
```

Example 2

```
% Multiply two server matrices
>> App = rand(1000*p);
>> Bpp = rand(1000*p);
>> tic; pptic; Cpp = App * Bpp; pptoc; toc;
Client/server communication report:
```

```
  Sent by server: 1 messages, 9.600e+01 bytes
  Received by server: 1 messages, 4.000e+01 bytes
  Total communication time: 6.127e-05 seconds
Server processing report:
  Duration of calculation on server (wall clock time): 1.149e-01s
  #ppchangedist calls: 0
----------------------------------------------------------------------
Total time: 1.251e-01 seconds
Elapsed time is 0.125299 seconds.
```

In the first example, you see that number of bytes received by the server is exactly the size of **`App`**, 1000*1000*8 bytes = 8 MB, and that the communication took 1.16 sec.

In the second example, the number of bytes received is 212 or 37,000 times smaller. These 212 bytes contain the instructions to the server that specify what operations need to be performed. The penalty you pay in the first example is 1.16 sec of data transfer, which could have been prevented by creating the variable **`App`** on the server instead of on the client.

The number of distribution changes reported by `pptic/toc` indicates how often Star-P® needed to make a temporary change to the distribution of a variable, for example, from row to column distributed, in order to perform a set of instructions. Distribution changes cost time and should be avoided whenever possible when optimizing code for performance (note that distribution changes become more expensive for slower interconnects between the processors, e.g., clusters). In general, keeping the distributions of all variables aligned, i.e., all row distributed or all column distributed, prevents distribution changes and improves performance.

INTERACTIVE supercomputing

Chapter 7

Supported MATLAB® Functions

This chapter lists the MATLAB[1] functions supported by Star-P®. The table in the section titled "Data Parallel Functions Listed Alphabetically" lists the supported data parallel functions in alphabetical order, while the tables in the section titled "Task-Parallel Functions Listed by Default Platform TPE" list task-parallel functions for what is referred to as "ppeval()" mode.

Refer to the support web page, http://www.interactivesupercomputing.com/support, for the most up-to-date function status.

Sparse matrices and functions operating on sparse matrices cannot currently be passed into a `ppeval` call, but may be used within the function called by `ppeval`.

Data Parallel Functions Listed Alphabetically

Table 1 lists the MATLAB® functions available for Data-Parallel Computing with Star-P® Release 2.7 x86/64 or Itanium-based Servers.

Table 1 Data-Parallel Functions

MATLAB Function	Function Class	Dense Array	Sparse Array
abs	elfun	Yes	Yes
acosd	elfun	Yes	Yes
acos	elfun	Yes	Yes
acosh	elfun	Yes	Yes
acotd	elfun		Yes

1. MATLAB® is a registered trademark of The MathWorks, Inc. Star-P® and the "star p" logo are registered trademarks of Interactive Supercomputing, Inc. Other product or brand names are trademarks or registered trademarks of their respective holders. ISC's products are not sponsored or endorsed by The MathWorks, Inc. or by any other trademark owner referred to in this document.

Table 1 Data-Parallel Functions

MATLAB Function	Function Class	Dense Array	Sparse Array
acot	elfun		
acoth	elfun		
acscd	elfun	Yes	Yes
acsc	elfun		
acsch	elfun		
all	ops	Yes	Yes
and	ops	Yes	Yes
angle	elfun	Yes	Yes
any	ops	Yes	Yes
asecd	elfun	Yes	Yes
asec	elfun		
asech	elfun		
asind	elfun	Yes	Yes
asin	elfun	Yes	Yes
asinh	elfun	Yes	Yes
atan2	elfun	Yes	Yes
atand	elfun		Yes
atan	elfun	Yes	Yes
atanh	elfun	Yes	Yes
blkdiag	elmat	Yes	Yes
cat	elmat	Yes	Yes
ceil	elfun	Yes	Yes
cell	datatypes	Yes	
chol	matfun	Yes	
clpxpair	elfun		
colon	ops		
colperm	sparfun		Yes

Table 1 Data-Parallel Functions

MATLAB Function	Function Class	Dense Array	Sparse Array
compan	elmat	Yes	Yes
complex	elfun	Yes	
cond	matfun	Yes	
conj	elfun	Yes	Yes
cosd	elfun	Yes	Yes
cos	elfun	Yes	Yes
cosh	elfun	Yes	Yes
cotd	elfun		Yes
cot	elfun		
coth	elfun		
cov	datafun	Yes	Yes
cscd	elfun		Yes
csc	elfun		
csch	elfun		
ctranspose (')	ops	Yes	Yes
cumprod	datafun	Yes	Yes
cumsum	datafun	Yes	Yes
deal	datatypes	Yes	Yes
diag	elmat	Yes	Yes
diff	datafun	Yes	
disp	lang	Yes	Yes
display	lang	Yes	Yes
dot	specfun	Yes	Yes
double	datatypes		
eig	matfun	Yes	
eigs	sparfun		Yes
ellipke	specfun		Yes

Table 1 Data-Parallel Functions

MATLAB Function	Function Class	Dense Array	Sparse Array
end	lang	Yes	Yes
eq	ops	Yes	Yes
exp	elfun	Yes	Yes
expm1	elfun	Yes	Yes
eye	elmat		
factorial	specfun		
factor	specfun		
false	elmat		
fft2	datafun	Yes	
fft	datafun	Yes	
fftshift	datafun		Yes
find	elmat	Yes	Yes
fix	elfun	Yes	Yes
flipdim	elmat		Yes
fliplr	elmat	Yes	Yes
flipud	elmat	Yes	Yes
fprintf	iofun	Yes	Yes
freqspace	elmat		
full	sparfun	Yes	Yes
ge	ops	Yes	Yes
gt	ops	Yes	Yes
hadamard	elmat		
hankel	elmat	Yes	Yes
hess	matfun	Yes	
hex2dec	strfun		
histc	datafun	Yes	
horzcat	ops	Yes	Yes

Table 1 Data-Parallel Functions

MATLAB Function	Function Class	Dense Array	Sparse Array
ifft2	datafun	Yes	
ifft	datafun	Yes	
ifftshift	datafun		Yes
imag	elfun	Yes	Yes
ind2sub	elmat		Yes
inf	elmat		
invhilb	elmat		
inv	matfun	Yes	
ipermute	elmat	Yes	
isa	datatypes	Yes	Yes
isempty	elmat	Yes	Yes
isequal	elmat	Yes	Yes
isequalwithequalnans	elmat	Yes	Yes
isfinite	elmat	Yes	Yes
isfloat	datatypes	Yes	Yes
isinf	elmat	Yes	Yes
islogical	datatypes	Yes	Yes
isnan	elmat	Yes	Yes
isnumeric	datatypes	Yes	Yes
isprime	specfun	Yes	Yes
isreal	elfun	Yes	Yes
isspace	strfun	Yes	Yes
issparse	sparfun	Yes	Yes
kron	ops	Yes	
ldivide (.\)	ops	Yes	Yes
length	elmat	Yes	Yes
le	ops	Yes	Yes

Table 1 Data-Parallel Functions

MATLAB Function	Function Class	Dense Array	Sparse Array
linspace	elmat		
log10	elfun	Yes	Yes
log1p	elfun		Yes
log2	elfun	Yes	Yes
log	elfun	Yes	Yes
logical	datatypes	Yes	Yes
logspace	elmat		
lt	ops	Yes	Yes
lu	matfun	Yes	
magic	elmat		
max	datafun	Yes	Yes
mean	datafun	Yes	Yes
median	datafun	Yes	Yes
meshgrid	elmat	Yes	Yes
min	datafun	Yes	Yes
minus (-)	ops	Yes	Yes
mldivide (\)	ops	Yes	Yes
mod	elfun	Yes	Yes
mpower (^)	ops	Yes	Yes
mrdivide (/)	ops	Yes	Yes
mtimes (*)	ops	Yes	Yes
nan	elmat		
nchoosek	specfun		
ndgrid	elmat		Yes
ndims	elmat	Yes	Yes
ne	ops	Yes	Yes
nnz	sparfun	Yes	Yes

Table 1 Data-Parallel Functions

MATLAB Function	Function Class	Dense Array	Sparse Array
normest	matfun		Yes
norm	matfun	Yes	Yes
not	ops	Yes	Yes
num2str	strfun		
numel	elmat	Yes	Yes
ones	elmat		
or	ops	Yes	Yes
orth	matfun	Yes	
permute	elmat	Yes	
pinv	matfun	Yes	
planerot	matfun	Yes	Yes
plus (+)	ops	Yes	Yes
pol2cart	specfun		Yes
power (.^)	ops	Yes	Yes
prod	datafun	Yes	Yes
qr	matfun	Yes	
rand	elmat		
randn	elmat		
rank	matfun	Yes	
rdivide (./)	ops	Yes	Yes
real	elfun	Yes	Yes
rem	elfun	Yes	Yes
repmat	elmat		
reshape	elmat	Yes	Yes
rot90	elmat	Yes	Yes
round	elfun	Yes	Yes
schur	matfun	Yes	

Table 1 Data-Parallel Functions

MATLAB Function	Function Class	Dense Array	Sparse Array
secd	elfun		Yes
sec	elfun		
sech	elfun		
sign	elfun	Yes	Yes
sind	elfun	Yes	Yes
sin	elfun	Yes	Yes
sinh	elfun	Yes	Yes
size	elmat	Yes	Yes
sort	datafun	Yes	Yes
sortrows	datafun	Yes	Yes
sparse	sparfun	Yes	Yes
spaugment	sparfun	Yes	Yes
spdiags	sparfun	Yes	Yes
speye	sparfun		
spfun	sparfun		Yes
sph2cart	specfun		Yes
spones	sparfun		Yes
sprandn	sparfun		
sprand	sparfun		
sprintf	strfun	Yes	Yes
sqrt	elfun	Yes	Yes
sqrtm	matfun	Yes	
squeeze	elmat	Yes	
std	datafun	Yes	Yes
sub2ind	elmat	Yes	
subsasgn	ops	Yes	Yes
subsindex	ops	Yes	Yes

Table 1 Data-Parallel Functions

MATLAB Function	Function Class	Dense Array	Sparse Array
subsref	ops	Yes	Yes
sum	datafun	Yes	Yes
svd	matfun	Yes	
svds	sparfun	Yes	Yes
tand	elfun	Yes	Yes
tan	elfun	Yes	Yes
tanh	elfun	Yes	Yes
times (.*)	ops	Yes	Yes
toeplitz	elmat	Yes	Yes
trace	matfun	Yes	Yes
transpose (.')	ops	Yes	Yes
tril	elmat	Yes	Yes
triu	elmat	Yes	Yes
true	elmat		
uminus (-)	ops	Yes	Yes
union	ops	Yes	Yes
unique	ops	Yes	Yes
unwrap	elfun	Yes	
uplus (+)	ops	Yes	Yes
vander	elmat	Yes	Yes
var	datafun	Yes	
vertcat	ops	Yes	Yes
xor	ops	Yes	Yes
zeros	elmat		

Task-Parallel Functions Listed by Default Platform TPE

Table 2 lists the MATLAB® functions available for Default Task Parallel Engine (TPE) for SGI-Altix/Itanium-based Servers Star-P® Release 2.7 and optional TPE for x86/64-based Servers.

Table 2 Default TPE Functions for Altix/Itanium

MATLAB Function	Function Class
conv	datafun
corrcoef	datafun
cov	datafun
cumprod	datafun
cumsum	datafun
cumtrapz	datafun
deconv	datafun
del2	datafun
detrend	datafun
diff	datafun
fft	datafun
fft2	datafun
fftn	datafun
fftshift	datafun
filter	datafun
filter2	datafun
gradient	datafun
hist	datafun
ifft	datafun
ifftn	datafun
max	datafun
mean	datafun
median	datafun

Table 2 Default TPE Functions for Altix/Itanium

MATLAB Function	Function Class
min	datafun
prod	datafun
sort	datafun
sortrows	datafun
std	datafun
sum	datafun
trapz	datafun
var	datafun
cast	datatypes
cell	datatypes
cell2mat	datatypes
cell2struct	datatypes
cellfun	datatypes
class	datatypes
deal	datatypes
double	datatypes
fieldnames	datatypes
func2str	datatypes
functions	datatypes
getfield	datatypes
isa	datatypes
iscell	datatypes
isfield	datatypes
isnumeric	datatypes
isstruct	datatypes
logical	datatypes
mat2cell	datatypes

Table 2 Default TPE Functions for Altix/Itanium

MATLAB Function	Function Class
num2cell	datatypes
orderfields	datatypes
rmfield	datatypes
setfield	datatypes
single	datatypes
str2func	datatypes
struct	datatypes
struct2cell	datatypes
abs	elfun
acos	elfun
acosh	elfun
acot	elfun
acoth	elfun
acsc	elfun
acsch	elfun
angle	elfun
asec	elfun
asech	elfun
asin	elfun
asinh	elfun
atan	elfun
atan2	elfun
atanh	elfun
ceil	elfun
clpxpair	elfun
complex	elfun
conj	elfun

Table 2 Default TPE Functions for Altix/Itanium

MATLAB Function	Function Class
cos	elfun
cosh	elfun
cot	elfun
coth	elfun
csc	elfun
csch	elfun
exp	elfun
fix	elfun
imag	elfun
isreal	elfun
log	elfun
log10	elfun
log2	elfun
mod	elfun
nextpow2	elfun
nthroot	elfun
pow2	elfun
real	elfun
rem	elfun
round	elfun
sec	elfun
sech	elfun
sign	elfun
sin	elfun
sind	elfun
sinh	elfun
sqrt	elfun
tan	elfun

Table 2 Default TPE Functions for Altix/Itanium

MATLAB Function	Function Class
tanh	elfun
unwrap	elfun
blkdiag	elmat
cat	elmat
circshift	elmat
compan	elmat
diag	elmat
eps	elmat
eye	elmat
find	elmat
flipdim	elmat
fliplr	elmat
flipud	elmat
flops	elmat
hankel	elmat
hilb	elmat
i	elmat
ind2sub	elmat
intmax	elmat
intmin	elmat
invhilb	elmat
ipermute	elmat
isempty	elmat
isequal	elmat
isequalwithequalnans	elmat
isinf	elmat
isnan	elmat
isscalar	elmat

Table 2 Default TPE Functions for Altix/Itanium

MATLAB Function	Function Class
isvector	elmat
j	elmat
length	elmat
linspace	elmat
logspace	elmat
meshgrid	elmat
ndims	elmat
numel	elmat
ones	elmat
pascal	elmat
permute	elmat
pi	elmat
rand	elmat
randn	elmat
realmax	elmat
realmax	elmat
repmat	elmat
reshape	elmat
rosser	elmat
rot90	elmat
rref	elmat
shiftdim	elmat
size	elmat
squeeze	elmat
sub2ind	elmat
toeplitz	elmat
tril	elmat
triu	elmat

Table 2 Default TPE Functions for Altix/Itanium

MATLAB Function	Function Class
vander	elmat
wilkinson	elmat
zeros	elmat
fminbnd	funfun
fminsearch	funfun
fzero	funfun
inline	funfun
ode23	funfun
ode45	funfun
quad	funfun
quadl	funfun
vectorize	funfun
addpath	general
ans	general
beep	general
brighten	general
cd	general
clear	general
computer	general
delete	general
diary	general
dir	general
dos	general
echo	general
exit	general
fileattrib	general

Table 2 Default TPE Functions for Altix/Itanium

MATLAB Function	Function Class
format	general
genpath	general
getenv	general
isdir	general
ispc	general
isunix	general
load	general
ls	general
mex	general
mkdir	general
more	general
pack	general
path	general
pwd	general
quit	general
rehash	general
rmdir	general
rmpath	general
save	general
savepath	general
system	general
type	general
unix	general
ver	general
which	general
who	general
whos	general

Table 2 Default TPE Functions for Altix/Itanium

MATLAB Function	Function Class
clc	iofun
csvread	iofun
csvwrite	iofun
fclose	iofun
feof	iofun
ferror	iofun
fgetl	iofun
fgets	iofun
fileparts	iofun
filesep	iofun
fopen	iofun
fprintf	iofun
fread	iofun
frewind	iofun
fscanf	iofun
fseek	iofun
ftell	iofun
fullfile	iofun
fwrite	iofun
home	iofun
rename	iofun
tar	iofun
tempdir	iofun
tempname	iofun
textread	iofun
untar	iofun
unzip	iofun

Table 2 Default TPE Functions for Altix/Itanium

MATLAB Function	Function Class
assignin	lang
break	lang
builtin	lang
case	lang
catch	lang
continue	lang
disp	lang
else	lang
elseif	lang
end	lang
error	lang
eval	lang
evalin	lang
exist	lang
feval	lang
for	lang
global	lang
if	lang
input	lang
inputname	lang
isglobal	lang
iskeyword	lang
isvarname	lang
keyboard	lang
lasterr	lang
lastwarn	lang
mislocked	lang
mlock	lang

Table 2 Default TPE Functions for Altix/Itanium

MATLAB Function	Function Class
munlock	lang
nargchk	lang
nargin	lang
nargout	lang
otherwise	lang
persistent	lang
return	lang
switch	lang
try	lang
varargin	lang
varargout	lang
warning	lang
while	lang
MATLAB Function	Function Class
balance	matfun
chol	matfun
cond	matfun
det	matfun
eig	matfun
expm	matfun
hess	matfun
inv	matfun
logm	matfun
lu	matfun
norm	matfun
null	matfun

Table 2 Default TPE Functions for Altix/Itanium

MATLAB Function	Function Class
orth	matfun
pinv	matfun
qr	matfun
qz	matfun
rank	matfun
schur	matfun
sqrtm	matfun
svd	matfun
trace	matfun
MATLAB Function	Function Class
all	ops
and	ops
any	ops
bitand	ops
bitcmp	ops
bitget	ops
bitmax	ops
bitor	ops
bitset	ops
bitshift	ops
bitxor	ops
eq	ops
ge	ops
gt	ops
horzcat	ops
intersect	ops

Table 2 Default TPE Functions for Altix/Itanium

MATLAB Function	Function Class
ismember	ops
kron	ops
ldivide (.\)	ops
le	ops
lt	ops
minus (-)	ops
mldivide (\)	ops
mpower (^)	ops
mrdivide (/)	ops
mtimes (*)	ops
ne	ops
not	ops
or	ops
plus (+)	ops
power (.^)	ops
rdivide (./)	ops
setdiff	ops
setxor	ops
times (.*)	ops
uminus (-)	ops
union	ops
unique	ops
uplus (+)	ops
vertcat	ops
xor	ops
interp1	polyfun
interp2	polyfun

Table 2 Default TPE Functions for Altix/Itanium

MATLAB Function	Function Class
interpft	polyfun
mkpp	polyfun
pchip	polyfun
poly	polyfun
polyarea	polyfun
polyder	polyfun
polyfit	polyfun
polyval	polyfun
polyvalm	polyfun
ppval	polyfun
residue	polyfun
roots	polyfun
spline	polyfun
ss2tf	polyfun
unmkpp	polyfun
colamd	sparfun
colperm	sparfun
dmperm	sparfun
etree	sparfun
etreeplot	sparfun
full	sparfun
gplot	sparfun
issparse	sparfun
luinc	sparfun
nnz	sparfun
nonzeros	sparfun
nzmax	sparfun

Table 2 Default TPE Functions for Altix/Itanium

MATLAB Function	Function Class
randperm	sparfun
spalloc	sparfun
sparse	sparfun
spconvert	sparfun
speye	sparfun
spfun	sparfun
spones	sparfun
spparms	sparfun
sprand	sparfun
sprandn	sparfun
sprandsym	sparfun
spy	sparfun
symamd	sparfun
airy	specfun
besselh	specfun
besseli	specfun
besselj	specfun
besselk	specfun
bessely	specfun
beta	specfun
betainc	specfun
betain	specfun
cart2pol	specfun
cart2sph	specfun
cross	specfun
dot	specfun
erf	specfun

Table 2 Default TPE Functions for Altix/Itanium

MATLAB Function	Function Class
erfc	specfun
erfinv	specfun
gamma	specfun
gammainc	specfun
gammaln	specfun
gcd	specfun
hsv2rgb	specfun
lcm	specfun
legendre	specfun
perms	specfun
pol2cart	specfun
primes	specfun
rgb2hsv	specfun
sph2cart	specfun
base2dec	strfun
bin2dec	strfun
blanks	strfun
cellstr	strfun
char	strfun
deblank	strfun
dec2base	strfun
dec2bin	strfun
dec2hex	strfun
findstr	strfun
hex2dec	strfun
hex2num	strfun
int2str	strfun

Table 2 Default TPE Functions for Altix/Itanium

MATLAB Function	Function Class
iscellstr	strfun
ischar	strfun
isletter	strfun
isspace	strfun
isstr	strfun
lower	strfun
mat2str	strfun
num2str	strfun
regexp	strfun
regexpi	strfun
regexprep	strfun
setstr	strfun
sprintf	strfun
sscanf	strfun
str2double	strfun
str2mat	strfun
str2num	strfun
strcat	strfun
strcmp	strfun
strcmpi	strfun
strfind	strfun
strjust	strfun
strmatch	strfun
strncmp	strfun
strncmpi	strfun
strrep	strfun
strtok	strfun
strtrim	strfun

Table 2 Default TPE Functions for Altix/Itanium

MATLAB Function	Function Class
strvcat	strfun
upper	strfun
calendar	timefun
clock	timefun
cputime	timefun
date	timefun
datenum	timefun
datestr	timefun
datevec	timefun
eomday	timefun
etime	timefun
now	timefun
pause	timefun
weekday	timefun
iqr	timeseries

Table 3 lists the MATLAB® functions available for Star-P® Release 2.7 Default Task-Parallel Engine (TPE) for x86/64-based Servers.

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
blkdiag	Arrays and Matrices
compan	Arrays and Matrices
cross	Arrays and Matrices
cumprod	Arrays and Matrices
cumsum	Arrays and Matrices
diag	Arrays and Matrices
dot	Arrays and Matrices

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
eye	Arrays and Matrices
full	Arrays and Matrices
hadamard	Arrays and Matrices
hankel	Arrays and Matrices
horzcat	Arrays and Matrices
ind2sub	Arrays and Matrices
ipermute	Arrays and Matrices
issparse	Arrays and Matrices
length	Arrays and Matrices
logspace	Arrays and Matrices
magic	Arrays and Matrices
ndims	Arrays and Matrices
nnz	Arrays and Matrices
nonzeros	Arrays and Matrices
numel	Arrays and Matrices
ones	Arrays and Matrices
pascal	Arrays and Matrices
permute	Arrays and Matrices
pinv	Arrays and Matrices
rand	Arrays and Matrices
randn	Arrays and Matrices
repmat	Arrays and Matrices
reshape	Arrays and Matrices
rosser	Arrays and Matrices
rot90	Arrays and Matrices
shiftdim	Arrays and Matrices
size	Arrays and Matrices
squeeze	Arrays and Matrices

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
sub2ind	Arrays and Matrices
sum	Arrays and Matrices
toeplitz	Arrays and Matrices
vander	Arrays and Matrices
vectorize	Arrays and Matrices
vertcat	Arrays and Matrices
wilkinson	Arrays and Matrices
zeros	Arrays and Matrices
conv2	Data Analysis
cov	Data Analysis
cumtrapz	Data Analysis
del2	Data Analysis
detrend	Data Analysis
diff	Data Analysis
fft	Data Analysis
fft2	Data Analysis
fftn	Data Analysis
fftshift	Data Analysis
filter	Data Analysis
filter2	Data Analysis
gradient	Data Analysis
hist	Data Analysis
histc	Data Analysis
ifft	Data Analysis
ifft2	Data Analysis
ifftn	Data Analysis

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
ifftshift	Data Analysis
issorted	Data Analysis
mean	Data Analysis
median	Data Analysis
mldivide	Data Analysis
mrdivide	Data Analysis
quad	Data Analysis
quadl	Data Analysis
randperm	Data Analysis
rcond	Data Analysis
sort	Data Analysis
sortrows	Data Analysis
std	Data Analysis
trapz	Data Analysis
var	Data Analysis
cat	Data Types
cell	Data Types
cell2mat	Data Types
cell2struct	Data Types
cellfun	Data Types
cellstr	Data Types
char	Data Types
class	Data Types
deal	Data Types
dec2base	Data Types
dec2bin	Data Types

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
dec2hex	Data Types
fieldnames	Data Types
findstr	Data Types
getfield	Data Types
hex2dec	Data Types
int16	Data Types
int2str	Data Types
int32	Data Types
int64	Data Types
int8	Data Types
intmax	Data Types
intmin	Data Types
isa	Data Types
iscell	Data Types
iscellstr	Data Types
ischar	Data Types
isequal	Data Types
isequalwithequalnans	Data Types
isfield	Data Types
isfinite	Data Types
isfloat	Data Types
isinf	Data Types
isinteger	Data Types
islogical	Data Types
isnan	Data Types
isnumeric	Data Types
isreal	Data Types
isscalar	Data Types

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
isstr	Data Types
isstruct	Data Types
isvector	Data Types
mat2cell	Data Types
mat2str	Data Types
num2cell	Data Types
num2hex	Data Types
num2str	Data Types
orderfields	Data Types
rmfield	Data Types
setfield	Data Types
setstr	Data Types
str2double	Data Types
str2mat	Data Types
str2num	Data Types
struct	Data Types
struct2cell	Data Types
subsasgn	Data Types
subsref	Data Types
substruct	Data Types
uint16	Data Types
uint32	Data Types
uint64	Data Types
uint8	Data Types
FALSE	Data Types
TRUE	Data Types
calendar	Date and Time

Table 3 **Default TPE Functions for x86/64**

MATLAB Function	Function Class
clock	Date and Time
cputime	Date and Time
date	Date and Time
datenum	Date and Time
datevec	Date and Time
eomday	Date and Time
etime	Date and Time
now	Date and Time
tic	Date and Time
toc	Date and Time
weekday	Date and Time
addpath	Desktop Tools
cd	Desktop Tools
chdir	Desktop Tools
clear	Desktop Tools
delete	Desktop Tools
dir	Desktop Tools
dos	Desktop Tools
fileattrib	Desktop Tools
fileparts	Desktop Tools
filesep	Desktop Tools
format	Desktop Tools
fullfile	Desktop Tools
getenv	Desktop Tools
isdir	Desktop Tools
ispc	Desktop Tools

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
isunix	Desktop Tools
load	Desktop Tools
mfilename	Desktop Tools
mkdir	Desktop Tools
path	Desktop Tools
pathsep	Desktop Tools
pwd	Desktop Tools
rmdir	Desktop Tools
rmpath	Desktop Tools
save	Desktop Tools
setenv	Desktop Tools
system	Desktop Tools
tempdir	Desktop Tools
tempname	Desktop Tools
unix	Desktop Tools
ver	Desktop Tools
version	Desktop Tools
which	Desktop Tools
who	Desktop Tools
whos	Desktop Tools
abs	Elementary Math
acos	Elementary Math
acosd	Elementary Math
acosh	Elementary Math
acot	Elementary Math
acotd	Elementary Math
acoth	Elementary Math

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
acsc	Elementary Math
acscd	Elementary Math
acsch	Elementary Math
airy	Elementary Math
all	Elementary Math
and	Elementary Math
angle	Elementary Math
any	Elementary Math
asec	Elementary Math
asecd	Elementary Math
asech	Elementary Math
asin	Elementary Math
asind	Elementary Math
asinh	Elementary Math
atan	Elementary Math
atan2	Elementary Math
atand	Elementary Math
atanh	Elementary Math
besselh	Elementary Math
besseli	Elementary Math
besselj	Elementary Math
besselk	Elementary Math
bessely	Elementary Math
beta	Elementary Math
betainc	Elementary Math
betaln	Elementary Math
bitand	Elementary Math
bitcmp	Elementary Math

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
bitget	Elementary Math
bitmax	Elementary Math
bitor	Elementary Math
bitset	Elementary Math
bitshift	Elementary Math
bitxor	Elementary Math
complex	Elementary Math
conj	Elementary Math
conv	Elementary Math
cos	Elementary Math
cosd	Elementary Math
cosh	Elementary Math
cot	Elementary Math
cotd	Elementary Math
coth	Elementary Math
cplxpair	Elementary Math
csc	Elementary Math
cscd	Elementary Math
csch	Elementary Math
ctranspose	Elementary Math
deconv	Elementary Math
eps	Elementary Math
eq	Elementary Math
erf	Elementary Math
erfc	Elementary Math
erfcx	Elementary Math
erfinv	Elementary Math
exp	Elementary Math

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
factor	Elementary Math
factorial	Elementary Math
find	Elementary Math
fix	Elementary Math
flipdim	Elementary Math
fliplr	Elementary Math
flipud	Elementary Math
floor	Elementary Math
gamma	Elementary Math
gammainc	Elementary Math
gammaln	Elementary Math
gcd	Elementary Math
ge	Elementary Math
gt	Elementary Math
hypot	Elementary Math
i	Elementary Math
imag	Elementary Math
inf	Elementary Math
intersect	Elementary Math
ipermute	Elementary Math
ismember	Elementary Math
isprime	Elementary Math
j	Elementary Math
lcm	Elementary Math
ldivide	Elementary Math
le	Elementary Math
legendre	Elementary Math
linspace	Elementary Math

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
log	Elementary Math
log10	Elementary Math
log1p	Elementary Math
log2	Elementary Math
lt	Elementary Math
max	Elementary Math
min	Elementary Math
minus	Elementary Math
mod	Elementary Math
mpower	Elementary Math
nan	Elementary Math
nchoosek	Elementary Math
ne	Elementary Math
nextpow2	Elementary Math
not	Elementary Math
nthroot	Elementary Math
or	Elementary Math
perms	Elementary Math
pi	Elementary Math
plus	Elementary Math
polyder	Elementary Math
polyfit	Elementary Math
polyval	Elementary Math
polyvalm	Elementary Math
pow2	Elementary Math
power	Elementary Math
primes	Elementary Math
prod	Elementary Math

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
rat	Elementary Math
rdivide	Elementary Math
real	Elementary Math
realmax	Elementary Math
realmin	Elementary Math
rem	Elementary Math
residue	Elementary Math
round	Elementary Math
sec	Elementary Math
secd	Elementary Math
sech	Elementary Math
setdiff	Elementary Math
setxor	Elementary Math
sign	Elementary Math
sin	Elementary Math
sind	Elementary Math
sinh	Elementary Math
sqrt	Elementary Math
tan	Elementary Math
tand	Elementary Math
tanh	Elementary Math
times	Elementary Math
transpose	Elementary Math
union	Elementary Math
unique	Elementary Math
unwrap	Elementary Math
uplus	Elementary Math
xor	Elementary Math

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
csvread	FileIO
csvwrite	FileIO
disp	FileIO
dlmread	FileIO
dlmwrite	FileIO
fclose	FileIO
feof	FileIO
ferror	FileIO
fgetl	FileIO
fgets	FileIO
fopen	FileIO
fprintf	FileIO
fputs	FileIO
fread	FileIO
frewind	FileIO
fscanf	FileIO
fseek	FileIO
ftell	FileIO
fwrite	FileIO
ls	FileIO
contour	Graphics
contourc	Graphics
inpolygon	Graphics
cart2pol	Interpolation and Computational Geometry

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
cart2sph	Interpolation and Computational Geometry
interp1	Interpolation and Computational Geometry
interp2	Interpolation and Computational Geometry
interpft	Interpolation and Computational Geometry
meshgrid	Interpolation and Computational Geometry
mkpp	Interpolation and Computational Geometry
ndgrid	Interpolation and Computational Geometry
pchip	Interpolation and Computational Geometry
pol2cart	Interpolation and Computational Geometry
ppval	Interpolation and Computational Geometry
pwch	Interpolation and Computational Geometry
sph2cart	Interpolation and Computational Geometry
spline	Interpolation and Computational Geometry
unmkpp	Interpolation and Computational Geometry
accumarray	Linear Algebra
balance	Linear Algebra
chol	Linear Algebra
cholupdate	Linear Algebra

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
circshift	Linear Algebra
cond	Linear Algebra
det	Linear Algebra
eig	Linear Algebra
ellipke	Linear Algebra
expm	Linear Algebra
hess	Linear Algebra
hilb	Linear Algebra
inv	Linear Algebra
invhilb	Linear Algebra
kron	Linear Algebra
linsolve	Linear Algebra
logm	Linear Algebra
lu	Linear Algebra
mpower	Linear Algebra
mtimes	Linear Algebra
norm	Linear Algebra
null	Linear Algebra
ordeig	Linear Algebra
orth	Linear Algebra
planerot	Linear Algebra
poly	Linear Algebra
qr	Linear Algebra
qrdelete	Linear Algebra
qrinsert	Linear Algebra
qz	Linear Algebra
rank	Linear Algebra
roots	Linear Algebra

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
rref	Linear Algebra
schur	Linear Algebra
sqrtm	Linear Algebra
ss2tf	Linear Algebra
svd	Linear Algebra
trace	Linear Algebra
tril	Linear Algebra
triu	Linear Algebra
ode23	ODE
ode45	ODE
ode78	ODE
odeset	ODE
'	Operators and Special Characters
-	Operators and Special Characters
!	Operators and Special Characters
%	Operators and Special Characters
%{ %}	Operators and Special Characters
&	Operators and Special Characters
&&	Operators and Special Characters
( )	Operators and Special Characters
*	Operators and Special Characters
,	Operators and Special Characters
.	Operators and Special Characters
.'	Operators and Special Characters

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
.( )	Operators and Special Characters
.*	Operators and Special Characters
..	Operators and Special Characters
...	Operators and Special Characters
./	Operators and Special Characters
.\	Operators and Special Characters
.^	Operators and Special Characters
/	Operators and Special Characters
:	Operators and Special Characters
;	Operators and Special Characters
@	Operators and Special Characters
[ ]	Operators and Special Characters
\	Operators and Special Characters
^	Operators and Special Characters
{ }	Operators and Special Characters
|	Operators and Special Characters
||	Operators and Special Characters
~	Operators and Special Characters
~=	Operators and Special Characters
+	Operators and Special Characters
<	Operators and Special Characters
<=	Operators and Special Characters
=	Operators and Special Characters
==	Operators and Special Characters
>	Operators and Special Characters
>=	Operators and Special Characters

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
assert	Programming
cast	Programming
else	Programming
elseif	Programming
end	Programming
error	Programming
exist	Programming
feval	Programming
for	Programming
func2str	Programming
global	Programming
if	Programming
lasterr	Programming
lasterror	Programming
lastwarn	Programming
nargchk	Programming
nargin	Programming
nargout	Programming
persistent	Programming
rethrow	Programming
return	Programming
str2func	Programming
switch	Programming
try	Programming
typecast	Programming
varargin	Programming
varargout	Programming
warning	Programming

Table 3 Default TPE Functions for x86/64

MATLAB Function	Function Class
while	Programming
blanks	String Functions
deblank	String Functions
isletter	String Functions
isspace	String Functions
lower	String Functions
regexp	String Functions
regexprep	String Functions
sprintf	String Functions
sscanf	String Functions
strcat	String Functions
strcmp	String Functions
strcmpi	String Functions
strfind	String Functions
strjust	String Functions
strmatch	String Functions
strncmp	String Functions
strncmpi	String Functions
strrep	String Functions
strtok	String Functions
strtrim	String Functions
strvcat	String Functions
upper	String Functions

Appendix A

Application Examples

Application Example: Image Processing Algorithm

The application examples in this section show pattern matching for an input image and a target image using the Fourier transform of the image, or, in basic terms, Fourier pattern matching.

The program performs Fourier analysis of an input image and a target image. This analysis tries to locate the target image within the input image. Correlation peaks show where the target image exists. The output matrix shows where high correlation exists in the Fourier plane. In other words, X marks the spot.

How the Analysis Is Done

The analysis in this simplified application takes the transform of the input and target images, multiplies the elements of the transforms, and then transforms the product back. This results in correlation peaks located where the target image is located within the input image. Since the image is in color, the processing is performed within three different color spaces, correlation matches occur three times. Strong peaks exist in the image along with the possibility of some noise. To further data reduce the image, a threshold is used which reduces the information to a two dimensional (2D) binary map. The image of the 2D binary map reduces the three color space images into a single binary map indicating the locations of the target image. The location of the ones (1) indicate the position of the target image within the input image.In this example, the ones exist in four separate clusters and the centroid of each cluster indicates the center location of the target image.

Application Examples

There are three application examples given in this section:

- An example not using Star-P®, see "Application Example Not Using Star-P®".
- An example using `*p` to distribute the computation, see "Application Example Using Star-P®".
- An example using `ppeval` to distribute the computation see "Application Example Using ppeval".

Images For Application Examples

The images used for the examples are shown in the figures below.

Figure A-1: Target Image

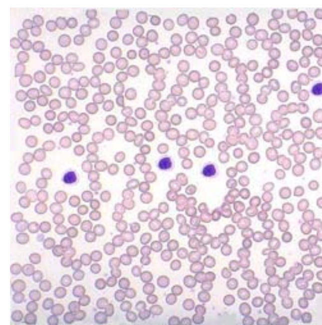


Figure A-2: Input Image

M Files for the Application Examples

There are two `.m` files used in each example. The files used for each example are as follows:

- Without Star-P® Example uses:
    - `patmatch_colordemo_noStarP.m`

    - `patmatch_calc.m`
- With Star-P® Example uses:
    - `patmatch_colordemo_StarP.m`
    - `patmatch_calc.m`
- ppeval example uses:
    - `patmatch_color_ppeval.m`
    - `patmatch_calc.m`

Note: The `patmatch_calc.m` file is the same for all three examples.

M files are text files which typically contain the following information:

File Element	Description
Function definition line	Informs MATLAB that the M-file contains a function. This line defines function number and the number and order of input and output arguments
Function or script body	Program code that performs the actual computations and assigns values to any output arguments
Comments	Text in the body of the program that explains the internal workings of the program

Application Example Not Using Star-P®

The following provides the actual flow for this application example where Star-P® is not used. The M files associated with this example are shown immediately after this table.

Step	Description
1	The input image Figure A-2: is separated into Hue, Saturation and Value (HSV).
2	The image is tiled and replicated. The color constituent parts are each replicated in a tiling fashion to make a larger H, S, and V images.

Step	Description
3	A correlation calculation is performed on the HSV components of the input and target images. The `patchmatch_calc.m` file is called. A pattern matching calculation is used. This particular function is called for each of the three HSV images. a. The function correlates the input and target image by padding the target image, which is assumed to be a smaller image. It is padded with bright regions or ones (1). • 1 represents background • 0 represents lack of background Basically, the size input image is found and then the target image is padded to that size. The padded image is shifted into Fourier space assuring accuracy. b. The actual correlation is done by multiplying the Fourier transform input image times the complex conjugate of the target image. Next it takes the inverse transform of the two images and that creates the amplitude and phase of the correlation. To create the observable image, this product image is multiplied by its complex conjugate completing the correlation. c. Once the correlation calculation is complete, the correlation image is scaled between zero and one for each of the HSV components.
4	A threshold operation is performed to find the target locations within the input image. The operation is performed for each of the HSV components and is done empirically to achieve the display result through a map of the input image.
5	Displays the results which is a fully reduced, binary map of the target image location.

patmatch_color_noStarP.m File

The following is the sample file that contains the program code for the application example. The numbers on the left correspond to the table in the previous section.

```
>> type patmatch_colordemo_noStarP
% Setup the variables
imr = 1; imc = 1;                % Set number of image tiled in rows and columns
target = 'Twhite.PNG';
img = '500x500.PNG';
```

1 Read in RGB data, convert to HSV

2 Image is tiled and replicated

3 Calculate the correlation on each of HSV components

4 Perform the threshold

5 Display the results

```
thres = 0.85;
% Load the data, comes in RGB, transfer to HSV space
a = rgb2hsv(imread(img    ));       % Get the image containing targets
b = rgb2hsv(imread(target));        % Get the filter mask
% Setup the input image tiling problem
if imr > 1 || imc > 1
    a = repmat(a,imr,imc);
end
% Perform correlation calculation in HSV space
d = zeros(size(a));
for i = 1:3
    d(:,:,i) = patmatch_calc(a(:,:,i),b(:,:,i));
end
% Threshold for finding target within input image
e = (1-d(:,:,2)) > 0.5 & d(:,:,3) > thres;
%Display the result
figure(1);
imagesc(hsv2rgb(a)); colormap jet;  title('Input Image');
figure(2);
imagesc(hsv2rgb(b)); colormap jet;  title('Filter Pattern');
figure(3);
imagesc(d(:,:,1));   colormap jet;  title('Correlation H');
figure(4);
imagesc(1-d(:,:,2)); colormap jet;  title('Correlation S');
figure(5);
imagesc(d(:,:,3));   colormap jet;  title('Correlation V');
figure(6);
imagesc(e);          colormap gray; title('Threshold Correlation');
```

patmatch_calc.m

This is the contents of the calculation file that is called by `patmatch_color_noStarP.m`, `patmatch_color_StarP.m`, and `patmatch_color_ppeval.m`.

3a Set up and pad for correlation

3b Calculate the correlation

```
>> type patmatch_calc.m
function corr = patmatch_calc(a,b)
%
% Pad the target input with bright areas to the size of the input image
%
[I,J]=ind2sub(size(b),1:numel(b));
pad = ones(size(a));
% Pad the target with ones, bright, to size of 'a'
pad(sub2ind(size(pad),floor(size(pad,1)/2)+(I-floor(size(b,1)/2)), ...
                      floor(size(pad,2)/2)+(J-floor(size(b,2)/2))))=b;
% Adjust the filter to the FFT space
pad = fftshift(pad);
% Calculate the pattern match of input image a with the target filter b
% Multiply Fourier transform of the input and target
c = ifft2(fft2(a).*conj(fft2(pad)));
% Measured optical intensity
d = c.*conj(c);
% Normalize the image to the tallest peak
```

3c
Scale between 0 and 1

```
corr = (d-min(min(d)))/max(max(d-min(min(d))));
```

Application Example Using Star-P®

The following provides the actual flow for this application example using Star-P®. The M files associated with this example are shown immediately after this table.

Step	Description
1	The input image is loaded and separated as previously described in "Application Example Not Using Star-P®". The main difference is that each of these images are transferred to the backend (server or HPC). From this point every subsequent operation or computation that occurs will occur on the backend.
2	This tiled image is now created on the backend. See "Application Example Not Using Star-P®".
3	The correlation calculation as described previously for "Application Example Not Using Star-P®" is performed on the backend. The `patmatch_calc.m` file is identical as for "Application Example Not Using Star-P®" except the calculation is performed on the backend. No changes required.
4	The threshold operation is performed on the backend (see "Application Example Not Using Star-P®").
5	The `ppfront` function moves the data to the frontend or client for viewing. (see "Application Example Not Using Star-P®").

patmatch_colordemo_StarP.m File

The following is the sample file that contains the program code for the application example. The numbers on the left correspond to the table in the previous section. Only the differences from the "Application Example Not Using Star-P®" are described.

```
>> type patmatch_colordemo_StarP
% Setup the variables
imr = 1; imc = 1;              % Set number of image tiled in rows and columns
target = 'Twhite.JPG';
img = '500x500.JPG';
thres = 0.85;
% Load the data, comes in RGB, transfer to HSV space
a = rgb2hsv(imread(img   ));   % Get the image containing targets
b = rgb2hsv(imread(target));   % Get the filter mask
% Transfer image data to the server
```

1
Image is transferred to back-end

```
a = ppback(a);
% Setup the input image tilling problem
if imr > 1 | imc > 1
    a = repmat(a,imr,imc);
end
% Perform correlation calculation in HSV space
d = zeros(size(a));
for i = 1:3
    d(:,:,i) = patmatch_calc(a(:,:,i),b(:,:,i));
end
% Threshold for finding target within input image
e = (1-d(:,:,2)) > 0.5 & d(:,:,3) > thres;
% Transfer results to the client
a = ppfront(a);
d = ppfront(d);
e = ppfront(e);
% Display the result
figure(1);
imagesc(hsv2rgb(a)); colormap jet;  title('Input Image');
figure(2);
imagesc(hsv2rgb(b)); colormap jet;  title('Filter Pattern');
figure(3);
imagesc(d(:,:,1));   colormap jet;  title('Correlation H');
figure(4);
imagesc(1-d(:,:,2)); colormap jet;  title('Correlation S');
figure(5);
imagesc(d(:,:,3));   colormap jet;  title('Correlation V');
figure(6);
imagesc(e);          colormap gray; title('Threshold Correlation');
```

5
Image is transferred from back-end

Application Example Using ppeval

The following provides the actual flow for this application example using `ppeval`. The M files associated with this example are shown immediately after the table.

About ppeval

`ppeval` executes embarrassingly parallel operations in a task parallel mode. The tasks are completely independent and are computed individually, with access only to local data. For example, if there are four function evaluations to be computed and Star-P® has four processors allocated, `ppeval` takes the function to be evaluated and sends it to each of the four processors for calculation.

About the ppeval Example

This function takes the HSV components for the input and target images and calculates all the correlations for each of these components simultaneously.

The technical explanation of the computation is identically the same as the previous example and is eliminated for brevity. The key difference in using the `patmatch_calc` function is the setup of `ppeval` that calls this function.

In the case of item 5, ppeval calls patmach_calc with the input image a and target image b. The parallelization is performed with the split function that breaks the input and target images into their respective HSV components. The split in each case is along the 3rd dimension. If you have three processors, processor 1 gets the H component, processor 2 gets the S component, and processor 3 gets the V component.

When `ppeval` executes, `patmatch_calc` is executed simultaneously on three processors.

Step	Description
1	The operation is the same as described for the previous two examples.
2	Not included for the `ppeval` because tiling to larger images or working with larger input images on a single processor limits the performance gains achieved by single processor calculation. In other words, single processor calculations provide performance on small data sizes.
3	The correlation calculation as described for the previous two examples is performed on an individual processor on the backend.
4	The operation is the same as described for the previous two examples.
5	The operation is the same as described for the previous two examples.

patmatch_color_ppeval.m

The following is the sample file that contains the program code for the application example. The numbers on the left correspond to the table in the previous section. Only the differences from the "Application Example Not Using Star-P®" are described.

2 Tiling is not included. It limits performance gains.

3 Correlation calculations performed on three backend processors

```
>> type patmatch_colordemo_ppeval
% Setup the variables
imr = 1; imc = 1;              % Set number of image tiled in rows and columns
target = 'Twhite.JPG';
img = '500x500.JPG';
thres = 0.85;
% Load the data, comes in RGB, transfer to HSV space
a = rgb2hsv(imread(img    ));   % Get the image containing targets
b = rgb2hsv(imread(target));   % Get the filter mask
% Setup the input image tiling problem
```

```
if imr > 1 | imc > 1
    a = repmat(a,imr,imc);
end
% Perform correlation calculation in HSV space
d = ppeval('patmatch_calc',split(a,3),split(b,3));
% Threshold for finding target within input image
e = (1-d(:,:,2)) > 0.5 & d(:,:,3) > thres;
% Transfer results to the client
d = ppfront(d);
e = ppfront(e);
% Display the result
figure(1);
imagesc(hsv2rgb(a)); colormap jet;  title('Input Image');
figure(2);
imagesc(hsv2rgb(b)); colormap jet;  title('Filter Pattern');
figure(3);
imagesc(d(:,:,1));   colormap jet;  title('Correlation H');
figure(4);
imagesc(1-d(:,:,2)); colormap jet;  title('Correlation S');
figure(5);
imagesc(d(:,:,3));   colormap jet;  title('Correlation V');
figure(6);
imagesc(e);          colormap gray; title('Threshold Correlation');
```

INTERACTIVE supercomputing

Appendix B

Solving Large Sparse Matrix and Combinatorial Problems with Star-P®

This chapter introduces a mode of thinking about a large class of combinatorial problems. Star-P® can be considered as a potential tool whenever you are faced with a discrete problem where quantitative information is extracted from a data structure such as those found on networks or in databases.

Sparse matrix operations are widely used in many contexts, but what is less well known is that these operations are powerfully expressive for formulating and parallelizing combinatorial problems. This chapter covers the basic theory and illustrates a host of examples. In many ways this chapter extends the notion that array syntax is more powerful than scalar syntax by applying this syntax to the structures of a class of real-world problems.

At the mathematical level, a sparse matrix is simply a matrix with sufficiently many zeros that it is sensible to save storage and operations by not storing the zeros or performing unnecessary operations on zero elements such as x+0 or x*0. For example, the discretization of partial differential equations typically results in large sparse linear systems of equations. Sparse matrices and the associated algorithms are particularly useful for solving such problems.

Sparse matrices additionally specify connections and relations among objects. Simple discrete operations including data analysis, sorting, and searching can be expressed in the language of sparse matrices.

Graphs and Sparse Matrices

Graphs are used for networks and relationships. Sparse matrices are the data structures used to represent graphs and to perform data analysis on large data sets represented as graphs.

Graphs: It’s all in the connections

In the following discussion, a “graph” is simply a group of discrete entities and their connections. While standard, the term is not especially illuminating, so it may be helpful to consider a graph as a “network”. Think of a phone network or a computer network or a social

network. The most important thing to know are the names and who is connected to whom. Formally, a graph is a set of nodes and edges. It is the information of who can directly influence whom or at least who has a link to whom.

Consider the route map of an airline. The nodes are cities, the edges are plane routes.

The earth is a geometrical object, i.e. continuous, yet the important information for the airline is the graph, the discrete object connecting the relevant cities. Next time you see a subway map, think of the graph connecting the train stops. Next time you look at a street map think of the intersections as nodes, and each street as an edge.

Electrical circuits are graphs. Connect up resistors, batteries, diodes, and inductors. Ask questions about the resistance of the circuit. In high school one learns to follow Ohm's law and Ampere's law around the circuit. Graph theory gives the bigger picture. We can take a large grid of resistors and connect a battery across one edge. Looked at one way, this is a discrete man-made problem requiring a purchase of electrical components.

The internet is a great source for graphs. We could have started with any communications network: telegraphs, telephones, smoke signals... but let us consider the internet. The internet can be thought of as the physical links between computers. The current internet is composed of various subnetworks of connected computers that are connected at various peering points. Run traceroute from your machine to another machine and take a walk along the edges of this graph.

More exciting than the hardware connections are the virtual links. Any web page is a node; hyperlinks take us from one node to another. Web pages live on real hardware, but there is no obvious relationship between the hyperlinks connecting web pages and the wires connecting computers.

The graph that intrigues us all is the social graph: in its simplest form, the nodes are people. Two people are connected if they know each other.

A graph may be a discretization of a continuous structure. Think of the graph whose vertices are all the USGS benchmarks in North America, with edges joining neighboring benchmarks. This graph is a mesh: its vertices have coordinates in Euclidean space, and the discrete graph approximates the continuous surface of the continent. Finite element meshes are the key to solving partial differential equations on (finite) computers.

Graphs can represent computations. Compilers use graphs whose vertices are basic blocks of code to optimize computations in loops. The heart of a finite element computation might be the sparse matrix-vector multiplication in an iterative linear solver; the pattern of data dependency in the multiplication is the graph of the mesh.

Oftentimes graphs come with labels on their edges (representing length, resistance, cost) or vertices (name, location, cost).

There are so many examples -- some are discrete from the start, others are discretizations of continuous objects, but all are about connections.

Sparse Matrices: Representing Graphs and General Data Analysis

Consider putting everybody at a party in a circle holding hands and each person rates how well they know the person on the left and right with a number from 1 to 10.

Each person can be represented with the index i, and the rating of the person on the right can be listed as $A_{i,i+1}$ while the person on the left is listed as $A_{i,i-1}$.

As an example;

```
PERSON        Right  Left
1             5      6
2             3      2
3             1      9
4             2      7
```

In serial MATLAB

```
>> i = [1 2 3 4]; j = [2 3 4 1]; k = [4 1 2 3];
>> r = [5 3 1 2];
>> l = [6 2 9 7];
>> sparse([i i],[j k], [r l])
ans =
   (2,1)        2
   (4,1)        2
   (1,2)        5
   (3,2)        9
   (2,3)        3
   (4,3)        7
   (1,4)        6
   (3,4)        1
>> full(ans)
ans =
     0     5     0     6
     2     0     3     0
     0     9     0     1
     2     0     7     0
```

With Star-P®

```
>> n = 1000*p;
>> i = 1:n;
>> j = ones(1,n);
>> j(1,1:end-1) = i(1,2:end); j(1,end) = i(1,1);
>> k = ones(1,n);
>> k(1,2:end) = i(1,1:end-1); k(1,1) = i(1,end);
>> r = rand(1,n);
>> l = rand(1,n);
>> A = sparse([i i], [j k], [r l])
A =
        dsparse object: 1000p-by-1000
```

The next example illustrates a circular network with unsymmetric weights.

```
>> B=spones(A)
```

gives the network without weights, and

```
>> [i,j] = find(B)
i =
        ddense object: 2000p-by-1
j =
        ddense object: 2000p-by-1
```

undoes the sparse construction.

Data Analysis and Comparison with Pivot Tables

Consider the following “database” style application:

Imagine we have an airline that flies certain routes on certain days of the week and we are interested in the revenue per route and per day. We begin with a table which can be simply an n x 3 array:

```
Route      Day    Revenue in Thousands
1          0      3
1          1      5
1          3      4
1          5      5
2          1      3
2          2      3
2          4      3
2          6      3
3          6      4
3          0      4
```

In Microsoft Excel, there is a little known feature that is readily available on the Data menu called PivotTable which allows for the analysis of such data.

MATLAB and Star-P® users can perform the same analysis with sparse matrices.

First we define the three column array:

```
>> m = [ 1 0 3; 1 1 5; 1 3 4; 1 5 5; 2 1 3; 2 2 3; 2 4 3; 2 6 3; 3 6 4; 3 0 4]
m =
     1     0     3
     1     1     5
     1     3     4
     1     5     5
     2     1     3
     2     2     3
     2     4     3
     2     6     3
```

```
     3     6     4
     3     0     4
```

Then we create the sparse matrix a:

```
>> a = sparse(m(:,1),m(:,2)+1,m(:,3))
a =
   (1,1)         3
   (3,1)         4
   (1,2)         5
   (2,2)         3
   (2,3)         3
   (1,4)         4
   (2,5)         3
   (1,6)         5
   (2,7)         3
   (3,7)         4
```

How much does each of the three routes make as revenue:

```
>> sum(a')
ans =
   (1,1)        17
   (1,2)        12
   (1,3)         8
```

Or what is the total for each day:

```
>> sum(a)
ans =
   (1,1)         7
   (1,2)         8
   (1,3)         3
   (1,4)         4
   (1,5)         3
   (1,6)         5
   (1,7)         7
```

Or what is the total revenue:

```
 >> sum(a(:))
ans =
     (1,1)        37
```

Since Star-P® extends the functionality of sparse matrices to parallel machines, one can do very sophisticated data analysis on large data sets using Star-P®.

Note that the `sparse` command also adds data with duplicate indices.

If the `sparse` constructor encounters duplicate (i,j) indices, the corresponding nonzero values are added together. This is sometimes useful for data analysis; for example, here is an

example of a routine that computes a weighted histogram using the `sparse` constructor. In the routine, `bin` is a vector that gives the histogram bin number into which each input element falls. Notice that `h` is a sparse matrix with just one column! However, all the values of `w` that have the same bin number are summed into the corresponding element of `h`. The MATLAB function `bar` plots a bar chart of the histogram.

```
>> type histw
function [yc, h] = histw(y, w, m)
% HISTW Weighted histogram.
%    [YC, H] = HISTW(Y, W, M) plots a histogram of the data in Y weighted
%    with W, using M boxes. The output YC contains the bin centers, and
%    H the sum of the weights in each bin.
%
%    Example:
%       y = rand(1e5,1);
%       histw(y,y.^2,50);
dy = (max(y) - min(y)) /  m;
bin = max(min(floor((y - min(y)) / dy) + 1, m), 1);
yy = min(y) + dy * (0:m);
yc = (yy(1:end-1) + yy(2:end)) / 2;
h = sparse(bin, 1, w, m, 1);
bar(yc, full(h));
```

Multiplication of a sparse matrix by a dense vector (sometimes called “matvec”) turns out to be useful for many kinds of data analysis that have nothing directly to do with linear algebra. We will see several examples later that have to do with paths or searches in graphs. Here is a simple example that has to do with the nonzero structure of a matrix.

Suppose `G` is a dsparse matrix with `nr` rows and `nc` columns. For each row, we want to compute the average of the column indices of the nonzeros in that row (or zero if the whole row is zero, say). The result will be a `ddense` vector with `nr` elements. The following code does this. (The first line replaces each nonzero in `G` with a one; it can be omitted if, say, `G` is the adjacency matrix of a graph or a 0/1 logical matrix.)

```
>> Gpp = sprandn(1e6*p,1e4,0.01);
>> Gpp = spones(Gpp);
>> [nr, nc] = size(Gpp);
>> vpp = (1:nc*p)';
>> epp = ones(nc*p,1);
>> rowcounts = Gpp * epp;
>> indexsums = Gpp * vpp;
>> averageindex = indexsums ./ max(rowcounts, 1);
```

Since **`epp`** is a column of all ones, the first matvec **`Gpp`**`*`**`epp`** computes the number of nonzeros in each row of **`Gpp`**. The second matvec **`Gpp`**`*`**`vpp`** computes the sum of the column indices of the nonzeros in each row. The `*max*` in the denominator of the last line makes `averageindex` zero whenever a row has no nonzeros.

Laplacian Matrices and Visualizing Graphs

The Laplacian matrix is a matrix associated with an undirected graph. Like the adjacency matrix, it is square and symmetric and has a pair of nonzeros (i,j) and (j,i) for each edge (i,j) of the graph. However, the off-diagonal nonzero elements of the Laplacian all have value -1, and the diagonal element $L_{i,i}$ is the number of edges incident on vertex i. If A is the adjacency matrix of an undirected graph, one way to compute the Laplacian matrix is with the following:

```
>> L = -spones(A);
>> L = L - diag(diag(L));
>> L = L + diag(sum(L));
```

This code is a little more general than it needs to be -- it doesn't assume that all the nonzeros in A have value 1, nor does it assume that the diagonal of A is zero. If both of these are true, as in a proper adjacency matrix, it would be enough to say:

```
>> L = diag(sum(A)) - A;
```

The Laplacian matrix has many algebraic properties that reflect combinatorial properties of the graph. For example, it is easy to see that the sums of the rows of L are all zero, so zero is an eigenvalue of L (with an eigenvector of all ones). It turns out that the multiplicity of zero as an eigenvalue is equal to the number of connected components of the graph. The other eigenvalues are positive, so L is a positive semidefinite matrix. The eigenvector corresponding to the smallest nonzero eigenvalue has been used in graph partitioning heuristics.

For a connected graph, the eigenvectors corresponding to the three smallest Laplacian eigenvalues can be used as vertex coordinates (the coordinates of vertex number i are (x_i, y_i, z_i), where x, y, and z are the eigenvectors), and the result is sometimes an interesting picture of the graph. Figure B-1: is an example of this technique applied to the graph created in Kernel 1 of the SSCA#2 benchmark.

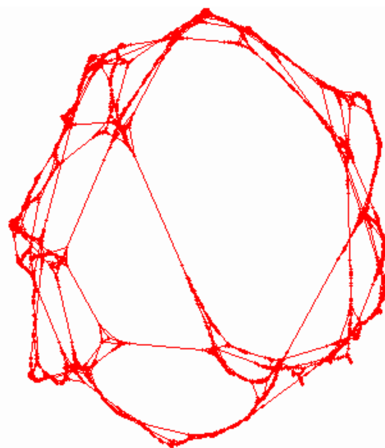


Figure B-1: 8192-vertex graph from Kern1 plotted with Fiedler coordinates

On Path Counting

You may want to know how many paths connect two nodes in a graph. The incidence matrix I is useful for this calculation, and is defined as:

I_{ij} = {1 if node i is connected to node j, otherwise 0}

The matrix a in "Sparse Matrices: Representing Graphs and General Data Analysis" is actually an adjacency matrix. For any particular path length k, each element of I^k represents the number of paths that connect node i to node j.

```
>> a = spones(sprandn(100*p,100,0.1));
>> b = a^3
b =
        ddense object: 100-by-100p
>> b(14,23)
ans =
     4
>> nnz(b)
ans =
        9928
```

In this example there are 11 paths of length 3 that connect nodes 14 and 23. Another characteristic of the graph that can be gleaned from this calculation is that almost all of the nodes are reachable from all other nodes with a path of length 3 (9946 out of 10000 entries).

Symbols
* , overloaded, **41**
*p syntax, **2**, **28**

A
about the Star-P® MATLAB® Programming Guide, **5**
accuracy of Star-P® routines, **89**
application example
  not using Star-P, **209**
  using ppeval, **213**
  using Star-P, **212**
application examples, **207**
array bounds, distributed, **30**
array bounds variable, distributed, **28**
assignments to p, **26**

B
bcast, **137**, **151**

C
C, **2**, **4**
C++, **2**, **4**
C++ as task parallel engine, **72**
calculus of distribution, **44**
calling non-MATLAB functions within ppeval, **73**
cell arrays, using in task parallel, **85**
changing/examining distributed matrices, **24**
circshift, **102**
client/server messages, excessive, **93**
client vs. server variables, ppeval, **66**
cluster configurations, command line options, **19**
coarse-grained parallelism, **59**
column distribution of matrices, **33**
combining data distribution mechanisms, **36**
command line options
  cluster configurations, **19**
command line options, examples, **18**
communication among processors in the parallel server, **101**
communication between the Star-P® client and server, **98**
communication dependencies, maintaining awareness of, **98**
complex number data, **27**
compressed sparse row format, **34**, **35**
computation of eigenvectors for non-Hermitian matrices, **90**
configuration, user specific Star-P® start-up, **14**
creating distributed arrays, **28**
cumsum, **82**

D
d
  Star-P® naming conventions, distrubtued, **22**
data distribution mechanisms, combining, **36**
data movement functions, **147**
data oarallelism with Star-P® and MATLAB, **2**
data parallelism with Star-P® and MATLAB, **21**
dcell, **36**
dddense
  propagation of distribution, **44**
ddense, **38**, **39**, **44**
ddensend, **34**, **38**, **44**
  propagation of distribution, **44**
deep copy vs. shallow copy, incompatibility of Star-P® and MATLAB, **32**
diag, **30**
distrbutions, types of, **32**
distributed and local data, mixing, **37**
distributed array bounds, **30**
distributed array bounds variable, **28**
distributed arrays, creating, **28**
distributed attribute, propagating the, **41**
distributed cell objects, **36**
distributed classes used by Star-P®, **38**
distributed data creation routines, **29**
distributed dense matrices and arrays, **32**
distributed dense multidimensional arrays, **34**
distributed matrices, examining/changing, **24**
distributed sparse matrices
  sparse matrices
    distributed, **34**
distribution, propagation of, **44**
dlayout, **28**, **38**, **40**
dsparse, **34**, **38**, **39**, **44**
  propagation of distribution, **44**

E
eigs, **41**
embarrassingly parallel, **4**
enhanced performance profiling in Star-P®,

103
examining/changing distributed matrices, **24**
examining Star-P® data, **22**
excessive client/server messages, **93**
excluding nodes in a cluster, **17**
explicit data movement with ppback and ppfront, **49**
extending MATLAB with Star-P®, **2**
external libraries in task parallel codes, **89**
eye, **29**

F
fclose, **53**
FEM, finite element method, **117**
FFT, **102**
fft, **41**
fopen, **53**
for loop into a ppeval call, transforming, **61**
Fortran, **2**, **4**, **78**
fread, **53**
frewind, **53**
fseek, **135**, **138**
fwrite, **53**

G
global array syntax, **4**
global variables for task parallel operations, **75**
graphs and sparse matrices, **217**
graphs - it's all in the connections, **217**

H
HDF5
  converting data from other formats to, **56**
  differences from MATLAB support, **56**
HDF5, Hierachical Data Format Version 5, **53**
HDF5, limitations with Star-P®, **56**
HDF5 file, querying variables stored inside, **55**
HDF5 file, reading variables from, **54**
HDF5 file, representation of data in, **55**
  complex data, **55**
  multidimensional arrays, **55**
  sparse matrices, **56**
HDF5 file, writing variables to, **54**
Hilbert matrix, **23**
histc, **41**, **82**
horzcat, **29**, **102**

I
ill-conditioned or singular operations, **90**
image processing algorithm, **207**
implicit communication, **99**
implicit data movement, **93**
indexing into distributed matrices or arrays, distributed matrices, indexing into, **30**
indexing operations, **48**
input arguments to ppeval, **64**

L
launching Star-P® with a MATLAB .m script, **19**
load, **28**
loading and saving data on the parallel server, **51**
local and distributed data, mixing, **37**
logical indexing, **49**

M
machine file
  path, **17**
  user default, **17**
MATLAB® functions, supported, **161**
MATLAB as task parallel engine, **91**
MATLAB with Star-P, extending, **2**
maximizing performance of Star-P® code, **98**
memory issues, MathWorks technical notes, **84**
memory issues, solving large problems, **84**
meshgrid, **29**
message passing, **4**
MIMD, **59**
mixing local and distributed data, **37**
monitor the server, UNIX commands, **132**
monte carlo simulations, **5**
MPI, **2**, **4**
multiple instruction multiple data, **59**

N
nnz, **31**
node-oriented languages, **4**
nodes in a cluster
  excluding, **17**
  specifying, **17**
  specifying a range, **17**
  specifying a set, **17**
non-Hermitian eigenvalue problems, **90**
non-MATLAB function within ppeval, calling, **73**

non-uniqueness of MATLAB and Star-P® routines, **89**
np, **25**, **139**
  Star-P® functions
    np, **135**

O

Octave as task parallel engine, **71**
ones, **29**, **37**, **41**
overloaded operators
  *, **41**
  ones, **41**

P

p, **24**, **135**, **139**
parallel computing 101, **4**
  data parallel computation, **4**
  message passing, **4**
  task parallel computation, **4**
password, user, **10**, **11**
patmatch_calc.m, **211**
patmatch_color_noStarP.m, **210**
patmatch_color_ppeval.m, **214**
patmatch_colordemo_Star-P.m, **212**
pattern matching, **207**
performance and productivity, **77**
performance bottlenecks, eliminating using ppperf, **117**
performance profiling in Star-P®, enhanced, **103**
performance tuning and monitoring, **91**
  client/server monitoring, **91**
  diagnostics and performance, **91**
permute, **102**
per processe execution for ppeval, **72**
pp, **26**, **135**, **139**
  Star-P® naming conventions, **22**
ppback, **28**, **136**, **147**
  warning messages, **51**
ppback, explicit data movement with, **49**
ppbench, **135**
ppchangedist, **137**, **148**
  warning messages, **51**
ppeval, **2**, **59**, **137**, **152**
  calling non-MATLAB functions within, **73**
  client vs. server variables, **66**
  distribution of input variables, **67**
  input arguments, **64**
    broadcasting, **65**
    default behavior, **64**
    splitting, **64**
  known differences between MATLAB and Octave functions, **153**
  output arguments, **67**
  per process execution, **72**
  requirements of functions passed to, **64**
  syntax and behavior, **63**
  transforming a for loop into a, **61**
ppeval_tic, **97**
ppeval_toc, **97**
ppeval, about, **59**
ppeval and ppevalc functions, the mechanism for task parallelism, **59**
ppevalc, **59**
ppevalcloadmodule, **138**, **154**
ppevalcunloadmodule, **138**, **154**
ppevalsplit, **69**, **137**, **154**
ppfopen, **52**, **136**, **145**
ppfront, **24**, **136**, **148**
  warning messages, **51**
ppfront, explicit data movement with, **49**
ppgetlog, **136**, **141**
ppgetlogpath, **136**, **142**
ppgetoption, **136**, **141**
pph5read, **54**, **137**, **150**
pph5whos, **55**, **136**, **146**
pph5write, **54**, **137**, **149**
ppinvoke, **136**, **143**
ppload, **28**, **51**, **137**, **150**
pploadpackage, **143**
ppperf, **103**, **106**, **131**, **138**, **155**
  displaying performance statistics, **106**
  gathering performance statistics, **106**
  graphical mode, **113**
    lessons learned, **117**, **131**
  interpretation of output, **108**
  lessons learned, **113**
  output preamble, **108**
  performance process measurement, **110**
  performance time measurement, **109**
  Star-P® functions
    ppperf, **155**
  using, **103**
  using to eliminate performance bottlenecks, **117**
ppperf clear, **106**, **132**, **155**
ppperf graph off, **132**, **156**

ppperf graph on, **123**, **156**
ppperf off, **106**, **132**, **155**
ppperf report, **106**, **107**, **132**, **156**
ppperf report detail, **107**, **132**, **156**
ppperf resume, **155**
ppprofile, **94**, **103**, **105**, **138**, **156**
ppquit, **136**, **145**
ppsave, **51**, **137**, **151**
ppsetoption, **136**, **141**
    configuring for high performance, **90**
    warning messages, **51**
ppstartup, **14**
pptic, **91**, **103**, **138**, **157**
pptoc, **91**, **103**, **138**, **157**
ppunloadpackage, **136**, **144**
ppwhos, **22**, **136**, **145**
profile, **103**
propagating the distributed attribute, **41**
propagation of distrbution
    exceptions for functions with multiple arguments, **47**
propagation of distribution, **44**
    ddense, **44**
    ddensend, **44**
    dsparse, **44**
    examples for functions of multiple arguments, **46**
    examples for functions of one argument, **45**
    functions of multiple arguments, **45**
    functions of one argument, **44**
    rules for, **46**
    summary, **49**
propgation of distribution
    functions of one argument, exceptions, **45**
ps, UNIX command for monitoring the server, **133**
Python, **1**

R

rand, **29**
randn, **29**
real number data, **27**
requirements of functions passed to ppeval, **64**
reshape, **30**, **102**
restructuring serial MATLAB code, **78**
reusing existing scripts, **23**
reusing scripts, **42**
round-off errors, **90**
row distribution of matrices, **33**
rules for propagation of distribution, **46**

S

saving and loading data on the parallel server, **51**
serial MATLAB code, restructuring, **78**
server
    configuring data I/O path, **16**
server/client messages, excessive, **93**
server vs. client variables, ppeval, **66**
shallow copy vs. deep copy, incompatibility of Star-P® and MATLAB, **32**
singleton dimensions, **44**
solving large problems - memory issues, **84**
solving large sparse matrix and combinatorial problems with Star-P, **217**
sort, **102**
sparse matrices
    data analysis and comparison with pivot tables, **220**
    graphs, and, **217**
    HDF5 file, representation of data in, **56**
    laplacian matrices, visualizing graphs, and, **223**
    on path counting, **224**
    representing graphs and general data analysis, **219**
sparse matrices, Star-P® representation of, **35**
special variables, p and np, **24**
specifying a range of nodes in a cluster, **17**
specifying a set of nodes in a cluster, **17**
specifying nodes in a cluster, **17**
speye, **29**
split, **2**, **137**, **152**
splitting on a scalar, workarounds for, **75**
sprand, **29**, **37**
sprandn, **29**
Star-P®
    sparse matrices, **35**
    support, **3**
Star-P® functions, **135**
    basic server function summary, **135**
    bcast, **137**, **151**
    data movement functions, **147**
    fseek, **135**, **138**
    np, **25**, **135**, **139**
    p, **24**, **139**

perfomance functions, **155**
pp, **26**, **135**, **139**
ppback, **28**, **136**, **147**
  warning messages, **51**
ppchangedist, **137**, **148**
  warning messages, **51**
ppeval, **2**, **59**, **137**, **152**
  broadcasting input arguments, **65**
  calling non-MATLAB functions within, **73**
  client vs. server variables, **66**
  distribution of input variables, **67**
  input agruments
    default behavior of, **64**
  input arguments, **64**
    broadcasting, **65**
    splitting, **64**
  known differences between MATLAB and Octave functions, **153**
  output arguments, **67**
  per process execution, **72**
  requirements of functions passed to, **64**
  splitting input arguments, **64**
  syntax and behavior, **63**
ppeval_tic, **97**
ppeval_toc, **97**
ppevalc, **59**
ppevalcloadmodule, **138**, **154**
ppevalcunloadmodule, **138**, **154**
ppevalsplit, **69**, **137**, **154**
ppfopen, **52**, **136**, **145**
ppfront, **24**, **136**, **148**
  warning messages, **51**
ppgetlog, **136**, **141**
ppgetlogpath, **136**, **142**
ppgetoption, **136**, **141**
pph5read, **54**, **137**, **150**
pph5whos, **55**, **136**, **146**
pph5write, **54**, **137**, **149**
ppinvoke, **136**, **143**
ppload, **28**, **51**, **137**, **150**
pploadpackage, **143**
ppper, **155**
ppperf, **103**, **106**, **131**, **138**
  displaying performance statistics, **106**
  gathering performance statistics, **106**
  graphical mode, **113**
  graphical mode, lessons learned, **117**, **131**
  interpretation of output, **108**
  lessons learned, **113**
  output preamble, **108**
  performance process measurement, **110**
  performance time measurment, **109**
  using to eliminate performance bottlenecks, **117**
 ppperf, using, **103**
 ppperf clear, **106**, **132**, **155**
 ppperf graph off, **132**, **156**
 ppperf graph on, **123**, **156**
 ppperf off, **106**, **132**, **155**
 ppperf report, **106**, **107**, **132**, **156**
 ppperf report detail, **107**, **132**, **156**
 ppperf resume, **155**
 ppprofile, **94**, **103**, **105**, **138**, **156**
 ppquit, **136**, **145**
 ppsave, **51**, **137**, **151**
 ppsetoption, **136**, **141**
  configuring for high performance, **90**
  warning messages, **51**
 ppstartup, **14**
 pptic, **91**, **103**, **138**, **157**
 pptoc, **91**, **103**, **138**, **157**
 ppunloadpackage, **136**, **144**
 ppwhos, **22**, **136**, **145**
 split, **2**, **137**, **152**
 task parallel functions, **151**
Star-P® naming conventions
  d, distributed, **22**, **60**
  pp, **22**, **60**
starp command, **10**, **15**
  configuring data I/O directory, **16**
  data I/O directory, **16**
starting Star-P® with MATLAB, **7**
  on a Linux client system, **7**
  on a Windows client system, **10**
startup.m, **14**
start-up configuration, user specific, **14**
string arrays, workarounds for, **75**
structs, using in task parallel, **85**
subsref, **48**
support, Star-P®, **3**
supported data types, **27**
supported MATLAB® functions, **161**
support website, **161**
svd, **41**

svds, **41**

T

task parallel, global variables, **75**
task parallel codes, tips for, **85**
task parallel engine
  choosing Octave, **71**
  using C++ for compiled codes, **72**
task parallel engine, choosing MATLAB, **91**
task parallelism with Star-P® and MATLAB, **2**, **59**
task parallelism workarounds and additional information, **75**
task parallel workarounds and additional information
  splitting on a scalar, **75**
  string arrays, **75**
tic/toc, **103**
tips and tools for high performance Star-P® code, **77**
tips for data parallel codes, **79**
  vectorization, **79**
tips for task parallel codes, **85**
  use of structs and cell arrays, **85**
  using external libraries, **89**
  vectorize for loops inside of ppeval calls, **86**
top, UNIX command for monitoring the server, **133**
transforming a for loop into a ppeval call, **61**
transpose, **102**
types of distributions, **32**

U

UNIX commands to monitor the server, **132**
user specific Star-P® start-up configuration, **14**

V

vectorization, **79**
vectorization, MathWorks online tutorial, **79**
vectorize for loops inside of ppeval calls, **86**
vertcat, **29**
Very High Level Languages (VHLL), **1**

W

which, **39**
whos, **22**

Z

zeros, **29**, **37**