

Star-P® User Guide



COPYRIGHT

© Copyright 2004-2006, Interactive Supercomputing, Inc. All rights reserved. Portions © Copyright 2003-2004 Massachusetts Institute of Technology. All rights reserved.

Trademark Usage Notice

MATLAB® is a registered trademark of The MathWorks, Inc. STAR-P™ and the "star" logo are 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.

Record of Revision

Version	Description
010	June 2006 (R2.3.0) Original publication

Contents

1 Star-P User Guide Overview	1
Introduction to Star-P	2
Star-P and MATLAB	3
Star-P Features	6
Interactive Computing Management	6
Library Management	8
Data Management	8
System Requirements	9
Hardware and Software	9
Acknowledgments	10
2 Quick Start Guide to Star-P	13
Starting Star-P on a Linux Client System	13
Starting Star-P on a Windows Client System	14
Star-P Sample Session	15
User-Specific Star-P Configuration	17
3 Using Star-P	19
Star-P Naming Conventions	19
“for” loops with a Distributed Index Variable	20
Examining Star-P Data	20
Special Variables: p and np	21
Assignments to p	24
Complex Data	24
Creating Distributed Arrays	25
The ppload and ppsave Star-P Commands	31
Writing variables to a HDF5 file	32
Reading variables from a HDF5 file	33
Querying variables stored inside a HDF5 file	33
Multidimensional arrays	34
Complex data	34
Sparse matrices	34

Explicit Data Movement with ppback and ppfront	36
Combining Data Distribution Mechanisms	38
Mixing Local and Distributed Data	39
Distributed Classes used by Star-P	40
Propagating the Distributed Attribute	43
Reusing Existing Scripts	47
Examining/Changing Distributed Matrices	47
The ppeval Function: The Mechanism for Task Parallelism	48
Input Arguments	49
Output Arguments	49
It's Still MATLAB: Using MATLAB Features on Parallel Programs	54
Configuring Star-P for High Performance	57
 4 Structuring Your Code for High Performance and Parallelism	55
Performance and Productivity	55
Parallel Computing 101	56
Vectorization	58
Star-P Solves the Breakdown of Serial Vectorization	61
Solving Large Problems: Memory Issues	63
 5 Data Organization in Star-P	65
Communication between the Star-P Client and Server	65
Implicit Communication	66
Client/Server Performance Monitoring	69
Communication Among the Processors in the Parallel Server	75
Types of Distributions	77
Two-Dimensional Block Cyclic Distribution	77
Row distribution	78
Column distribution	78
Block-cyclic distribution	79
How Star-P Represents Sparse Matrices	81
Distributed Partitioned Objects (dpart)	83
Calculus of Distribution Rule:	85

6 Application Examples	89
Application Example: Image Processing Algorithm	89
How the Analysis Is Done	89
Application Examples	89
patmatch_color_noStarP.m File	93
patmatch_calc.m	94
patmatch_color_StarP.m File	95
About ppeval	96
patmatch_color_ppeval.m	97
7 Solving Large Sparse Matrix and Combinatorial Problems with Star-P	99
Graphs and Sparse Matrices	99
Graphs: It's all in the connections	99
On Path Counting	106
8 Advanced Topics	109
Useful Commands to Monitor the Server	109
Architecture	110
Running Star-P in Batch	112
Performance Infrastructure	113
Star-P Software Development Kit	113
Introduction	113
Prerequisites	114
Programming Concepts	114
Invocation	114
Numerical Accuracy	115
Computation of Eigenvectors for Non-Hermitian Matrices	115
Glossary	117
administration server	117
backend	117
client	117
data parallel	117
frontend	117
global array syntax	117

high performance computer (HPC)	117
parallelism through polymorphism	117
pool	117
propagation of parallelism	117
server	117
Star-P processor pool	117
task parallelism	118
transparent parallelism	118
workgroup, Star-P workgroup	118
Index	119

1 Star-P User Guide Overview

Star-P is an interactive supercomputing software platform that enables use of high performance computers (HPCs) by eliminating the re-programming often associated with porting desktop application programs or by eliminating the need to program in a low-level parallel interface when rapid application development is desired. It is intended for scientists, engineers and analysts who want to solve large and complex MATLAB®¹ problems that can no longer be done productively on the desktop computer. Star-P fundamentally transforms the workflow, substantially shortening the time to production. Star-P delivers the best of both worlds: it couples the interactive and familiar use of desktop application with the computing power of HPCs.

This chapter provides an overview of Star-P and includes sections on the following topics:

- ["Introduction to Star-P"](#) provides an overview of the goals and capabilities of the Star-P software platform.
- ["Star-P and MATLAB"](#) describes how Star-P parallelizes MATLAB programs with minimal modification.
- ["Star-P Features"](#) summarizes the many features that Star-P provides in the areas of interactive computing management, library management, and data management.
- ["System Requirements"](#) is a list of the hardware and software required to run Star-P.
- ["Documentation and Support"](#) lists the documentation that is available for the Star-P system.
- ["About This Guide"](#) summarizes the topics covered in this document.

NOTE: The latest version of this User Guide will always be available at www.interactivesupercomputing.com/support

¹ MATLAB® is a registered trademark of The MathWorks, Inc. STAR-P™ and the "star" logo are 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.

Introduction to Star-P

Star-P was developed with the following major goals:

- usability is the first priority
- large data sets should not be moved around without the user's explicit request
- existing MATLAB code should run as much as practical without modification, and any necessary modifications should use constructs as familiar as possible for a MATLAB user.

Star-P enables a client-server model in which a desktop client application such as MATLAB, a popular desktop scientific computing tool, is transparently linked with a powerful remote HPC *server*, also referred to as the parallel or back-end system, through a standard computer network. The *client*, or front-end, provides the familiar interactive user interface while the HPC server manipulates, computes, saves and visualizes the applications data.

Star-P eliminates the need for users to program the linkage to a remote HPC system and the need to code the details of parallel programs. Instead, Star-P delivers interactive performance by automatically:

- sending computations to the HPC
- providing access to built-in world-class parallel computing libraries
- enabling extension with community or user-specific parallel libraries
- managing inter-processor communication
- managing the flow and memory storage of large data sets

The interactive performance results in a faster response time that enables the user to work with large data sets from the beginning (rather than restricting the size to what fits on the desktop) and interactively refine the model.

Star-P allows the user to use the familiar MATLAB interface on a parallel system to greatly speed up desktop computations and the resulting solution time, allowing larger problem sets to be handled. Standard MATLAB commands and functions are available and perform in a parallel manner transparently to the user. The large parallel objects are stored distributed across the parallel system, greatly reducing communication costs and delays when performing operations.

While designed for the non-expert desktop user, Star-P also features the controls often demanded by users familiar with parallel computing. Users wanting greater control over parallel execution can use a set of functions for that purpose.

Star-P and MATLAB

With Star-P, existing MATLAB scripts can be re-used to run larger problems in parallel with minimal modification. No programming needs to be done beyond standard MATLAB programming, with very few minor modifications. No new function calls need to be learned or added in order to take advantage of the power of parallel computing through data parallelism. Use of task parallelism requires one new construct.

Star-P overloads ordinary MATLAB commands with the ***p** construct. This simply multiplies (*) array dimension(s) by a symbolic variable denoting that the dimension is to be distributed. A class of MATLAB programs becomes parallel with the simple insertion of this construct. This provides a simple method to parallelize MATLAB programs. 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. 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.

The user needs 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.

With all this discussion about distributed matrices and operations, it's easy to forget that you're still using MATLAB as your desktop tool. This means that you can run an existing MATLAB program in Star-P with 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 lots of features of the MATLAB environment that are still relevant for 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 doesn't 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. The ease of use is demonstrated with the example of a randomly generated matrix, which is inverted to create a second matrix, multiplied by its inverse to result with a matrix of ones.

The following example illustrates MATLAB executed on the client or front-end;

```
>> tic;
>> x=rand(2000,2000);
>> xinv=inv(x);
>> xo=x*xinv;
>> max(max(xo))
>> toc
ans =
    1.0000
Elapsed time is 60.782155 seconds.
>> whos
Name      Size      Bytes      Class
ans       1x1         8         ddense array
x         2000x2000  32000000   ddense array
xinv      2000x2000  32000000   ddense array
xo        2000x2000  32000000   ddense array
```

Grand total is 12000001 elements using 96000008 bytes

The following example illustrates execution of the same script, but modified to be run by Star-P in parallel on the HPC system:

```
>> tic;
>> x=rand(2000*p,2000*p);
>> xinv=inv(x);
>> xo=x*xinv;
>> max(max(xo))
>> toc
```

```

ans =
    1.0000
Elapsed time is 11.824418 seconds.
>> ppwhos
Your variables are;

```

Name	Size	Bytes	Class
ans	1x1	8	double array
x	2000px2000p	32000000	ddense array
xinv	2000px2000p	32000000	ddense array
xo	2000px2000p	32000000	ddense array

```

Grand total is 12000001 elements using 96000008 bytes
MATLAB has a total of 1 elements using 8 bytes
Star-P server has a total of 12000000 elements using
96000000 bytes

```

As seen above, the example problem is captured in a simple sequence of MATLAB operators and run either on the MATLAB client front-end, without Star-P, or on the back-end parallel processor with Star-P. The operation is timed with **tic/toc** to demonstrate the performance increase using a 4-processor Altix configuration. The front-end-executed code operates solely on double variables, which reside in the memory on the MATLAB client machine.

To execute the same MATLAB code on the parallel processor, the ***p** construct is used to tell MATLAB to create the 2000x2000 random elements of the new matrix in the HPC's memory. As indicated in the **ppwhos** trace, the Star-P analog to MATLAB **whos**, the **x** variable becomes **ddense**; a dense matrix residing in the distributed memory. The subsequent **inv** and multiplication operations inherit distribution from their variables and respective results are **ddense**. When the result of an operation results in a scalar variable, this variable is placed on the front-end as illustrated by **ans**; the result of **max(max(xo))**. It does not make sense to distribute a scalar value. This yields straightforward parallelization.

Most of this manual is focused on writing a single application, and the mechanics of how to do that. Of course, just as you can run multiple instances of MATLAB at the same time on your desktop, doing the same or different jobs, you can run multiple *sessions* of Star-P at the same time. Obviously two sessions cannot use the same processors at the same instant, so there will be some scheduling by the server, but especially if they use distinct sets of processors, multiple sessions can be an aid to user productivity.

Star-P Features

Proprietary ISC technology provides many of the features of the Star-P Server. These are in the following areas:

- Interactive computing management

Star-P handles session scheduling, allocation of processors and memory to workgroups, and user priorities. It can be used within the framework of an existing batch system. († Batch system linkage functionality is deferred.)

- Library management

Some third-party libraries have already been incorporated into Star-P, examples being ScaLAPACK, FFTW, and SuperLU. Those libraries have been exercised by the Star-P testing process, just as Star-P code has, and any errors found in the libraries themselves have been corrected or worked around, to the extent possible.

You can plug other libraries or applications into Star-P. Some third-party packages will already be prepared to plug into Star-P, so you'll just need to acquire them, including any licenses if they are commercial products. For your own libraries, you can follow the information in the Software Developer Kit.

- Data management

Star-P allows for the interoperability of all packages integrated into the system. It also handles all aspects of data management in conjunction with the underlying parallel system, including persistence and sharing of data among users, without the data having to be sent back to the front-end (desktop) machine. The Star-P server works with the underlying OS and file system to ensure data security, privacy, and integrity.

The features in each of these areas are described in the following sections.

Interactive Computing Management

- Flexible scheduling

Users can customize the system to match specific needs. Users can assign priorities, number of CPUs allocated for interactive use, memory allocated, etc. Star-P handles all session scheduling, user priorities, and processor resource allocation.

- Robust commercial grade HPC server

The Star-P server has been fielded and tested under production conditions. It can recover from most errors,

including MPI workgroup errors, and prevents individual user failures from affecting other users.

- Comprehensive system administration

A full administration tool suite is provided for the system administrator to check usage, add and configure users, install Star-P options and licenses, and perform job control

- Standard port and protocols

Star-P uses standard port and protocols to communicate with the HPC server. When tunneling through SSH is selected, port 22 is the default, so that most outbound communication can work through standard firewalls, without requiring extensive configuration.

- Multiple clients

With the purchase of appropriate licenses, multiple users can access Star-P simultaneously. The single installation of Star-P can support multiple users, some of whom may be performing simultaneous calculations; others may be connected by temporarily idle (e.g., performing local tasks or typing commands).

- Multiple Star-P servers

Multiple client sessions on a single user's PC can connect to the same Star-P server. This allows the user to have multiple Star-P sessions open concurrently, for working on different problems, or dividing the work and keeping variables and projects separated. This can be useful for comparative debugging. Connecting to multiple Star-P servers allows multiple computations to be ongoing in parallel. († Implementation of this feature is deferred.)

- Pooled processors

Processors are divided into workgroups for the performance of different tasks. The system administrator configures workgroups based on user requirements and the system automatically assigns the pools of processors

- Seamless integration with MATLAB

Star-P is not a MATLAB compiler or simulator. Your code is interpreted by the MATLAB program, with some of the actual calculations performed on the (remote) HPC system and some on the client MATLAB system. This means that new releases of MATLAB will be supported with minor changes to the Star-P software and user model code.

- Advanced cluster management

In concert with the underlying OS and workload manager, the Star-P server takes care of all session scheduling, monitoring, allocation of processors, and allocation of memory. It handles administration of users. It works within the framework of existing batch systems. († Batch system linkage functionality is deferred.)

- Custom levels of security

At the highest level of security, both client and server are run on secure platforms over a secure network using only secure shell (**ssh**) encryption technology. Customers with less demanding security needs can run servers and clients directly across corporate intranets and even the Internet.

Library Management

- Integration of new library and application packages

New distributed computing packages can be integrated into Star-P through the Software Development Kit, or called directly on the server using the **ppinvoke** function. These packages can be publicly or commercially available, or customer proprietary.

Data Management

- Efficient handling of distributed data

Users have the ability to save and restore distributed variables and work environments. Variables and work environment can be stored directly from the Star-P system; they do not have to be loaded from the front end client again. This allows for large amounts of data to be loaded/created/stored easily.

- Distributed sparse matrix calculations

A full range of calculations can be performed on distributed sparse matrices.

- Distributed dense matrix calculations

A full range of calculations can be performed on distributed dense matrices.

- Data decomposition

Star-P supports the following object distribution methods: row-distributed, column-distributed, and block- cyclic distributed for dense matrices, and compact sparse row distributions for sparse matrices.

- 64 bit support

The server code (when running on Itanium processors) is a full 64-bit application, allowing arrays and matrices with total of 10^{19} bytes of memory, plus array indices up to 10^{19} elements, subject to amount of total installed memory on the HPC system.

System Requirements

Hardware and Software

For information on the hardware and software requirements for the Star-P HPC server, the Star-P Administration Server, and the Star-P Client, see the *Star-P Installation and Administration Guide*.

Supported Clusters

Star-P is supported on SGI Altix in single system image or cluster configurations.

Documentation and Support

Star-P documentation can be found on the Support page of the Interactive Supercomputing, Inc. website, www.interactivesupercomputing.com:

- [FAQ - Frequently Asked Questions](#)
- [Knowledge Base](#)
- [Known Bugs and Workarounds](#)
- [Most up-to-date documentation regarding the Star-P product](#)

About This Guide

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

- ["Quick Start Guide to Star-P"](#), takes you through a sample session to illustrate how to start up Star-P from a web or command line interface and to execute a simple program that illustrates the use of Star-P's ability to parallelize MATLAB code.
- ["Using Star-P"](#), provides more extensive instructions on using Star-P to create distributed arrays and on moving data between the front-end and the HPC server.
- ["Advanced Topics"](#), includes information on an assortment of administrative and coding issues that may arise when working in the Star-P environment.
- ["Structuring Your Code for High Performance and Parallelism"](#), is an overview of the issues you face when writing code that will be parallelized. It includes an overview of parallel computing and covers the topics of vectorization and profiling code to take advantage of parallelization.
- ["Data Organization in Star-P"](#), describes how communication takes place between the frontend system and the HPC server and how you might evaluate this and

account for it in writing your code. The chapter also describes how Star-P distributes matrices among the parallel servers.

- "[Application Examples](#)", includes some example programs that illustrate how MATLAB programs can take advantage of Star-P's capabilities.
- "[Solving Large Sparse Matrix and Combinatorial Problems with Star-P](#)", introduces the more advanced mathematical topics of sparse matrices and combinatorial problems and how you can take advantage of parallelization when solving these problems.

Acknowledgments

The developers of Star-P would like to gratefully acknowledge the following pioneers in high performance computing software. These state-of-the-art libraries are far more complicated than their serial predecessors. In many cases, these libraries represent ongoing research projects. It is our hope that by providing an easy interface, Star-P can accelerate these research projects by growing the number of users thereby fleshing out the numerical and performance issues that affect us all.

- FFTW

The fastest Fourier transform in the west was developed by Matteo Frigo and Steven Johnson when students at MIT. Alan Edelman is pleased that both students were in his graduate parallel computing class at MIT but otherwise acknowledges that these superstars developed FFTW completely on their own. We believe that the FFTW program is one of the best scientific computing library projects of modern times.

For more information, go to <http://www.fftw.org/>

- Octave

Our parallel evaluate function uses the "Octave" software whose creation was led by John Eaton. As Eaton has told us, Octave was created to allow students to learn chemical engineering. Octave is highly compatible with MATLAB, but runs on platforms where MATLAB is not supported. This makes Octave of high value for server computations.

For more information, go to <http://www.octave.org/>

- PARPACK

The "Parallel Arnoldi Package" is underneath our sparse eigensolvers and singular value decompositions. The

software uses the Arnoldi algorithm through a matrix-vector multiply interface to "project" large sparse problems onto smaller dense problems.

[1] K. J. Maschhoff and D. C. Sorensen.

PARPACK: An efficient portable large scale eigenvalue package for distributed memory parallel architectures.

In Jerzy Wasniewski, Jack Dongarra, Kaj Madsen, and Dorte Olesen, editors, *Applied Parallel Computing in Industrial Problems and Optimization*, volume 1184 of *Lecture Notes in Computer Science*, Berlin, 1996. Springer-Verlag

for more information, go to http://www.caam.rice.edu/~kristyn/parpack_home.html

- ScaLAPACK

The ScaLAPACK project led by Jack Dongarra and Jim Demmel is one of the largest software development projects for high performance computers. This "scalable linear algebra package" covers the full range of matrix computations and is beginning its second round of development.

[1] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

For more information, go to <http://www.netlib.org/scalapack/>

- SPRNG

Nearly everyone's first parallel random number generator is invalidated by the use of the same seed in every processor. In order to have high quality parallel random number generation Star-P uses the SPRNG software from Florida State University.

For more information, go to <http://sprng.cs.fsu.edu/>

[1] Mascagni, Ceperley, and Srinivasan, *SPRNG: A Scalable Library for Pseudorandom Number Generation*, ACM Transactions on Mathematical Software, 2000.

- SuperLU

SuperLU underlies our linear solver ("") for sparse matrices. This is the direct solver that collects nodes into supernodes, produced by Sherri Li, Jim Demmel, and John Gilbert .

For more information, go to <http://crd.lbl.gov/~xiaoye/SuperLU/>

In addition, we wish to thank the scientific computing community for a grand history that has raised our expectations of numerical accuracy, performance, and ease of use. We hope that high performance computing can exceed these expectations. We single out some special software of note:

- MATLAB

We thank Cleve Moler, Jack Little, and the entire staff of Mathworks for creating MATLAB. The inviting interface of MATLAB has welcomed so many new users into the realm of scientific and engineering computations. We also thank Cleve Moler in particular for providing encouragement to the academic project at MIT through what was (no doubt informally) referred to as the "Friends of Cleve" software licensing program.

- The Connection Machine Scientific Software Library

A few of us still remember what was probably the best parallel scientific software library of all time: CMSSL developed in the early 1990s. The project, led by Lennart Johnsson, was pioneering for its time and impressive in its depth. Some of the current libraries are still catching up to the high quality of CMSSL. Among the important ideas not lost on us that were realized at the time were that library software has to interoperate with user written programs and that block cyclic linear algebra algorithms can be implemented in time rather than space.

- Parallel MATLABs

In 2003 we counted 27 projects somewhat worthy of the name "Parallel MATLAB." We believe that the ultimate parallel MATLAB is a serial MATLAB working faster and/or on larger problems. All the parallel MATLABs should strive for this elusive goal.

Some important projects of note are the MultiMATLAB project in the late 1990s at Cornell University lead by Anne Trefethen et al. This is a precursor and in many ways similar to the MatlabMPI project lead by Jeremy Kepner at MIT's Lincoln Laboratory. More recently pMATLAB has been adding global array syntax to an environment that has MATLAB on every processors, and the Mathworks Distributed Processing Toolbox allows users to message pass among MATLABs on multiple processors.

Quick Start Guide to Star-P

This chapter is intended for users who have a working Star-P installation on a client system as well as a server high performance computer.

Users running Linux should read "[Starting Star-P on a Linux Client System](#)". Users running Windows XP should read "[Starting Star-P on a Windows Client System](#)". Other operating systems are not supported.

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 `sauron`) and connect to the Star-P server configured by the administrator, which happens to be a system named `hope2.americas.sgi.com`.

```
sauron% starp
spr@hope2.americas.sgi.com's password: *****
```

```
< M A T L A B >
Copyright 1984-2005 The MathWorks, Inc.
Version 7.0.4.352 (R14) Service Pack 2
January 29, 2005
```

```
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 Client.
(C) MIT 2002-2005.
(C) Interactive Supercomputing, LLC 2004,2005.
All Rights Reserved.
Evaluation copy. Not to be redistributed.
By using this software you agree to the terms and conditions
described
in AGREEMENT. Type help AGREEMENT
```

>>

As you can see, the HPC server will typically require a password for user authentication, which you'll need to supply 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 used to from prior MATLAB experience, and can start to use the Star-P extensions described in "[Using Star-P](#)".

The preceding instructions assume you are using an Administration Server that has been properly configured. To configure your Administration Server, see "Using the Administration Server" in the *Star-P Administration Guide*.

Star-P can be used without first installing or configuring an Administration Server. However, running in this mode requires additional information to be specified on the `starp` command line. Thus, if you will be running in this mode, you must either invoke `starp` from a DOS command prompt or from a batch script.

A full description of the **starp** command and its options is provided in the **Star-P Command Reference Guide**, or type `starp --help` at the command line.

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 Administration Guide*.

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

Start -> All Programs -> Star-P Client Software -> starp

You will be prompted for a username and password for access to the Star-P Administration Server. This username and password will have been assigned by your administrator during the configuration of the Star-P Administration Server, and can be changed via the Administration Server's web

interface. Given this username and password, the Administration Server will automatically be queried for a Star-P session. Star-P sessions define which HPC server should be used, which UNIX username should be used on that HPC server, and how many CPUs should be accessed.

If more than one Star-P session has been configured, a menu will appear, prompting you to choose the desired Star-P session. Once the session has been selected, the Star-P Client will attempt to access the Star-P HPC Server program on the HPC machine using `ssh`. If `ssh` has not been configured to work without a password, you will be prompted for your operating system account password on the HPC machine.

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

The preceding instructions assume you are using an Administration Server that has been properly configured. To configure your Administration Server, see “Using the Administration Server” in the *Star-P Installation and Administration Guide*.

Star-P can be used without first installing or configuring an Administration Server. However, running in this mode requires additional information to be specified on the `starp` command line. Thus, if you will be running in this mode, you must either invoke `starp` from a DOS command prompt or from a batch script.

A full description of the **starp** command and its options is provided in the **Star-P Command Reference Guide**, or type `starp --help` at the command line.

Star-P Sample Session

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

Check to see whether the server is alive, and the number of processes running

```
>> np
ans =
     4
```

Create a 100x100 random dense matrix:

```
>> A = randn(100,100*p);
A =
      ddense object: 100-by-100p
```

Retrieve the first element:

```
>> A(1,1)
ans =
    0.3855
```

Create a 100x100 random dense matrix:

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

Solve the system $AX=B$:

```
>> X = A\B;
```

Check the answer:

```
>> norm(A*X-B)
ans =
    2.7259e-13
```

Get information about variables:

```
>> ppwhos
Your variables are:
```

Name	Size	Bytes	Class
A	100x100p	80000	ddense array
B	100px100	80000	ddense array
X	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
```

To end Star-P execution, you can use either the **quit** or the **exit** command:

```
>> quit
sauron 74=>
```

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

User-Specific Star-P 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 MATLAB/Star-P client 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 which sparse linear solver to use (see **ppsetoption** documentation in the **Star-P Command Reference Guide**) or for loading your own packages (see the **Star-P Software Development Kit Tutorial**).

3 Using Star-P

The `matlab2pp` command is replaced by `ppback` and `pp2matlab` is replaced by `ppfront`.

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 implicitly run 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 programming.

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. For further information, see the examples in "[Propagating the Distributed Attribute](#)".

It is possible to use Star-P to apply MATLAB functions to the chunks of a distributed matrix or to individual processors. This is discussed in "[Data Organization in Star-P](#)".

NOTE: In the examples throughout this chapter, the output of the `whos` and `ppwhos` command may be edited for brevity, such that the total number of bytes/elements represented by all the arrays may not equal the summary at the end of the output.

Star-P Naming Conventions

Star-P commands and data types generally follow 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.

- Star-P data types begin with the letter *d*, to indicate “distributed”. For example, the Star-P `dsparse` class implements distributed sparse matrices.

These commands and classes are discussed throughout this chapter.

“for” loops with a Distributed Index Variable

When a distributed variable is used in the range of a `for` loop, Star-P silently converts the range into a front-end double.

For example, the loop

```
for i = 1:1000*p
```

will silently be converted into

```
for i = 1:1000
```

The user is cautioned that if a very large distributed value is placed in the index range of the `for` loop, it will be converted to a double and the loop will execute on the client. This may result in a slowly performing loop.

Examining Star-P Data

This section describes how you can look at your variables and see their sizes and 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 is unaware of the true sizes of the distributed arrays. Star-P supports a similar command called `ppwhos`. Its output looks like this:

```
>> clear
>> n = 1000;
>> a = ones(n*p);
>> b = ones(n*p,n);
>> ppwhos;
```

Your variables are:

Name	Size	Bytes	Class
a	1000px1000p	8000000	ddense array
b	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;
>> np
ans =
    256
>> aa = rand(sqrt(n)*sqrt(np)*p, sqrt(n)*sqrt(np)*p);
>> ppwhos
Your variables are:
      Name      Size      Bytes      Class
      aa      357770px357770p  1.023995e+12 ddense array
      n        1x1           8          double array
```

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:

```
>> whos
      Name      Size      Bytes      Class
      a      1000x1000      1160 ddense object
      b      1000x1000      1160 ddense object
      n        1x1           8 double array
```

```
Grand total is 71 elements using 2328 bytes
```

The `ppwhos` command gives the correct information.

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.

```
>> z = 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.

```
sauron 69=> starp
```

```
[...Star-P initial display appears...]
```

After Star-P initializes, 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 =
    2
```

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

```
>> n = 100;
>> a = ones(n*p);
>> b = ones(n*p,n);
>> c = b*b;
>> p = 3.14;
>> z = p*p;
>> z, p,
z =
```

```

          9.8596
p =
          3.1400

>> ppwhos
Your variables are:
      Name      Size      Bytes      Class
      a         100px100p   80000    ddense array
      b         100px100    80000    ddense array
      c         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 240282 bytes
MATLAB has a total of 4 elements using 282 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
      a         100px100p   80000    ddense array
      ans        1x1        258    dlayout array
      b         100px100    80000    ddense array
      c         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.

The variable **pp** is a synonym for **p**.

Assignments to 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.

Complex Data

Complex numbers in Star-P are supported as in MATLAB; i.e., they 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.

```
>> n = 1000

>> a = rand(n*p,n)
a =
    ddense object: 1000p-by-1000
>> b = randn(n*p,n)
b =
    ddense object: 1000p-by-1000
>> c = complex(a,b)
c =
    ddense object: 1000p-by-1000
>> cc = a+b*i
cc =
    ddense object: 1000p-by-1000
>> d = real(c)
d =
    ddense object: 1000p-by-1000
>> e = imag(c)
e =
    ddense object: 1000p-by-1000
>> f = conj(c)
f =
    ddense object: 1000p-by-1000
>> ppwhos
Your variables are:
      Name      Size      Bytes      Class
      a      1000px1000    8000000    ddense array
```

```

b          1000px1000      8000000  ddense array
c          1000px1000      16000000 ddense array (complex)
cc         1000px1000      16000000 ddense array (complex)
d          1000px1000      8000000  ddense array
e          1000px1000      8000000  ddense array
f          1000px1000      16000000 ddense array (complex)
n          1x1             8         double array

```

```
>> isreal(b)
```

```
ans =
```

```
1
```

```
>> isreal(c)
```

```
ans =
```

```
0
```

```
>> isreal(cc)
```

```
ans =
```

```
0
```

```
>> isreal(d)
```

```
ans =
```

```
1
```

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

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

```
a =
```

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

```
>> b = fft(a)
```

```
b =
```

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

```
>> ppwhos
```

```
Your variables are:
```

Name	Size	Bytes	Class
a	1000x1000p	8000000	ddense array
b	1000x1000p	16000000	ddense array (complex)

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**, or **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()` (this command is equivalent to the `matlab2pp` command) 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 details on `p`, see ["Complex Data"](#).

Also, see ["Special Variables: p and np"](#).

```
>> p
ans =
    1p
>> whos
```

	Name	Size	Bytes	Class
	ans	1x1	258	dlayout object

Grand total is 4 elements using 258 bytes

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, and indeed causes MATLAB to abort.

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 `A` as a distributed dense matrix with normally distributed random numbers:

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

The first and second examples create matrices that are distributed in the first and second dimensions, respectively. The last two examples do the same thing; namely, create a matrix that is distributed in both dimensions. For more detail, see ["Types of Distributions"](#).

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

```
A = 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; catenation 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:

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

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

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

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

```
>> a = rand(100,100);
>> aa = reshape(a,100,100*p)
aa =
```

```

ddense object: 100-by-100p
>> bb = reshape(aa,100,100);
>> ppwhos
Your variables are:
      Name      Size      Bytes      Class
      a         100x100    80000    double array
      aa        100x100p   80000    ddense array
      bb        100x100p   80000    ddense array

Grand total is 30000 elements using 240000 bytes
MATLAB has a total of 10000 elements using 80000 bytes
Star-P server has a total of 20000 elements using 160000
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
      n         1x1        258      dlayout object
Grand total is 4 elements using 258 bytes
>> A = rand(n)
A =
      ddense object: 1000p-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:

```

>> a = rand(1000*p);
>> b = rand(1000*p);

```

Operations that result in distributed matrices

```
% indexing large (sub)sections of the elements of a
distributed object
% in most ways results in a distributed object
>> c = a(1:end,1:end);
>> d = a(18:23,47:813);
>> f = a(:); % linearize 2D array into 1D vector
% doing assignment via the linearization approach works
% naturally
>> b(:) = 0;
```

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

```
>> e = a(47,418); % scalar goes to front-end
>> nnz(a) % scalar answer 'ans' goes to
front-end
ans =
    1000000
>> nnz(b) % scalar answer 'ans' goes to
front-end
ans =
    0
```

```
>> ppwhos
Your variables are:
      Name      Size      Bytes      Class
      a      1000px1000p      8000000      ddense array
      ans      1x1          8      double array
      b      1000px1000p      8000000      ddense array
      c      1000px1000p      8000000      ddense array
      d      6px767p      36816      ddense array
      e      1x1          8      double array
      f      1000000px1      8000000      ddense array
Grand total is 5006605 elements using 40053276 bytes
MATLAB has a total of 1000003 elements using 8000460
bytes
Star-P server has a total of 4006602 elements using
32052816 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 *b* and *d* 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 *d* 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 `a = b`, as soon as that assignment is complete, you can modify either `a` or `b` 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 `a` or `b`, the contents of both `a` and `b` get modified. Because of the semantics of the MATLAB language, this is only relevant for assignments of portions of `a` or `b`; i.e., `a(18, :) = ones(1, 100*p)` or `a(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 `a = b(:, :)`. Another example that works for all non-logical arrays is `a = +b`.

Loading Data to 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 A and B 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 A B
>> ppwhos
Your variables are:
      Name      Size      Bytes      Class
      A      1000x1000p    8000000    ddense array
      B      1000x1000p    8000000    ddense array
```

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: `ppback` is equivalent to the `matlab2pp` command; `ppfront` is equivalent to the `pp2matlab` command.

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. (See "[The ppload and ppsave Star-P Commands](#)" for a full description of **ppload** and **ppsave**.)

The ppload and ppsave Star-P Commands

The distributed I/O commands **ppsave** and **ppload** 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 itself 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 frontend to the backend via a file, the MATLAB **save** command must use the `-v6` format, as in `save('gink', 'w', '-v6')` for saving variable `w` in file `gink`. 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 wind up 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.

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 a 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:

```
>> pph5write('/tmp/temp.h5', matrix_a, '/my_matrices/a', ...
            matrix_b, '/my_matrices/workspace/temp/
            matrix_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:

```
>> pph5write('/tmp/temp.h5', 'append', matrix_c, '/
            my_matrices/workspace2/temp/matrix_c');
```

Reading variables from a 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/temp/matrix_b');
```

Querying variables stored inside a HDF5 file

It is possible to obtain a list of variables stored in a 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')
```

Name	Size	Bytes	Class
/my_matrices/a	10x10x10	8000	double array
/my_matrices/workspace/temp/matrix_b	100x100(10(nnz))	800	double array (sparse)
/my_matrices/workspace2/temp/matrix_c	100x100	80000	double array

Representation of data in the HDF5 file

This section describes the internal representation of HDF5 files used by the functions described in 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.

Explicit Data Movement with **ppback** and **ppfront**

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

NOTE: The **ppback** command is equivalent to the **matlab2pp** command; the **ppfront** command is equivalent to the **pp2matlab** command.

```
>> ppwhos
```

```
Your variables are:
```

Name	Size	Bytes	Class
ans	1x10	20	char array
mA	1000x1000	8000000	double array
mB	1000x1000	8000000	double array
n	1x1	258	dlayout array

```
Grand total is 2000011 elements using 16000278 bytes
MATLAB has a total of 2000011 elements using 16000278
bytes
Star-P server has a total of 0 elements using 0 bytes
```

*ppback is equivalent
to the matlab2pp
command*

```
>> A = ppback(mA)
```

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

```
>> ppwhos
```

```
Your variables are:
```

Name	Size	Bytes	Class
A	1000px1000p	8000000	ddense array
ans	1x10	20	char array
mA	1000x1000	8000000	double array
mB	1000x1000	8000000	double array
n	1x1	258	dlayout

```
array
```

```
Grand total is 3000011 elements using 24000278 bytes
MATLAB has a total of 2000011 elements using 16000278
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 sometimes users 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.

```
>> a = rand(17000,17000*p)
a =
      ddense object: 17000-by-17000p
>> ppwhos
Your variables are:
      Name      Size      Bytes      Class
      a      17000x17000p  2.312000e+09  ddense array

Grand total is 289000000 elements using 2.312000e+09
bytes
MATLAB has a total of 0 elements using 0 bytes
Star-P server has a total of 289000000 elements using
2.312000e+09 bytes
>> b = ppfront(a);
Jul 29, 2005 3:46:54 PM com.isc.starp.ppclient.PPClient
call
SEVERE: Receieved error in PPClient.call
java.lang.OutOfMemoryError
??? Error using ==> ppclient
Client error: java.lang.OutOfMemoryError.

Error in ==> error:<starp_root>/matlab/ppfront.p,22,1
ppfront at 22
```

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.

NOTE: The `ppback` command is equivalent to the `matlab2pp` command; the `ppfront` command is equivalent to the `pp2matlab` command.

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 the **Star-P Command Reference Guide**.

Combining Data Distribution Mechanisms

The data distribution mechanisms can be combined in a program. For instance, the array `A` 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 A
>> [rows cols] = size(A)
rows =
      1000
cols =
      1000p
>> B = zeros(rows,cols);
>>
>> ppwhos
Your variables are:
      Name      Size      Bytes      Class
      A      1000x1000p    8000000    ddense array
      B      1000x1000p    8000000    ddense array
      cols      1x1        258        dlayout array
      n        1x1        358        dlayout array
      rows      1x1         8          double array
```

```
Grand total is 2000003 elements using 16000524 bytes
MATLAB has a total of 3 elements using 524 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 or scripts or toolboxes that are not aware of the distributed nature of their input, but will work anyway. For example, the function `gink` is defined as follows:

```
>> c = function gink(a)      % now executing in the
function gink
```

```
>> [rows cols] = size(a)
>> b = rand(rows,cols); % create 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 `gink` are defined for distributed objects as well as regular MATLAB objects:

```
>> a = rand(1000*p);
>> c = gink(a)
>>
>> ppwhos
Your variables are:
  Name      Size      Bytes      Class
  a         1000px1000p  8000000    ddense array
  b         1000px1000p  8000000    ddense array
  c         1000px1000p  8000000    ddense array
  cols      1x1              258        dlayout array
  rows      1x1              258        dlayout array
```

```
Grand total is 3000002 elements using 24000516 bytes
MATLAB has a total of 2 elements using 516 bytes
Star-P server has a total of 3000000 elements using
24000000 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);
>> bb = rand(100*p)
bb =
      ddense object: 100p-by-100p
>> pptic ; c = a + bb ; pptoc
```

```
Client/server communication info:
      Send msgs/bytes      Recv msgs/bytes      Time spent
      4e+00 / 2.080e+02B   4e+00 / 8.054e+04B   9.413e-01s
Server info:
      execution time on server: -2.271e-03s
      #ppchangedist calls: 0
```

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.

```
>> bb = rand(100*p)
bb =
      ddense object: 100p-by-100p
>> a = bb * pi
a =
      ddense object: 100p-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.

```
>> q = rand(10000*p,16);
>> r = rand(16,1);
>> s = q*r
s =
      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 **ppwhos** output, 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;
>> a = ones(n*p) ;
>> b = ones(n*p,n) ;
>> ppwhos;
Your variables are:
      Name      Size      Bytes      Class
      a      1000px1000p    8000000    ddense array
      b      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 `ddense`. For instance, if you wanted to know where the routine is that implements the `sum` operator, you would use the MATLAB `which` command, as in the following example:

```
>> which sum
/opt/matlab/toolbox/matlab/datafun/sum.m
>> which @double/sum
/opt/matlab/toolbox/matlab/datafun/@double/sum.bi %
double method
>> which @ddense/sum
<starp_root>/matlab/@ddense/sum.p % ddense method
>> which @dsparse/sum
<starp_root>/matlab/@dsparse/sum.p % dsparse method
>> which @ddensend
<starp_root>/matlab/@ddensend/sum.p
```

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 **ddense**, 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/dsparse** and will work in the obvious way, and for which no class-specific version of the routine is necessary.

```
>> which hilb
/opt/matlab/toolbox/matlab/elfmat/hilb.m
>> which @double/hilb
'@double/hilb' not found.
>> which @ddense/hilb
'@ddense/hilb' not found.
%ddensend method
```

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 the **Star-P Command Reference Guide**. 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.

```
>> clear
>> n = 1000;
>> a = rand(n*p)
a =
      ddense object: 1000p-by-1000p
>> [rows cols] = size(a)
rows =
      1000p
cols =
      1000p
>> ppload imagedata A
>> B = inv(A)
```

```

B =
      ddense object: 1000p-by-1000p
>> [Brows Bcols] = size(B)
Brows =
      1000p
Bcols =
      1000p
>> ppwhos
Your variables are:
      Name      Size      Bytes      Class
      A      1000x1000p    8000000    ddense array
      B      1000px1000p    8000000    ddense array
      Bcols    1x1        258        dlayout array
      Brows    1x1         8          dlayout array
      a      1000px1000p    8000000    ddense array
      ans     1x26         52         char array
      cols    1x1        258        dlayout array
      n       1x1         8          double array
      rows    1x1        258        dlayout array

```

```

Grand total is 3000031 elements using 24000842 bytes
MATLAB has a total of 31 elements using 842 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 `a` is created as a distributed object through the `*p` syntax, `b` will be created as distributed.

```
>> a=ones(100*p);
>> b = 2 * a;
>> ppwhos
Your variables are:
      Name      Size      Bytes      Class
      a         100px100p   80000      ddense array
      b         100px100p   80000      ddense array
```

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 X , and stores the result in a matrix e , which resides on the server.

```
>> x = rand(10000*p);
>> e = eig(x);
```

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 with 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:

```
[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:

```
>> a = ones(1000*p,1000)
a =
    ddense object: 1000p-by-1000

>> % now executing 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 =
    0.9989
>> gg = sum(sum(e))
gg =
    2.5012e+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 v] = find(b > 0.95)
i =
```

```

ddense object: 50471p-by-1
j =
ddense object: 50471p-by-1
v =
ddense object: 50471p-by-1
>> q = sparse(i, j, v, 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	1000x1p	16000	ddense array (complex)
f	1x1000p	8000	ddense array
ff	1x1	8	double array
gg	1x1	8	double array
h	1000px1000p	16000000	ddense array (complex)
i	50471px1	403768	ddense array
j	50471px1	403768	ddense array
q	1000px1000	617652	dsparse array
r	1000px1000	629160	dsparse array
rows	1x1	258	dlayout array
s	1000px1	8000	ddense array
t	1x4	32	double array
v	50471px1	403768	ddense array

```

Grand total is 6256321 elements using 58490422 bytes
MATLAB has a total of 7 elements using 306 bytes
Star-P server has a total of 6256314 elements using
58490116 bytes

```

As long as the size of the resulting array(s) is dependent on the size of the input array(s) and hence will likely be used in further parallel computations, the output array(s) 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, and `t`, whose size is based on the size of a value passed to `svds`. Even in cases where

dimensionality is reduced, as for `find`, when the resulting object is large, it is created as distributed.

Reusing Existing Scripts

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

```
%HILB Hilbert matrix.
% HILB(N) is the N by N matrix with elements 1/(i+j-1),
% which is a famous example of a badly conditioned matrix.
% See INVHILB for the exact inverse.
%
% This is also a good example of efficient MATLAB programming
% style where conventional FOR or DO loops are replaced by
% vectorized statements. This approach is faster, but uses
% more storage.

%
% See also INVHILB.

% C. Moler, 6-22-91.
% Copyright 1984-2004 The MathWorks, Inc.
% $Revision: 5.10.4.3 $ $Date: 2004/07/05 17:01:18 $

% I, J and E are matrices whose (i,j)-th element
% is i, j and 1 respectively.

J = 1:double(n);
J = J(ones(n,1),:);
I = J';
E = ones(n,n,'double');
H = E./(I+J-1);
```

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.

```
>> H = hilb(4096*p)
```

By exploiting MATLAB's object-oriented features in this way, existing scripts can run in parallel under Star-P without any 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. The text description of 10 million floating-point numbers, for example, 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 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.

```
>> size(a)
ans =
    1000p    1000
>> a(423,918)
ans =
    0.9362
>> a(2,3), a(2,3)=5; a(2,3)
ans =
    0.1422
ans =
     5

>> a(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** (`ppfront` is equivalent to the `pp2matlab` command) to move them to the front-end.

```
>> ppfront(a(1:5,1:5))
ans =
    0.5173    0.9226    0.0450    0.3776    0.8884
    0.6571    0.5675    0.6616    0.7332    0.3847
    0.6068    0.4153    0.9904    0.7291    0.3559
    0.6706    0.3300    0.2009    0.9978    0.6714
    0.6509    0.8918    0.3306    0.0526    0.6729
```

The ppeval Function: The Mechanism for Task Parallelism

In the previous examples, 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 course-grained parallel computation where operations are conducted on blocks, course-grains, of the data. This course-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).

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

```
[o1 o2 ... oN]= ppeval('function', i1, i2, ..., iN)
```

function is either a built-in MATLAB function or a user-supplied MATLAB function in a `.m` file. The input arguments can be either arrays or scalars, and can be distributed in several ways (see below for details). The output arguments `o1` through `oN` correspond to the output argument(s) of *function*. Currently `ppeval` is defined only for dense distributed arrays.

The semantics of `ppeval` are that the input array(s) are partitioned on the last dimension (i.e., columns for 2D matrices) and the function is called on each column of the input, in parallel. The output arguments will have the same size in the last dimension as the first input argument. You can view **`ppeval`** as a parallel loop. You cannot assume anything about the order in which the iterations occur or the processor(s) they occur on. For example, trying to pass intermediate results from one iteration to another will give undefined results.

Input Arguments

The input arguments, other than the function name, can be recast in several ways to match the needed behavior.

- default: the argument is split by columns, if it's a `ddense` or `double` array (equivalent to `split(a, 2)`).
- default: If the argument is scalar, it is broadcast. Every invocation of *function* gets the entire argument. If the argument is a vector or an array, the `bcast()` of that array must be made explicitly to broadcast that variable.
- `split(ddense, d)` : split along dimension `d`.
- `split(ddense, 0)` : split into its constituent elements. If the matrix is `m x n`, then there will be `m x n` calls to *function*, with each element of the matrix.
- `bcast(a)` : `a` is broadcast (passed to all calls to *function*), whether it's a scalar, vector or a matrix.

Output Arguments

Output arguments from *function* must be the same size for each invocation. The output is reshaped before `ppeval` returns. In general, if there are `n` invocations of *function*, then each resulting array will be the size of the individual corresponding return value from *function* with an additional dimension of length `n`.¹

NOTE: The use of `*p` to make objects distributed and thereby make operators parallel can almost always be made backwards compatible with MATLAB itself by setting `p = 1`. However, the use of `ppeval` is not trivially backward compatible with MATLAB.

Examples

Let's 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.

NOTE: `ppfront` is equivalent to the `pp2matlab` command.

```
>> n = 100
n =
    100
>> a = 1:n*p
a =
        ddense object: 1-by-100p
>> b = repmat(a,n,1)
b =
        ddense object: 100-by-100p
>> ppfront(b(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
>> c = ppeval('sum',b)
c =
        ddense object: 1-by-100p
>> ppfront(c(1,1:6))
ans =
    100    200    300    400    500    600
>> e = ppeval('sum',split(b,1))
e =
```

-
1. Earlier versions of the `ppeval` function did not automatically call `reshape` on the output. For backwards compatibility, this prior functionality is still accessible via the function `ppevalsplit`.

```

ddense object: 1-by-100p
>> ppfront(e(1,1:6))
ans =
    5050    5050    5050    5050    5050    5050

```

- The first call in the previous example to `ppeval` just uses the default behavior to split its arguments along the last dimension (columns, in the case of two-dimensional).
- The variable `b` in the previous example does not need to be distributed, created with `*p` construct or transferred to the server using `ppback` (`ppback` is equivalent to the `matlab2pp` command), because `ppeval` will automatically handle distribution to the server.
- The second call to `ppeval` wants to split it along rows, so it has to use the `split` function to get that behavior.

In this example, `'sum'` is the function called on each column of the input array. While useful for a simple example, you would never do this in practice, because the `sum` operator on the whole array has the same behavior and it's simpler to use. However, as shown in an example below, the `'function'` argument does not have to perform the same computation for each input, and thus can be used to implement so-called MIMD (multiple instruction multiple data) or task parallelism. (Many users think of the `sum` operator as data parallel, by contrast.)

The function `'function'` can come from one of two sources: it can be from

- a built-in set of MATLAB operators, with a complete list in the **Star-P Command Reference Guide²**, or
- a function you write yourself.

In either case, the function will need to operate on the split of the input matrices. As with any function called from MATLAB (or Star-P), the function must exist in a file of the name `function.m` in one of the directories in the MATLAB directory search list on the system where the Star-P client is running. As well as the `function.m` file itself, files containing any functions that are called by `function`, 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 there.

Another example that exploits the MIMD nature of the `ppeval` function uses the `quad` function, 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.

If the file `myfun.m` contains the function:

```
function b = myfun(a)
b = (a^2-1) / (a+eps);
```

then `ppeval` can be called as follows.

```
>> n = 100;           %number of intervals
>> a = (0:n-1*p)/n    %lower bounds of intervals
a =
    ddense object: 1-by-100p
>> b = (1:n*p)/n      %upper bounds of intervals
b =
    ddense object: 1-by-100p
>> c = ppeval('quad',@myfun,a,b)
c =
    ddense object: 1-by-100p
>> ppfront(c(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 (`@myfun`).

[2] 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.

The `ppeval` function can also be used to call a non-MATLAB program, via the MATLAB `system` operator, 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.

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

Calling this from `ppeval` works as follows:

```
>> a = ppback([503 510 512 514 516 523 524 817])
a =
    ddense object: 1-by-8p
>> b = ppeval('callapp2',split(a))
b =
    ddense object: 1p-by-8p
```

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

```
>> ppfront(b)
ans =
    0    7   31    0    3   14   11    0
```

You may want to note several things about this example.

- First, 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.
- Second, there is no built-in way for each column to know which column of the input it is. If that's necessary for some reason, you'd need to create such a vector and pass it in as another argument, for example by:

```
>> b = ppeval('callapp2',a,1:size(a,2))
```

- Third, 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 for you. Screen output

from the called function will not appear on the Star-P client. If you're reading or writing files in the called function, you'll need to do those via paths relative to the file system structure on the server system, not the client MATLAB current working directory, as that is a client notion, not a server notion. (Of course, if the filesystems 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`, including end-user applications (written in C or Fortran or ...) or third-party applications such as ANSYS, NASTRAN, FLUENT, or Gaussian.

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, and indeed causes MATLAB to abort. 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.

It's Still MATLAB: Using MATLAB Features on Parallel Programs

Great care has been taken in the design of Star-P constructs to make them fit naturally into the MATLAB style. One of the benefits of this is that many MATLAB features work in the obvious way for parallel programs or distributed data.

An example is program development using the MATLAB debugger. You may be used to prototyping a piece of code interactively with MATLAB, then inserting it into a `.m` file and confirming it works by stepping through it with the MATLAB debugging commands.

Consider the example in "[The ppeval Function: The Mechanism for Task Parallelism](#)" using `myfun`. Assume that the script from this example has been placed in a file `ppe3.m` as follows:

```
n = 100;           %number of intervals
a = (0:n-1*p)/n
b = (1:n*p)/n
c = ppeval('quad',@myfun,a,b)
ppfront(c(1,1:6))
```

The file `myfun.m` contains the following:

```
function b = myfun(a)
b = (a^2-1) / (a+eps);
```

You can step through the program with the following string.

```
>> clear
>> dbstop in ppe3 at 3
>> ppe3
n = 100;           %number of intervals
a = (0:n-1*p)/n
a =

        ddense object: 1-by-100p

K>> ppwhos
Your variables are:
      Name      Size      Bytes      Class
      a         1x100p     800       ddense array
      n          1x1        8        double array

Grand total is 101 elements using 808 bytes
MATLAB has a total of 1 elements using 8 bytes
Star-P server has a total of 100 elements using 800
bytes

K>> ppfront(a(1,1:20))
ans =
Columns 1 through 7
0         0.0100    0.0200         0.0300    0.0400    0.0500         0.0600
Columns 8 through 14
0.0700    0.0800    0.0900         0.1000    0.1100    0.1200         0.1300
Columns 15 through 20
0.1400    0.1500    0.1600         0.1700    0.1800    0.1900

K>> dbstep
b = (1:n*p)/n
        ddense object: 1-by-100p
c = ppeval('quad',@myfun,a,b)

K>> ppwhos
Your variables are:
      Name      Size      Bytes      Class
      a         1x100p     800       ddense array
      ans        1x20       160       double array
      b         1x100p     800       ddense array
      n          1x1        8        double array

Grand total is 221 elements using 1768 bytes
MATLAB has a total of 21 elements using 168 bytes
```

Star-P server has a total of 200 elements using 1600 bytes

```
K>> ppfront(b(1,1:20))
ans =
Columns 1 through 7
    0.0100 0.0200 0.0300 0.0400 0.0500 0.0600 0.0700
Columns 8 through 14
    0.0800 0.0900 0.1000 0.1100 0.1200 0.1300 0.1400
Columns 15 through 20
    0.1500 0.1600 0.1700 0.1800 0.1900 0.2000

K>> dbcont
c =
    ddense object: 1-by-100p
ppfront(c(1,1:6))
ans =
   -31.4384   -0.6930   -0.4052   -0.2873   -0.2227   -0.1818
```

```
>> ppwhos
Your variables are:
      Name      Size      Bytes      Class
      a         1x100p     800       ddense array
      ans        1x6        48       double array
      b         1x100p     800       ddense array
      c         1x100p     800       ddense array
      n          1x1         8       double array
```

Grand total is 307 elements using 2456 bytes
 MATLAB has a total of 7 elements using 56 bytes
 Star-P server has a total of 300 elements using 2400 bytes

The overloading of the colon (":") operator by Star-P and the operation of ppeval doesn't change the usefulness of the MATLAB debugger. It allows you to step through your .m files as you would with regular MATLAB code. (Of course, there are levels of Star-P code you can't step through, just as there is with MATLAB code; i.e., the low-level software that's calling built-in functions and the built-in functions themselves.)

Similarly, the user interface functions in MATLAB, like the arrow keys that let you scroll back to prior commands, are just as useful in Star-P as in MATLAB proper.

Configuring Star-P for High Performance

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

1. With Star-P release 2.3 running in conjunction with MATLAB release 7 SP3 or later, Star-P takes advantage of an optimized client implementation which significantly improves the performance of each individual call to the Star-P server. When using older versions of Star-P or using Star-P with MATLAB release 7 SP2, this optimized client is not enabled.
2. 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')
```

3. The client provides a list of unused matrices to the server on a regular basis so that the server can free up memory which is no longer being utilized. By default, this happens once every 30 server calls. Decreasing the frequency of this garbage collection calls may increase the average amount of memory consumed by the server processes, but can improve overall run time by reducing the overhead associated with these calls. The server will initiate a GC on its own if it's running low on memory, without waiting for a regularly scheduled GC to occur. The frequency of client garbage collection requests can be adjusted via the following command, where in this example the frequency is reduced to once every 100 calls.

```
ppsetoption('ppgcFreq',100)
```

By 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.

4. 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')
```

5. 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')
```

4 Structuring Your Code for High Performance and Parallelism

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” on page 60)
- 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.

There is one piece of good news. Very often the changes to programs 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. Star-P is today 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.

Parallel Computing 101

Textbooks on parallel computing will list a variety of models for parallelizing programs

- data parallel
- message passing
- embarrassingly parallel computation
- partitioning and “divide and conquer”
- pipelining

You may wish to check the Internet¹ or any of the numbers of the 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: $C=A+B$, where A and B are matrices. When we add two $n \times n$ matrices, we perform n^2 data parallel additions; i.e., 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 or 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 ($C=A*B$) and prefix sums ($b=cumsum(a)$).

¹ For example, a web site that has several points related to parallel computing is <http://beowulf.csail.mit.edu/18.337>.

The description of Star-P that is most useful for most users is as a global array syntax language, meaning that when you work in Star-P, the user variable `A` refers to the entirety of a distributed object on the back end. The power of this is probably obvious. The abstraction of an array that contains many elements is a powerful construct. With one variable name such as `A`, 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 in the language places more of a cognitive burden on you the programmer.

Embarrassingly parallel computations are those 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, it is typical that such computations 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.

Partitioning and *divide & conquer* are strategies whereby a problem is divided in half, and each subpiece is solved simultaneously.

Pipelining is a process whereby the computation goes through various stages with a steady state stream of data moving through the stages.

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 way. A global array that is an input to the `ppeval` function is partitioned into sections, each of which is converted to an array local to a single instance of a MATLAB function. The reverse process is used for output arrays; the assemblage of the sections into global arrays.

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 by Star-P 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
>> s
Elapsed time is 1.219000 seconds.
s =
5.0000e+011
```

The following line is vectorized:

```
>> tic, s=sum(v); toc, s;
Elapsed time is 0.015000 seconds.
```

The two ways of summing the elements of *v* give the same answer, yet the vectorized version using the *sum* operator runs almost 100 times faster. This is an extreme case of the speed-up due to vectorization, but

not rare. The main point is that as you express your algorithm in high level operators, you allow more opportunities for optimization by Star-P (or MATLAB) developers within those operators, resulting in better performance.

The following MATLAB code is parallelized:

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

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=1:1e8;
>> w=0*v;
>> tic
>> for i=1:length(v)
>>     w(i)=v(i)^3+2*v(i);
>> end
>> toc
```

Elapsed time is 32.642216 seconds.

The following code is vectorized:

```
>> tic
>> w=v.^3+2*v;
>> toc
```

Elapsed time is 18.008545 seconds

The following code is parallelized:

```
>> tic
>> v=1:1e8*p;
>> w=v.^3+2*v;
>> toc
```

Elapsed time is 3.097014 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; namely, that 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`:

```
for i=1:n
    for j=1:n
        c(i,j) = dot(a(i,:),b(:,j));
    end
end
>> n = 1000
n =
    1000
>> a = rand(n); b = rand(n);
>> tic, mxm, toc           % unvectorized
Elapsed time is 46.197213 seconds.
>> tic, d = a*b; toc       % vectorized on a single processor
Elapsed time is 0.565492 seconds.
>> n = 1000*p
n =
    1000p
>> a = rand(n); b = rand(n);
>> tic, d = a*b; toc       % vectorized on 4P
Elapsed time is 0.234927 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 4.797000 seconds.
w =
Columns 1 through 8
998923   1998531   2996411   3996445   4998758   6000422   6999318   8000845
Columns 9 through 11
9001023   10000000   10000000
```

Form 2: Cumulative sum and histogram:

```
>> tic,cumsum(histc(v, 0:.1:1)),toc
ans =
Columns 1 through 8
998923   1998531   2996411   3996445   4998758   6000422   6999318   8000845
Columns 9 through 11
9001023  10000000   10000000
Elapsed time is 1.140000 seconds.
```

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.

```
>> a = rand(10^4,10^4);
>> aa = rand(10^4,10^4*p);
>> tic, b = fft(a); toc
Elapsed time is 225.866275 seconds.
>> tic, bb=fft(aa), toc
Elapsed time is 21.259804 seconds.
```

While you'd 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 slowly. 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 `a=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 14.344000 seconds.  
Elapsed time is 37.922000 seconds.  
Elapsed time is 4.015000 seconds.  
Elapsed time is 3.844000 seconds.  
Elapsed time is 3.844000 seconds.  
Elapsed time is 3.828000 seconds.  
Elapsed time is 3.657000 seconds.  
Elapsed time is 3.703000 seconds.  
Elapsed time is 3.687000 seconds.  
Elapsed time is 3.891000 seconds.  
Elapsed time is 3.969000 seconds.  
Elapsed time is 3.218000 seconds.  
Elapsed time is 3.063000 seconds.  
Elapsed time is 2.890000 seconds.  
Elapsed time is 2.938000 seconds.  
Elapsed time is 3.250000 seconds.  
Elapsed time is 3.453000 seconds.  
Elapsed time is 4.078000 seconds.  
Elapsed time is 4.641000 seconds.  
Elapsed time is 6.219000 seconds.  
Elapsed time is 8.812000 seconds.  
Elapsed time is 12.719000 seconds.  
Elapsed time is 22.797000 seconds.  
Elapsed time is 42.765000 seconds.  
Elapsed time is 70.141000 seconds.  
Elapsed time is 136.438000 seconds.
```

Solving Large Problems: Memory Issues

The power of MATLAB and Star-P, their ability to easily make large matrices and manipulate them, 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 MATLAB¹ 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.

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

If you are running into memory capacity issues, 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.

5 Data Organization in Star-P

Communication between the Star-P Client and Server

The `matlab2pp` command is replaced by `ppback` and `pp2matlab` is replaced by `ppfront`.

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, its overuse can cause performance problems, 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** (matlab2pp command is equivalent to `ppfront`), **dsave**, 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      %load data for variable a from data file
>> aa = ppback(a) ;
>> ppwhos
```

Your variables are:

Name	Size	Bytes	Class
a	100x100	80000	double array
aa	100px100p	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 **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.

NOTE `ppfront` is equivalent to the `pp2matlab` command



```
>> bb=aa.*aa;
>> c = ppfront(bb);
>> ppwhos
```

Your variables are:

Name	Size	Bytes	Class
a	100x100	80000	double array
aa	100px100p	80000	ddense array
bb	100px100p	80000	ddense array
c	100x100	80000	double array

Grand total is 40000 elements using 320000 bytes

MATLAB has a total of 20000 elements using 160000 bytes

Star-P server has a total of 20000 elements using 160000 bytes

For input 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 ["Using Star-P"](#).

Implicit Communication

The communication between the client and the server can also be implicit. The most frequent case of this 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 as much as possible minimizing the use of control structures such as, `for` and `while`, with operators that match what you want to achieve. This is discussed more fully in ["Solving Large Sparse Matrix and Combinatorial Problems with Star-P"](#).

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(bb));
>> e = norm(bb);
>> ppwhos
```

Your variables are:

Name	Size	Bytes	Class
bb	100px100p	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 reductions mentioned above are an exception to that, as the results of reduction operations are usually much smaller than the original data, and when they are reductions to scalars, as in the case above, the result is often then used in other MATLAB client computations.

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(10,10)
f =
  Columns 1 through 7
  0.0159    0.5943    0.0550    0.5165    0.5074    0.6594    0.1353
  0.6283    0.3020    0.6756    0.2126    0.2919    0.9234    0.0285
  0.8125    0.5442    0.3081    0.4850    0.1834    0.5238    0.8892
  0.2176    0.2362    0.1911    0.4113    0.4514    0.8555    0.5510
  0.4054    0.1605    0.7659    0.2088    0.5771    0.6777    0.6685
  0.5699    0.4533    0.4741    0.2452    0.3589    0.1297    0.9616
  0.4909    0.7413    0.2572    0.0396    0.7385    0.0921    0.7800
  0.1294    0.6009    0.3252    0.8854    0.0861    0.6624    0.2836
  0.5909    0.5841    0.0845    0.0348    0.4469    0.7109    0.7718
  0.8985    0.2706    0.6618    0.6223    0.2273    0.8290    0.4244
  Columns 8 through 10
  0.1503    0.7455    0.1551
  0.1032    0.7627    0.6196
  0.6944    0.7827    0.7581
  0.3858    0.8807    0.5107
  0.9967    0.2052    0.9383
  0.2912    0.7732    0.4046
  0.8826    0.9212    0.9974
  0.0404    0.4695    0.3764
  0.0542    0.5072    0.6043
  0.6427    0.4903    0.5947
```

While this makes good sense for small data sizes, printing out the data sizes possible with Star-P distributed objects (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.

```
>> ff=rand(10*p,10)
ff =
      ddense object: 10p-by-10
```

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:

```
>> ff(1,4)
ans =
      0.4528
```

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

```
>> fsub = ppfront(ff(1:4,:));
>> ppwhos
Your variables are:
      Name      Size      Bytes      Class
      f         10x10      800      double array
      ff        10x10p     800      ddense array
      fsub       4x10      320      double array
```

```
Grand total is 40242 elements using 321936 bytes
MATLAB has a total of 20142 elements using 161136 bytes
Star-P server has a total of 20100 elements using 160800
bytes
```

Note that when you call **ppfront** (which is equivalent to the `pp2matlab` command) and leave off the final semicolon, MATLAB will print out the whole contents of the array, which may not be what you want.

Note that communication can happen implicitly as described in ["Mixing Local and Distributed Data"](#).

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.

```
>> tic, b = fft(a), toc
b =
    ddense object: 1000-by-1000p
Elapsed time is 2.472861 seconds.
>> pptic, b = fft(a), pptoc
b =
    ddense object: 1000-by-1000p
Client/server communications stats
  Send msgs/bytes    Recv msgs/bytes
    8e+00 / 4.19e+02B  8e+00 / 9.91e+02B
```

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

```
>> tic, pptic, b = fft(a), toc, pptoc
b =
    ddense object: 1000-by-1000p
Elapsed time is 2.592732 seconds.
Client/server communications stats
  Send msgs/bytes    Recv msgs/bytes
    8e+00 / 4.19e+02B  8e+00 / 9.91e+02B
```

The `pptic` and `pptoc` commands can be used on various amounts of code, to home in 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 quite a number of bytes moved.

NOTE: `ppfront` is equivalent to the `pp2matlab` command.

```
>> pptic, ma = ppfront(a); pptoc
Client/server communications stats
  Send msgs/bytes    Recv msgs/bytes
    1e+00 / 8.00e+06B  1e+00 / 1.13e+02B
>> ppwhos
Your variables are:
```

Name	Size	Bytes	Class
a	1000x1000p	8000000	ddense array
ma	1000x1000	8000000	double array

But there might be places where implicit data movement occurs when you are not expecting it. For example, an early version of the **find** operator for ddense arrays in one case moved an array to the client because a certain function was not yet implemented on the server, then moved the array back to the server. This approach worked, but it was much slower than expected. **pptic/pptoc** allows you to see unexpected data movement like this.

```
>> pptic, i = find(a > 0.5), pptoc
i =
      ddense object: 499534p-by-1
Client/server communications stats
  Send msgs/bytes    Recv msgs/bytes
    2e+01 / 9.51e+02B  2e+01 / 2.22e+03B
>> pptic , [i j v] = find(a > 0.5), pptoc
i =
      ddense object: 499534p-by-1
j =
      ddense object: 499534p-by-1
v =
      ddense object: 499534p-by-1
Client/server communications stats
  Send msgs/bytes    Recv msgs/bytes
    4e+01 / 1.20e+07B  4e+01 / 1.20e+07B
```

The single-return case of **find** moves a few hundreds or thousands of bytes between the client and the server. The three-return case moves the whole array, showing up as megabytes of data moved.

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.

```
tic, pptic
for i=1:double(size(a,1))
    c(i,1) = a(i,1)*2 + b(i,1)*7.4;
end
toc, pptoc
```

```
Elapsed time is 975.796453 seconds.
Client/server communications stats
  Send msgs/bytes    Recv msgs/bytes
    4e+03 / 2.05e+05B  4e+03 / 5.90e+05B
```

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. (See "[Solving Large Sparse Matrix and Combinatorial Problems with Star-P](#)" for more information on the benefits and methods of vectorization of MATLAB or Star-P programs.)

```
tic, ptic
c = a(:,1)*2 + b(:,1)*7.4;
toc, ptoc
```

```
Elapsed time is 4.853982 seconds.
Client/server communications stats
  Send msgs/bytes    Recv msgs/bytes
    2e+01 / 1.07e+03B  2e+01 / 2.32e+03B
```

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.

```
>> clear
>> a=rand(1000,1000*p);
>> ppprofile on
>> doffts

echo on
b = fft(a);
c = ifft(b);
diff = max(max(abs(c-a)))
diff =
    7.7747e-16
if diff > 100*eps
>> ppprofile report
```

function	calls	time	avg time	%calls	%time
ppfftw_fft	2	0.30054	0.15027	22.2222	34.2312
ppdense_abs	1	0.14641	0.14641	11.1111	16.6756
ppdense_max	2	0.14066	0.070328	22.2222	16.0202
ppdense_elminus	1	0.11991	0.11991	11.1111	13.6568
ppdense_viewelement	2	0.11495	0.057473	22.2222	13.092
ppbase_removeMatrix	1	0.055525	0.055525	11.1111	6.3241
Total	9	0.87799	0.097554		

The report prints out all the server functions that are used between the calls to **pppprofile on** and **pppprofile 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 way, compared to the vectorized way. The script has these contents:

```
function c = domxmp(a,b)
%   Do matrix multiply by various methods (bad to good
perf)
%
%   First, do it in an unvectorized style.

[m n] = size(a);

pppprofile clear, ppprofile on
tic
c = zeros(m,n);
for i=1:double(m)
    for j = 1:double(n)
        c(i,j) = dot(a(i,:),b(:,j));
    end
end
fprintf('MxM via ddot takes ');
toc
pppprofile report

%   Second, do it in a vectorized style

pppprofile clear, ppprofile on
tic
```

```

c = a*b;
fprintf('MxM via dgemm takes ');
toc
pppprofile report

```

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

```

>> a = rand(20,20*p)
a =
      ddense object: 20-by-20p
>> b = rand(20,20*p)
b =
      ddense object: 20-by-20p
>> c = domxmp(a,b)
MxM via ddot takes Elapsed time is 163.300579 seconds.

```

function	calls	time	avg time	%calls	%time
ppdense_subsref_col	400	30.8712	0.077178	19.333	20.1097
ppdense_subsref_row	400	30.2725	0.075681	19.333	19.7197
ppdense_transpose	400	29.9194	0.074799	19.333	19.4897
ppblas_dot	400	29.182	0.072955	19.333	19.0093
ppdense_setelement	400	28.3638	0.070909	19.333	18.4763
ppbase_gc_many	68	4.8412	0.071194	3.2866	3.1536
ppdense_zeros	1	0.064182	0.064182	0.048333	0.041808
Total	2069	153.5143	0.074197		

MxM via dgemm takes Elapsed time is 0.084823 seconds.

function	calls	time	avg time	%calls	%time
ppblas_gemm	1	0.060883	0.060883	100	100
Total	1	0.060883	0.060883		

```

c =
      ddense object: 20-by-20p

```

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 difference 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 **pppprofile display** option.

```

>> ppprofile off
>> ppprofile on, ppprofile display
>> doffts

echo on

```

```

% fft/iff create output the same size as the input,
hence will
% preserve distributed attribute
%size(a)
b = fft(a);
ppfftw_fft   time=0.13424
%size(b)
c = ifft(b);
ppfftw_fft   time=0.24997
%size(c)
% max reduces the dimensionality of its input; when it
becomes a
% scalar, will bring it back to the front-end
diff = max(max(abs(c-a)))
ppdense_elminus   time=0.10178
ppdense_abs       time=0.14426
ppdense_max       time=0.071176
ppbase_removeMatrix   time=0.055262
ppdense_max       time=0.065765
ppdense_viewelement   time=0.055748
ppdense_viewelement   time=0.055729
diff =
    7.7963e-16
%size(diff)
if diff > 100*eps

```

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 **pppprofile on -detail full** option coupled with the **pppprofile display** option.

```

>> ppprofile off
>> ppprofile on -detail full
>> ppprofile display
>> doffts
echo on
b = fft(a);
ppfftw_fft   ppbase_gc_many   time=0.11616   stime=0
chdist=0
time=3.1397   stime=0.078125   chdist=0
c = ifft(b);
ppfftw_fft   time=0.33294   stime=0.10938   chdist=0

```

```

diff = max(max(abs(c-a)))
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
if diff > 100*eps

```

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.

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.

```

>> aa = rand(100*p,100);
>> bb = rand(100*p,100);
>> cc=aa+bb;
>> dd=aa.*bb;
>> ppwhos
Your variables are:

```

Name	Size	Bytes	Class
aa	100px100	80000	ddense array

bb	100px100	80000	ddense array
cc	100px100	80000	ddense array
dd	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, `aa` and `ee` are distributed differently, so internally Star-P redistributes `ee` to the same distribution as `aa` before doing the element-wise operations.

```
>> ee=rand(100,100*p) ;
>> ff=aa.*ee;
>> ppwhos
```

Your variables are:

Name	Size	Bytes	Class
aa	100px100	80000	ddense array
ee	100x100p	80000	ddense array
ff	100px100	80000	ddense array

Grand total is 60000 elements using 480000 bytes
 MATLAB has a total of 0 elements using 0 bytes
 Star-P server has a total of 60000 elements using 480000 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. Creating a submatrix by indexing into a distributed array does too, for the same reason.

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 (as described in "[Solving Large Sparse Matrix and Combinatorial Problems with Star-P](#)") 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.

Types of Distributions

Distributed Dense Matrices and Arrays

Star-P uses the MATLAB terminology of 2-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.

Two-Dimensional Block Cyclic Distribution

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. Each process receives only one block of the matrix. These distributions may suffer from load balancing problems.

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

```
>> aa = rand(100*p, 100 );
>> bb = rand(100 , 100*p);
>> cc = rand(100*p, 100*p);
```

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

Row distribution

In the example above, *aa* is created with groups of rows distributed across the memories of the processors in the parallel server. Thus, with 10,000 rows on 16 processors, the first 10,000/16 = 625 rows would be on the first processor, the next 625 on the second processor, and so forth, in a style known as row-distributed. Figure 5-1 illustrates the layout of a row-distributed array.

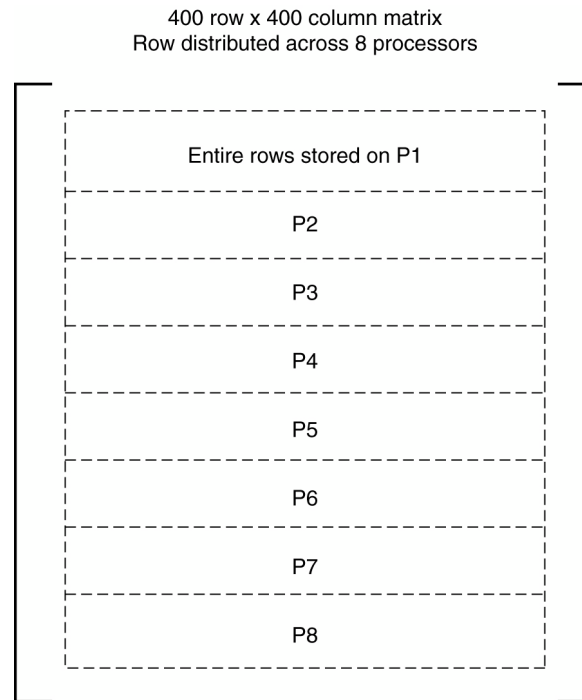


Figure 5-1 Row Distribution

Column distribution

Column-distributed works just the same as row distribution, except columns are distributed; *bb* is created that way above. Figure 5-2 illustrates the layout of a column distributed array.

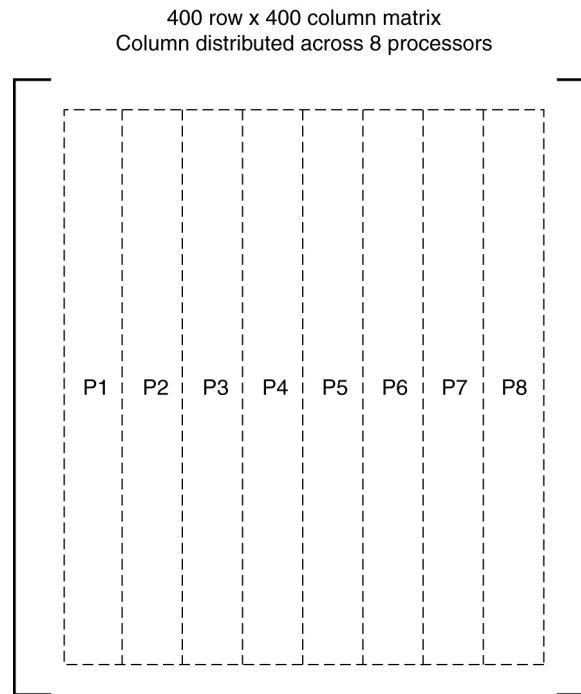


Figure 5-2 Column Distribution

Block-cyclic distribution

When multiple indices of an array are distributed, as in the creation of `cc` above, a distribution known as block-cyclic or just cyclic is used. In this distribution, the array is logically divided into blocks of some size (currently 64x64 in Star-P), and then those blocks are distributed across the processors as shown in Figure 5-3. The details of this distribution involve several indexing calculations, but they are often worthwhile. The block-cyclic distribution is a trade-off among load-balancing, amount of communication, and efficiency of local computation. ScaLAPACK is an example of a library that uses block-cyclic distribution.

A good example that illustrates the load-balancing problem is Gaussian elimination. The Gaussian elimination algorithm solves a square linear system by 'zeroing out' the subdiagonal entries.

```
for i = 1:n
  for j = i+1:n
    A(j,i:n) = A(j,i:n) - A(i,i:n)*A(j,i)/A(i,i);
  end
end
```

The algorithm proceeds column by column from left to right. For column distribution, as the algorithm goes past the columns stored in process i , process 0, ..., i will be idle for the rest of the computation. Row distribution suffers from a similar deficiency.

Two-dimensional block cyclic distribution avoids this problem by 'dealing out' blocks of the matrix to the processes in a cyclic fashion. This way all processes remain busy throughout the computation, thus achieving load balancing. This approach carries the additional advantages of allowing BLAS level-2 and BLAS level-3 operations on local blocks.

For further information on block-cyclic distribution, see <http://www.netlib.org/scalapack/slug/node75.html#SECTION04431000000000000000>

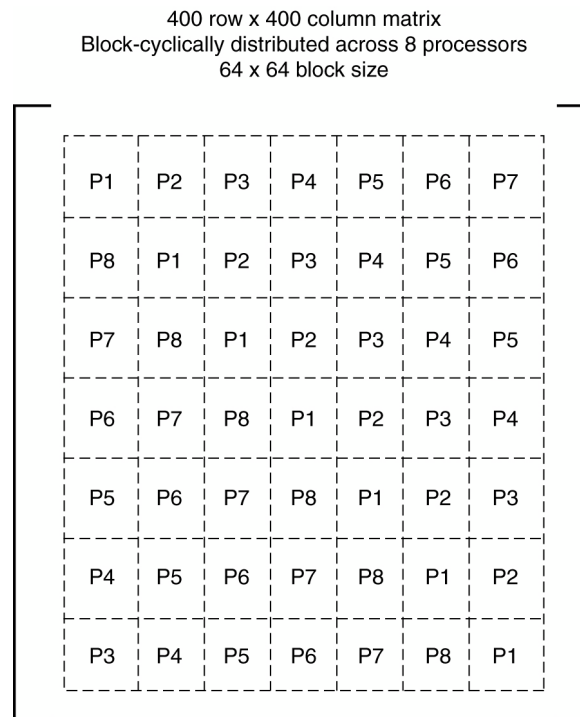


Figure 5-3 Block-Cyclic Distribution

Distributed Dense 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. The rationale for this is that arrays are often used as repositories of data, not for direct operations, and hence indexing and data movement operations need to be fast. Hence if you create a distributed

object with the following command, then `aa` will be distributed on the third dimension:

```
aa = rand(n,n,n*p,n)
```

If you should happen to request distribution on more than one dimension, a warning will be displayed and the last distributed non-singleton dimension of the matrix will be distributed.

```
>> a = zeros(10*p,10,10*p,10*p,10*p)
Warning: Do not support n-dimensional block
distributions yet.
> In error:<starp_root>/matlab/@dlayout/zeros.m, 59, 1
dlayout.zeros at 59
a =
      ddensend object:10-by-10-by-10-by-10-by-10p
```

Distributed multidimensional arrays are primarily intended as repositories for lower-dimensional matrices. Hence less than the full set of operators is supported for distributed dense arrays. See the full list in **Star-P Command Reference Guide**.

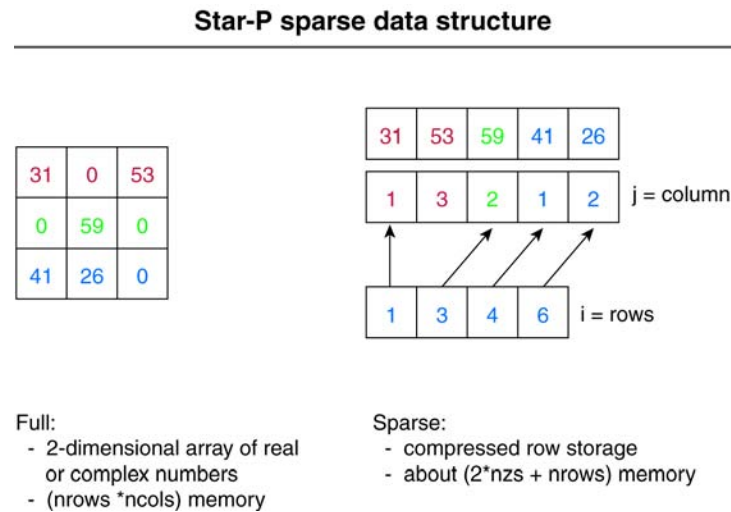
Distributed Sparse Matrices

Distributed sparse matrices in Star-P use the compressed sparse row format. 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 per-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.

For further information on sparse array formats, see http://ce.et.tudelft.nl/~robbert/sparse_matrix_compression.html.

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 5-4** 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 5-4, 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].

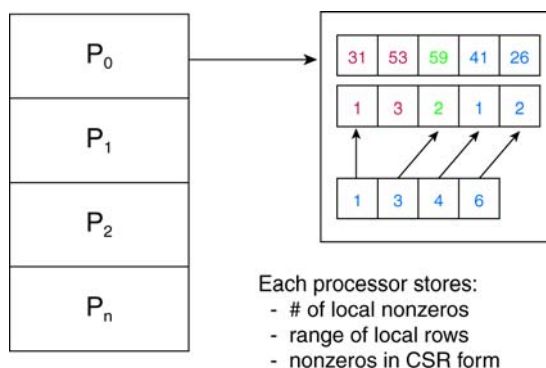
**Figure 5-5** Star-P Distributed Sparse Data Structures

Figure 5-5 shows what happens when the sparse data structure from Figure 5-4 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 Partitioned Objects (dpart)

The `dpart` type is different from the other distributed matrix or array types, as it contains the return from a `ppevalsplit` call, which may not have the same number of return elements per iteration and hence doesn't have the same degree of regularity as the other distributions. Because of this potential irregularity, a `dpart` 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 `dpart` 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 `dpart` arrays or manipulate them.

Propagation of Distribution

A natural question that is 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 that has been carefully implemented in the case of indexing and a large class of functions, and perhaps like the irregular verbs of a natural language, there are a number of special cases, some of which we list here that do not follow these rules.

In Star-P, the default is that other than performance and floating point issues, the operation should not depend on the distribution. We therefore do not guarantee that in future releases the distribution rule will remain the same.

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

<code>ddense</code>	row, column, block cyclic
<code>ddensend</code>	linear distribution along any dimension
<code>dsparse</code>	row distribution only

Note that block cyclic will likely be deprecated as an externally available distribution in future releases of Star-P.

Rule #1:

For many 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: `cos(A), A.^2, lu(A), fft(A), fft2(A)` (for `ddense` and `ddensend` where applicable)

Exceptions: Many Linear Algebra functions do not follow this rule. See below.

Rule #2: For many basic functions of two arguments where the sizes of the two inputs are the same and match the size of the output, the output distribution for `ddense` arrays matches that of the first input, unless that input is on the front end, in which case the second input is used. For `ddensend` arrays, the resulting distribution follows the array that is distributed on the highest dimension.

Examples: `A+B, A.*B, A*B` (where applicable).

NOTE: Note backslash and forward slash do not follow this rule.

Rule #3: Indexing, transpose, `horzcat`, `vertcat`, `reshape` and a number of other functions follow a calculus of distributions rule that is sometimes unintuitive, but careful thought indicates that such a calculus is almost necessary to logically account for the myriad of cases that are confronted. This rule is deferred until later in this section. Suffice it to point out that indexing can turn a tall fat matrix (which probably ought to be row distributed) into a short and skinny matrix (which really ought to be column distributed). This point alone should illustrate that the simple expectation of following the original arrays distribution is by itself misguided.

Rule #4: (The irregular verb rule) The following operations do not have any particular logic to them and must be accounted for one by one. The following list is only the non-trivial cases.

Table 5-1 Single `ddense` arguments (irregular cases)

	1 output	2 outputs	3 outputs
<code>qr(ddense)</code> or <code>qr(ddense,0)</code>	column	matches input	column
<code>svd(ddense)</code> or <code>svd(ddense,0)</code> or <code>svd(ddense,'econ')</code>		block cyclic	block cyclic
<code>inv</code>	block cyclic		

Table 5-1 Single ddense arguments (irregular cases)

	1 output	2 outputs	3 outputs
eig(ddense) (no-sym)		block cyclic (not officially supported)	
eig(ddense) (sym)		[block cyclic, row]	
schur	block cyclic	block cyclic	

Table 5-2 Two ddense arguments (irregular cases)

	1 output
kron	column
backslash “\”	Matches 2nd argument, or block cyclic if the 2nd argument is local
forward slash “/”	Matches 1st argument or block cyclic if the 1st argument is local

Calculus of Distribution Rule:

There is a very elegant rule that is carefully implemented in the case of `subref`, `subsasgn`, `transpose` and a number of other functions:

In many cases, the size of the output is a function of the sizes of the input. The components of the input size when tagged with a “p” propagate in the computation.

- For `ddense` arrays, the distribution output follows the matrix constructor rules, for example:
 - a tag in the row results in a row distribution,
 - a tag in the column is a column distribution, and
 - a tag in both places is block cyclic.
- For `ddensend` arrays, the right-most tag is used. An extra propagate the parallelism rule applies if the result would be local according to the above rules. This rule gives the same default as `ppback` would give.

For example, the output of the element-wise cosine function (`cos`) has the same dimension sizes as the input, so the dimension's distributions are also preserved.

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

```
>> B = cos(A)
B =
ddense object: 1000p-by-4
```

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

```
>> B = A'
B =
ddense object: 4-by-1000p
```

As a less trivial example, consider $C = \text{kron}(A, B)$. The size of the dimensions of C are calculated through the following formula:

```
size(C) = size(A) * size(B)
```

The resulting distribution follows the irregular verb Rule #4:

```
>> A = rand(1000*p, 4);
>> B = rand(10, 100*p);
>> C = kron(A,B)
C =
ddense object 10000-by-400p
```

Operators that combine two or more matrices of the same size inherit the distribution attributes from the first distributed matrix according to RULE #2. For example:

```
>> A = rand(1000, 4);
>> B = rand(1000*p, 4);
>> C = A + B
C =
ddense object: 1000p-by-4
```

Here, $\text{size}(C)$ is defined as $\text{size}(A)$ OR $\text{size}(B)$, so the distribution attributes are inherited first from A , or from B if none of A 's dimensions are distributed.

To propagate the distributed attribute as frequently as possible, there is an important extension to this general rule: If some of the inputs to an operator are distributed, but none of the distributed dimension sizes propagate through to the output, then the output will be distributed anyway, as though `ppback` had been invoked on it. In this case, the last output dimension with size greater than one will be distributed.

Operators that reduce the dimensionality of their input, such as `max`, `sum` or `find`, are often subject to this extension:

```
>> A = rand(5000*p, 1000);
>> B = sum(A, 1)
```

```
B =
ddense object: 1-by-1000p
```

Here, B is distributed, even though the distributed row dimension "5000*p" is not propagated from A to B.

Let's look at a few more examples, such as matrix multiplication:

```
>> A = rand(1000*p, 1000);
>> V = rand(1000*p, 1);
>> C = A * V
```

```
C =
ddense object: 1000p-by-1
```

In the previous example, `size(A * V)` is defined as `[size(A,1) size(V,2)]` which simplifies to `[1000p 1]`.

```
>> B = rand(1000, 1000*p);
>> C = A * B
```

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

According to RULE #2.

```
>> D = B * A
D =
ddense object: 1000-by-1000p
```

Here, `size(A * B)` is defined as `[size(A,1) size(B,2)]` which simplifies to `[1000 1000]`, and none of the distributed dimensions propagate through to the output, so the extension dictates the last dimension is distributed.

For `B = reshape(A, r, c)`, `size(B) = [r c]`, so we have:

```
>> A = rand(9, 4);
>> B = reshape(A, 6, 6*p)
B =
ddense object: 6-by-6p
>> C = reshape(B, 36, 1)
C =
ddense object: 36p-by-1
```

Indexing is a particularly thorny example, because `subsref` has many different forms. `B = A(:, :)` has the same distribution as A, because `size(B) = size(A)`. `B = A(:)` vectorizes (linearizes) the

elements of A , so the output will be row or column distributed accordingly.

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

```
>> A = rand(10*p,10);
ddense object: 10p-by-10
>> I = ppback(magic(10))
ddense object: 10p-by-10p
>> B = A(I)
B =
ddense object: 10p-by-10p
```

But, consider $B = A(R, C)$ with the following:

```
>> R = randperm(100*p)
R =
ddense object: 1-by-100p
>> C = randperm(100);
>> B = A(R,C)
B =
ddense object: 100p-by-100
```

Here, $\text{size}(B)$ is defined as $[\text{prod}([1 \ 100p]) \text{prod}([1 \ 100])]$ $[100p \ 100]$ which simplifies to $[\text{prod}([10p \ 10p]) \ \text{prod}([10 \ 10])]$ and then $[100p \ 100]$. So the distribution of

- B's row dimension is inherited from both of R's dimensions, and
- B's column dimension is inherited from C.

As a final example, consider logical indexing:

```
>> A = rand(100*p,100);
>> I = A > 0.5
I =
ddense object: 100p-by-100
>> B = A(I)
B =
ddense object: 5073p-by-1
```

This might be unexpected, but is so because $A(I)$ is essentially the same as $A(\text{find}(I))$, and $\text{find}(I)$ returns a row-distributed column vector.

To summarize:

- For many functions of one argument where the size of the output is the size of the input, the output distribution matches that of the input. However, many Linear Algebra functions do not follow this rule.
- For many basic functions of two arguments where the sizes of the two inputs are the same and match the size of the output, the output distribution for ddense arrays matches that of the first input, unless that input is on the front end, in which case the second input is used. For ddensend arrays, the resulting distribution follows the array that is distributed on the highest dimension. However, backslash and forward slash do not follow this rule.
- Indexing, transpose, horzcat, vertcat, reshape and a number of other functions follow a calculus of distributions rule that is sometimes unintuitive. Additionally several operations do not have a particular logic to them and must be accounted for on a case by case basis.

6 Application Examples

Application Example: Image Processing Algorithm

The matlab2pp command is replaced by ppback and pp2matlab is replaced by ppfront.

How the Analysis Is Done

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 plan. In other words, X marks the spot.

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.

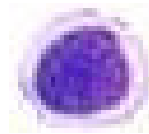


Figure 6-1 Target Image

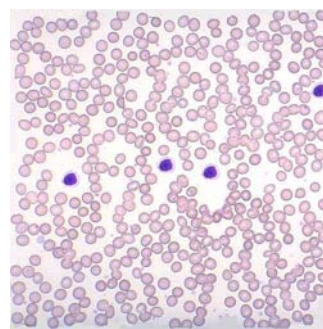


Figure 6-2 Input Image

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:

Table 6-1 M File Description

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.

Table 6-2 Application Example Not Using Star-P

Step	Description
1	The input image Figure 6-2 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.

Table 6-2 Application Example Not Using Star-P

Step	Description
3	A correlation calculation is performed on the HSV components of the input and target images. The <code>patchmatch_calc.m</code> 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.

1 Read in RGB data, convert to HSV	<pre>% 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;</pre>
2 Image is tiled and replicated	<pre>% Load the data, comes in RGB, transfer to HSV soace 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</pre>
3 Calculate the correlation on each of HSV components	<pre>% Perform correlation calculation in HSV space with H, S, and V on different processors d = zeros(size(a)); for i = 1:3 d(:,:,i) = patmatch_calc(a(:,:,i),b(:,:,i)); end</pre>
4 Perform the threshold	<pre>e = (1-d(:,:,2))>0.5 & d(:,:,3)>thres; %Threshold finding target within input image % Display the results figure(1) image(hsv2rgb(a));title('Input Image') figure(2) image(hsv2rgb(b));title('Filter Image')</pre>
5 Display the results	<pre>figure(3) imagesc(d(:,:,1));title('Correlation, H') figure(4) imagesc(1-d(:,:,2));title('Correlation, S') figure(5) imagesc(d(:,:,3));title('Correlation, V') figure(6) imagesc(e);colormap gray;title('Threshold Correlation')</pre>

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.

```
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:prod(size(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;
pad = fftshift(pad); %Adjust the filter to the FFT space
%
% Calculate the pattern match of input image a with the target filter b
%
c = ifft2(fft2(a).*conj(fft2(pad))); %Multiply Fourier transform of the input and target
d = c.*conj(c); %Measured optical intensity
corr = (d-min(min(d)))/max(max(d-min(min(d))))); %Normalize the image to the tallest peak
```

3a

Set up and
pad for
correlation

3b

Calculate the
correlation

3c

Scale between
0 and 1

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.

Table 6-3 Application Example Using Star-P

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 ").

Table 6-3 Application Example Using Star-P

Step	Description
5	The <code>ppfront</code> function (equivalent to the <code>pp2matlab</code> command) moves the data to the frontend or client for viewing. (see " Application Example Not Using Star-P ").

patmatch_color_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.

1
Image is
transferred
to back-end

```
% Setup the variables
imr = 1; imc = 1; %Set number of imag tiled in rows and columns
target = 'Twhite.JPG';
img = '500x500.JPG';
thres = 0.85;
% Load the data, comes in RGB, transfer to HSV soace
a = rgb2hsv(imread(img)); %Get the image containing targets
b = rgb2hsv(imread(target)); %Get the filter mask
% Transfer image data to the server
a = ppback(a,2);
% Setup the input image tiling problem
if imr > 1 | imc > 1
    a = repmat(a,imr,imc);
end
% Perform correlation calculation in HSV space with H, S, and V on different processors
d = zeros(size(a));
for i = 1:3
    d(:, :, i) = patmatch_calc(a(:, :, i), b(:, :, i));
end
e = (1-d(:, :, 2)) > 0.5 & d(:, :, 3) > thres; %Threshold finding target within input image
% Display the results
figure(1)
image(hsv2rgb(ppfront(a))); title('Input Image')
figure(2)
image(hsv2rgb(b)); title('Filter Image')
figure(3)
d = ppfront(d)
imagesc(d(:, :, 1)); title('Correlation, H') %StarP - transfer data to client to view
figure(4)
imagesc(1-d(:, :, 2)); title('Correlation, S') %StarP - calc on HPC, data to client to view
figure(5)
imagesc(d(:, :, 3)); title('Correlation, V') %StarP - transfer data to client to view
figure(6)
imagesc(ppfront(e)); colormap gray; title('Threshold Correlation') %StarP - data to client to view
```

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 this 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 there eliminated for brevity. The key difference in using the `patmatch_calc` function is the setup of `ppeval` that calls it.

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 <code>ppeval</code> 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.

NOTE: ppfront is equivalent to the pp2matlab command.

2
Tiling is not included. It limits performance gains.

5
Correlation calculations performed on three backend processors

```
% Setup the variables
target = 'Twhite.JPG';
img = '500x500.JPG';
thres = 0.85;
% Load the data, comes in RGB, transfer to HSV soace
a = rgb2hsv(imread(img));           %Get the image containing targets
b = rgb2hsv(imread(target));        %Get the filter mask
% Perform correlation calculation in HSV space with H, S, and V on different processors
d = ppeval('patmatch_calc',split(a,3),split(b,3));
e = (1-d(:,:,2))>0.5 & d(:,:,3)>thres; %Threshold finding target within input image
% Display the results
figure(1)
image(hsv2rgb(a));title('Input Image')
figure(2)
image(hsv2rgb(b));title('Filter Image')
figure(3)
d = ppfront(d)
imagesc(d(:,:,1));title('Correlation, H') %StarP - transfer data to client to view
figure(4)
imagesc(1-d(:,:,2));title('Correlation, S') %StarP - calc on HPC, data to client to view
figure(5)
imagesc(d(:,:,3));title('Correlation, V') %StarP - transfer data to client to view
figure(6)
imagesc(ppfront(e));colormap gray;title('Threshold Correlation') %StarP - data to client to view
```


7 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).

Sparse Matrices: Representing Graphs and General Data Analysis

There are so many examples -- some are discrete from the start, others are discretizations of continuous objects, but all are about connections.

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:end-1); k(1,1)=i(1,end);
>> r=rand(n,1);
>> l=rand(n,1);
>> A=sparse([i i],[j k],[r l]);
```

The next example illustrates a circular network with unsymmetric weights.

```
>> B=spones(A)
```

gives the network without weights, and

```
>> [i,j]=find(B)
```

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 $n \times 3$ 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.

```
>> 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=randn(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.)

```
>> G = spones(G);
>> [nr, nc] = size(G);
>> v = (1:nc*p)';
>> e = ones(nc*p,1);
>> rowcounts = G * e;
>> indexsums = G * v;
>> averageindex = indexsums ./ max(rowcounts, 1);
```

Since e is a column of all ones, the first matvec $G*e$ computes the number of nonzeros in each row of G . The second matvec $G*v$ 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 7-1 is an example of this technique applied to the graph created in Kernel 1 of the SSCA#2 benchmark.

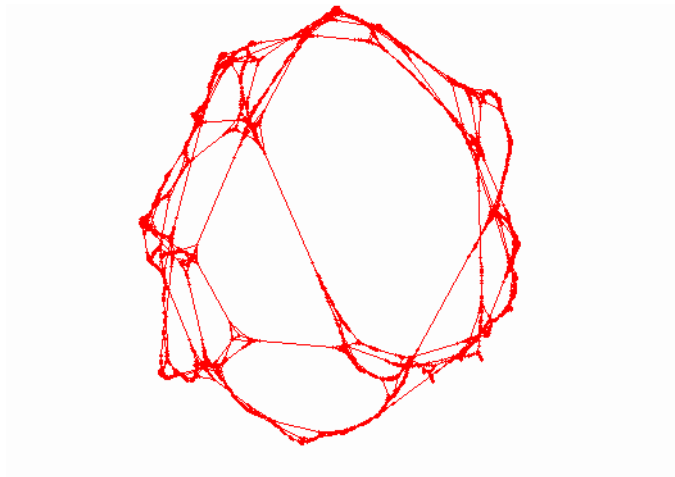


Figure 7-1 8192-vertex graph from Kern1 plotted with Fiedler coordinates

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:

$I_{ij} = \{1 \text{ if node } i \text{ is connected to node } j, \text{ otherwise } 0\}$

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(sprand(100*p,100,0.1))
a =
      dsparse object: 100p-by-100
>> b = a^3
b =
      ddense object: 100p-by-100
>> b(14,23)
ans =
      11
>> nnz(b)
ans =
```

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).

8 Advanced Topics

This chapter provides information for a variety of Star-P topics. It includes sections on the following:

- "Useful Commands to Monitor the Server"
- "Architecture"
- "Running Star-P in Batch"
- "Performance Infrastructure"
- "Star-P Software Development Kit"
- "Numerical Accuracy"
- "Computation of Eigenvectors for Non-Hermitian Matrices"

Useful Commands to Monitor the Server

While Star-P is designed to allow you to program at the level of the MATLAB / Star-P 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 on the server.

When the server is an SGI™ Altix™ system, the following commands will often be useful.

- **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. Because it's so easy to create big arrays with Star-P, it's easy to create arrays mistakenly that cause memory oversubscription, which can drastically reduce Star-P server and overall HPC server performance.
- **ps**: Perhaps obviously, 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 pids you need to kill the processes.

Architecture

The Star-P system is functionally composed of six modules as shown in Figure 4-1. The **Client** module runs on the client system, typically your own desktop system but sometimes a small server used specifically for Star-P clients. The **Administration Server** (shown in Figure 8.1 as the Master Control Module) can run either on the Server system or on a distinct system. (This latter approach matches the security constraints in place for some sites.) The remainder of the Star-P software, collectively known as the **Star-P server**, runs on the Server system.

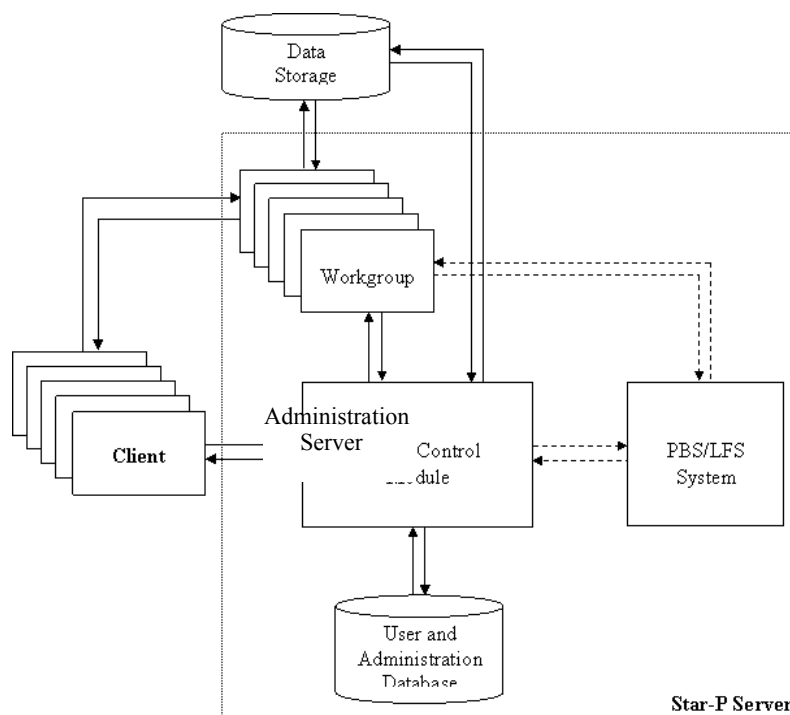


Figure 4-1 Star-P Architecture

These modules of the Star-P interactive platform have the following modules and interfaces as described below.

- Client

The client enables a user application to request a Star-P server to perform parallel computations on its behalf. The client consists of several components focused on enabling

the desktop application (MATLAB) which implements the actual user interface. Other client components include the `starpd` system daemon process, local files, garbage collection mechanism, and a browser. While initially the client supports MATLAB, a variety of additional interactive clients will be announced in future versions of Star-P.

- Administration Server

The Administration Server is the center of all system control activity in the Star-P server. It communicates with clients through an access portal, providing services such as including client authentication, session monitoring, data storage management, client configuration, HPC server administration, user profile management, client keep-alive services, and license management. The Administration Server interactively provisions workgroups when required by clients. The Administration Server, together with the workgroup, perform error management and error recovery.

- Workgroup

Workgroups are a server-provisioned collection of CPUs, memory, distributed application data, numerical libraries, manager and worker processes which form the receiving end of client compute requests. Typically the elements of a workgroup form a Message Passing Interface (MPI) world. Each workgroup contains the parallel numerical computation libraries (e.g., ScaLAPACK, FFTW, or PARPACK) called by the client application to perform calculations using the distributed data resident in workgroup memories.

- User and Administration Database

The Star-P server maintains a persistent store of state information within the User and Administration Database. Star-P uses a standard SQL relational database management system organized into tables. The database is used to store relationships between the following types of information.

- system users and their profiles
- client configuration and connection state
- installed numerical library capabilities of HPC server
- installation status and package version control information
- user session information
- machine CPU and memory configuration
- workgroup provisioning state and information

- session data location in storage system and checkpointing information

- software license status

- Data Storage

The data storage module offers a shared file system such as NFS or AFS that is visible to each process running in the HPC server. As workgroups are formed, the Administration Server controls transfer of distributed data from the data storage system into the memory of the new workgroup. Clients may subsequently request to attach to a prior session, at which time the distributed data is restored from data storage. († Implementation of this feature is deferred.)

Running Star-P in Batch

Release 2 of Star-P supports a limited mode of batch processing. Integrated support for popular batch workload management systems is planned for Release 3.

A Release 2 batch process is implemented from a dedicated MATLAB client. The process is as follows:

1. The user creates an interactive session on the Administration Server.
2. The user manually launches the Star-P client, MATLAB and the HPC Server as usual.
3. The user MATLAB program is placed into a main “batch loop” and run on the client.
4. The batch run proceeds to termination.
5. Data is saved on the HPC server using standard `ppsave/ppload` commands.
6. The user’s Star-P session is manually closed.

Performance Infrastructure

This section outlines Star-P infrastructure and associated methodology intended to measure, analyze and improve the performance of Star-P applications. Each of these activities is described in some detail below. Although some insight into the performance of Star-P could be gained through analysis of the code, timing data is fundamental to performance tuning. Timings need to be performed on a dedicated quiescent server system that can be re-used over time, so as to provide a history of data during the course of Star-P development.

Although various measurements are possible, the fundamental quantity measured will be the wall clock or total time, obtained via the following construct;

```
>> tic; function(); toc
```

These times can be further broken down into each of the following component times:

- Client

The time spent in the .m file on the client, before and after communication with the server.

- Communication with Server

The time spent on the TCP/IP calls. To be useful, the client server connection would need to be the same one across a series of tests.

- Server data management

Time spent inside the matrix manager and other utility functions up to the library.

- Server work function

Time spent in the library call doing the actual work.

The functions **pptic** and **pptoc** yield additional information specifically related to Star-P server performance

Star-P Software Development Kit

Introduction

The Star-P Software Development Kit (SDK) enables users to connect Star-P to their custom libraries/codes running on the server. The SDK serves as a glue that allows them to pass the data from the StarP runtime on the server to their custom codes, then collect the results and make them available to Star-P. The SDK is supported by a number of client-side commands that

enable the loading/unloading of the libraries integrated through the SDK and calling the functions exposed through it.

Prerequisites

In order to create functions using the SDK a user must possess:

- access to the server machine where Star-P is running
- ability to compile C++ on the server
- ability to make the resulting shared objects accessible to StarP server

In order to use the functions created through the SDK a user needs

- a MATLAB client
- knowledge of where the libraries built using the SDK reside and functions they expose

Programming Concepts

A function exposed through the SDK takes a number of inputs (accessible through `StarpSDK.get_input_XXX()` methods) and it either returns zero or more outputs (through `StarpSDK.add_output_XXX()`) or sets an error (`StarpSDK.set_error()`). The inputs and outputs can be distributed matrices, global matrices, scalars and strings. Apart from strings, all other parameter types can contain real (ELTYPE) or complex (CELTYPE) data.

Distributed matrices are made accessible to the user code through smart pointers to the `matrix_t` class objects. Each of these objects exposes the meta-data about the datatype and layout of the data in the matrix, as well the way the data is distributed among processors. `Matrix_t` objects also give user code access to raw data in the matrix. Finally, `matrix_factory_t` allows users to create new matrices in the StarP runtime.

SDK functions must be registered with Star-P runtime by adding a symbolic registering a symbolic name to each function. This name is used to call each function from the client. The registration is done by making a static `register_func()` call on `StarpSDK` object inside a special `starpInit()` routine, there has to be one of these present in each SDK-enabled package.

Invocation

Once the SDK function is written, compiled and the resulting shared object is made available to Star-P, it can be invoked from the client by first loading the shared object using the `pploadpackage()` MATLAB command and then calling `ppinvoke()` passing in the name of the function and the arguments it takes.

Numerical Accuracy

Star-P's underlying libraries are high quality numerical libraries. Arithmetical operations comply to the IEEE standards. You may want to understand the influence of small round-off errors on numerical results. It is well known that even addition is not associative in the presence of rounding errors. This means that $(a+b)+c$ and $a+(b+c)$ can yield answers that are different. While splitting computations across multiple processors can allow users to enjoy larger data sets and faster execution, it is also quite likely that operations will not perform in a deterministic order. Thus, the price that is paid is that answers may differ when run with different numbers of processors, as compared with a serial run. While round-off differences can lead theoretically to qualitatively different answers, the Star-P developers have taken great care that the parallel algorithms used are numerically stable, either by circumventing this issue or attempting to constrain parallel execution where this issue could have a high-level effect. Users who demand without compromise answers that are bit-wise the same as MATLAB may not find parallel execution beneficial.

Computation of Eigenvectors for Non-Hermitian Matrices

The current release of the Star-P server only supports the computation of eigenvectors of a hermitian matrix. To check if a distributed matrix A is hermitian, the eigenvalue solver computes the following:

$$\|A - A^H\|_F$$

This is quite expensive for large matrices. However, this test can be bypassed by explicitly passing the `sym` option to the solver. (See `help @ddense/eig`).

If a matrix is non-hermitian and `eig` is used to compute its eigenvalues and eigenvectors, the resulting eigenvalues will be correct, but the eigenvectors may be incorrect (they in fact correspond to the Schur reduced form of the matrix, which are the eigenvectors only if the matrix is hermitian).

To determine the eigenvector x_i of a matrix A corresponding to the eigenvalue λ_i , we first solve the system of equations:

$$(A - \lambda_i I) \tilde{x}_i = b$$

for a random right-hand side b where $\tilde{\lambda}_i$ is obtained from λ_i by slight perturbation (say $\lambda_i \pm \varepsilon$). We then obtain x_i as

$$x_i = \frac{\tilde{x}_i}{\|\tilde{x}_i\|_2}$$

The above procedure is quite effective in practice, provided only a *few* eigenvectors corresponding to *well-spaced* eigenvalues are desired.

Glossary

administration server	The machine that enables login to the Star-P system. It can run on the same hardware as the high performance compute (HPC) server but for security or resource allocation purposes some installations run the administration and HPC servers on different machines.
backend	The high performance “server” computer is informally denoted the backend. It is thought of as the “behind the scenes” hardware that accelerates programs and holds large data sets.
client	The Star-P client, informally, is the MATLAB application seen by the user. It may also denote the hardware this application runs on. More formally, this is the Star-P client hardware. The Star-P client software consists of the ISC proprietary software that, in addition to MATLAB, creates the entire interactive experience as seen from the frontend.
data parallel	A classification of a parallel computing language in which a single stream of instructions are applied simultaneously to the elements of a data structure such as an array.
frontend	The “client” computer in which the user is running the desktop application such as MATLAB.
global array syntax	A description of a computer language in which arrays are referred to as a single abstraction even if the data elements are distributed. ⁼²⁰ This is a more descriptive name for what many refer to as data parallelism.
high performance computer (HPC)	An HPC is a multiprocessor computer intended for the solution of larger problems or the reduction of run times.
parallelism through polymorphism	The ability to obtain parallel operations through the use of object oriented programming overloading operations so that the parallel operations are transparent to the user.
pool	See <i>Star-P processor pool</i>
propagation of parallelism	The ability of a software language to propagate the parallelism in the code through the use of syntax and semantics
server	The high performance computer where Star-P runs its high performance computations. It is synonymous with backend. See <i>administration server</i> .
Star-P processor pool	The available physical processors. This is the hardware that the administrator makes available for Star-P use. This may be an entire machine, multiple machines, or a fraction of a single machine.

task parallelism	Coarse-grained parallelism (sometimes called "embarrassingly parallel" or "task parallelism") is a powerful method to carry out many independent calculations in parallel, such as Monte Carlo simulations, or "un-rolling" serial FOR loops.
transparent parallelism	Serial code is said to run transparently on a parallel computer if no user changes are required to run the code.
workgroup, Star-P workgroup	The processors that a user has access to during a session. These processors are usually a subset of the processors in the interactive processor pool.

Index

Symbols

*p syntax, 3, 26

A

Administration Database, 111
Administration Server, 110, 111
architecture, 110
array bounds, 28
arrays, distributed, 77

B

block cyclic distribution, 77
block-cyclic distribution, 79

C

Client module, 110
client-server model, 2
column distribution, 78
combinatorial problems, 99
communication in Star-P, 65
 implicit, 66
 interprocessor, 75
complex numbers, 24

D

data creation routines, 26
data management, 8

data storage module, 112
ddense class, 40
dense arrays, distributed, 80
dense matrices, distributed, 77
distributed array bounds, 28
distributed arrays, creating, 25
distributed classes, 40
distributed data, 19
distributed data creation routines, 26
distributed dense arrays, 77, 80
distributed dense matrices, 77
distributed matrices, examining, 47
distributed partitioned objects (dpart), 83
distributed sparse matrices, 81
dlayout class, 42

F

features, Star-P, 6
Fourier analysis, 89

H

hardware requirements, 9
Hilbert matrix, 47

I

image processing algorithm, 89
indexing, 28

L

Laplacian matrix, 105
library management, 6, 8
Linux client system, startup, 13
loading data, 31

M

MATLAB and Star-P, 3
MATLAB features, 54
matlab2pp function, 31, 36

N

naming conventions, 19
np function, 21

P

p function, 21
path counting, 106
performance monitoring, 69
pivot tables, 102
pp2matlab function, 31, 48
ppeval application example, 96
ppeval function, 48, 96
ppload function, 31
ppprofile function, 71
ppsave function, 31
pptic function, 69
pptoc function, 69
propagation, distributed attribute, 43

R

requirements, Star-P system, 9
row distribution, 78

S

serial vectorization, 61
sparse matrices, 99
sparse matrices, distributed, 81
sparse matrices, representation in Star-P, 81
Star-P
 architecture, 110
 configuration, 17
 documentation support, 9
 naming conventions, 19
 server, 110
 system requirements, 9
 whose command, 20
startup
 Linux client system, 13
 Windows client system, 14
system requirements, 9

V

vectorization
 serial, 61

W

whose command, 20
Windows client system, startup, 14
workgroups, 111