Star-P® User Guide

10/25/06

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
011	August 2006 (R2.4.0) Original publication

INTERACTIVE supercomputing

Contents

1 Star-P User Guide Overview 1

Introduction to Star-P 2

Star-P and MATLAB 3

Star-P Features 7

Interactive Computing Management 7

Library Management 9

Data Management 9

System Requirements 10

Hardware and Software 10

Acknowledgments 11

2 Quick Start Guide to Star-P 15

Starting Star-P on a Linux Client System 15

Starting Star-P on a Windows Client System 16

Star-P Sample Session 17

User-Specific Star-P Configuration 19

3 Using Star-P 21

Star-P Naming Conventions 21

Examining Star-P Data 22

Special Variables: p and np 23

Assignments to p 25

Complex Data 26

Creating Distributed Arrays 27

The ppload and ppsave Star-P Commands 33

Writing variables to a HDF5 file 34

Reading variables from a HDF5 file 35

Querying variables stored inside a HDF5 file 35

Multidimensional arrays 36

Complex data 36

Sparse matrices 36

Explicit Data Movement with ppback and ppfront 38

Combining Data Distribution Mechanisms 40
Mixing Local and Distributed Data 41
Distributed Classes used by Star-P 42
Propagating the Distributed Attribute 45
Reusing Existing Scripts 49
Examining/Changing Distributed Matrices 49
The ppeval and ppevalc Functions: The Mechanism for Task Parallelism 50
Input Arguments 51
Output Arguments 52
Difference Between ppeval/ppevalc and ppevalsplit/ppevalcsplit 59
How to use create_starp_library 59
Example 60
It's Still MATLAB: Using MATLAB Features on Parallel Programs 60
Configuring Star-P for High Performance 63

4 Structuring Your Code for High Performance and Parallelism 67
Performance and Productivity 67
Parallel Computing 101 68
Vectorization 70
Star-P Solves the Breakdown of Serial Vectorization 73
Solving Large Problems: Memory Issues 75

5 Data Organization in Star-P 77
Communication between the Star-P Client and Server 77
Implicit Communication 78
Communication Among the Processors in the Parallel Server 80
Types of Distributions 82
Row distribution 83
Column distribution 83
Block-cyclic distribution 84
How Star-P Represents Sparse Matrices 86
Distributed Cell Objects (dcell) 88
Propagation of Distribution 88
Functions of One Argument 89
Functions of Multiple Arguments 89
Indexing Operations 91

Summary for Propagation of Distribution 93

6 Application Examples 95
Application Example: Image Processing Algorithm 95
How the Analysis Is Done 95
Application Examples 95
patmatch_color_noStarP.m File 99
patmatch_calc.m 100
patmatch_color_StarP.m File 101
About ppeval 102
patmatch_color_ppeval.m 103

7 Solving Large Sparse Matrix and Combinatorial Problems with Star-P 105
Graphs and Sparse Matrices 105
Graphs: It's all in the connections 105
On Path Counting 112

8 Advanced Topics 115
Useful Commands to Monitor the Server 115
Architecture 116
Running Star-P in Batch 117
Example 118
Performance Infrastructure 119
Star-P Software Development Kit 119
Introduction 119
Prerequisites 120
Programming Concepts 120
Invocation 121
Numerical Accuracy 121
Computation of Eigenvectors for Non-Hermitian Matrices 121
Diagnostics and Performance 122
Client/Server Performance Monitoring 123
ppeval_tic/toc: 129

9 Supported MATLAB® Functions . . . 133
Overloaded MATLAB Functions, Alphabetically . . . 134
Supported MATLAB Functions, by Function Class . . . 144
MATLAB datafun function class . . . 144
MATLAB datatypes function class . . . 145
MATLAB elfun function class . . . 146
MATLAB elmat function class . . . 148
MATLAB iofun function class . . . 150
MATLAB lang function class . . . 151
MATLAB matfun function class . . . 152
MATLAB ops function class . . . 153
MATLAB sparfun function class . . . 154
MATLAB specfun function class . . . 155
MATLAB strfun function class . . . 156

10 Star-P Functions . . . 157
Basic Server Functions Summary . . . 157
Known Differences Between MATLAB and Octave Functions . . . 160
General Functions . . . 160
np . . . 160
p . . . 160
pp . . . 160
ppgetoption . . . 160
ppsetoption . . . 161
ppgetlog . . . 161
ppinvoke . . . 161
pploadpackage . . . 162
ppunloadpackage . . . 162
ppfopen . . . 162
ppquit . . . 163
ppwhos . . . 163
pph5whos . . . 163
Data Movement Functions . . . 164
ppback . . . 164
ppfront . . . 165
ppchangedist . . . 166
pph5write . . . 166

pph5read 167
ppload 168
ppsave 168
Task Parallel Functions 169
bcast 169
split 170
ppeval 170
ppevalc 171
Example 171
ppevalc Function Classes 172
pearg_t class functions 172
Public Functions 173
Returns 174
ppevalc_module_t Function Class 176
Member Functions 176
Example 177
ppevalsplit 178
ppevalcsplit 178
ppevalcloadmodule 179
ppevalcunloadmodule 179
ppprofile 179
Example 180
pptic/toc 180

11 Star-P User Commands 183
Examples 184

12 Additional Information for ppevalc 187
ppevalc Application Programming Interface (API) 187
pearg_t Function Class 187
Input Arguments 187
Public Functions 187
Returns 188
ppevalc_module_t Function Class 190
Public Member Functions: Input 190
Public Member Functions: Return 190

13 Glossary **197**
administration server 197
backend 197
client 197
data parallel 197
frontend 197
global array syntax 197
high performance computer (HPC) 197
parallelism through polymorphism 197
pool 197
propagation of parallelism 197
server 197
Star-P processor pool 197
task parallelism 198
transparent parallelism 198
workgroup, Star-P workgroup 198

Index **199**

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 8.300713 seconds.
  Name        Size                       Bytes  Class
  a          10x10                        1472  ddense object
  ans         1x1                            8  double array
  b          10x10                        1486  ddense object
  bb        100x100                       1472  ddense object
  c          36x1                         1478  ddense object
  i          10x10                        1476  ddense object
  q       10000x16                        1472  ddense object
  r          16x1                          128  double array
  x        2000x2000                  32000000  double array
  xinv     2000x2000                  32000000  double array
  xo       2000x2000                  32000000  double array

Grand total is 12000203 elements using 96008992 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(2000,2000*p);
 >> xinv=inv(x);
 >> xo=x*xinv;
 >> max(max(xo))
 >> toc
ans =
    1.0000
Elapsed time is 3.420039 seconds.
Your variables are:
  Name       Size            Bytes         Class
  a          10px10          800           ddense array
  ans        1x1             8             double array
  b          10x10p          800           ddense array
  bb         100x100p        80000         ddense array
  c          36px1           288           ddense array
  i          10x10p          800           ddense array
  q          10000px16       1280000       ddense array
  r          16x1            128           double array
  x          2000x2000p      32000000      ddense array
  xinv       2000px2000p     32000000      ddense array
  xo         2000x2000p      32000000      ddense array

Grand total is 12170353 elements using 97362824 bytes
MATLAB has a total of 17 elements using 136 bytes
Star-P server has a total of 12170336 elements using
97362688 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

  On the SGI Altix platform, 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.
- "Advanced Topics", provides information for a variety of Star-P topics.
- "Supported MATLAB® Functions", lists the MATLAB functions that are supported with Star-P.
- "Star-P Functions", summarizes the Star-P functions that are not part of standard MATLAB and describes their implementation.
- "Star-P User Commands", provides users with options that allows them to override some of the defaults to control Star-P behavior more precisely.

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 one every processors, and the Mathworks Distributed Processing Toolbox allows users to message pass among MATLABs on multiple processors.

2 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 =
     6
```

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 =
    0.2998
```

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 =
   8.0512e-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 programing.

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

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          1000x1000p       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 =
     6
Your variables are:
  Name       Size             Bytes        Class
  aa         3464x3464p       95994368     ddense array
  ans        1x1              8            double array
```

```
  n          1x1             8             double array

Grand total is 11999298 elements using 95994384 bytes
MATLAB has a total of 2 elements using 16 bytes
Star-P server has a total of 11999296 elements using
95994368 bytes
```

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

```
  Name       Size                      Bytes  Class
  aa      3464x3464                     1472  ddense object
  ans        1x1                           8  double array
  n          1x1                           8  double array

Grand total is 31 elements using 1488 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,
ans =
     1p
ans =
     6
z =
    9.8596
p =
    3.1400
Your variables are:
  Name       Size          Bytes      Class
  a          100x100p      80000      ddense array
  ans        1x1           8          double 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 240032 bytes
MATLAB has a total of 4 elements using 32 bytes
Star-P server has a total of 30000 elements using 240000
bytes
>> clear p ;
>> p,
ans =
     1p
Your variables are:
  Name       Size          Bytes      Class
  a          100x100p      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 =
        ddense object: 1000p-by-1000
c =
        ddense object: 1000p-by-1000
cc =
        ddense object: 1000p-by-1000
d =
        ddense object: 1000p-by-1000
e =
        ddense object: 1000p-by-1000
f =
        ddense object: 1000p-by-1000
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
Grand total is 7000001 elements using 80000008 bytes
MATLAB has a total of 1 elements using 8 bytes
Star-P server has a total of 7000000 elements using 80000000 bytes
```

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

```
 >> a = rand(1000,1000*p)
ans =
     1
ans =
     0
ans =
     0
ans =
```

```
        1
a =
        ddense object: 1000-by-1000p
b =
        ddense object: 1000-by-1000p
Your variables are:
  Name       Size           Bytes        Class
  a          1000x1000p     8000000      ddense array
  b          1000x1000p     16000000     ddense array (complex)

Grand total is 2000000 elements using 24000000 bytes
MATLAB has a total of 0 elements using 0 bytes
Star-P server has a total of 2000000 elements using 24000000 bytes
```

Creating Distributed Arrays

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

- The data can be initially allocated as distributed using the *p syntax in conjunction with a variety of constructor routines such as **zeros**, **ones**, **rand**, **randn**, **spones**, **sprand**, 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 create a matrix that is distributed in the second dimension. For more detail, see "Types of Distributions".

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

```
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 =
     21     21p
ans =
     21     21p
```

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
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: 1000-by-1000p
```

Indexing into Distributed Matrices or Arrays

Indexing allows creation of new matrices or arrays from subsections of existing matrices or arrays. Indexing on distributed matrices or arrays always creates a distributed object, unless the result is a scalar, in which case it is created as a local object. Consider the following example:

```
>> 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           1000x1000p     8000000      ddense array
  ans         1x1            8            double array
  b           1000x1000p     8000000      ddense array
  c           1000x1000p     8000000      ddense array
  d           6x767p         36816        ddense array
  e           1x1            8            double array
  f           1000000px1     8000000      ddense array

Grand total is 4004604 elements using 32036832 bytes
MATLAB has a total of 2 elements using 16 bytes
Star-P server has a total of 4004602 elements using 32036816 bytes
```

In order to propagate the distribution of data as broadly as possible, Star-P interprets indexing operations on distributed objects as creating new distributed objects, hence the distributed nature of `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: 1000-by-1000p
 >> ppwhos
Your variables are:
   Name        Size             Bytes        Class
   A           1000x1000p      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: 100-by-100p
Client/server communication info:
```

```
    Send msgs/bytes    Recv msgs/bytes     Time spent
  0e+00 / 0.000e+00B  3e+00 / 8.039e+04B   1.355e-02s
Server info:
  execution time on server: 0.000e+00s
  #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: 100-by-100p
 >> a = bb * pi
a =
         ddense object: 100-by-100p
```

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

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

```
 >> 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          1000px1000       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/elmat/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: 1000-by-1000p
 >> [rows cols] = size(a)
rows =
      1000
cols =
      1000p
 >> ppload imagedata A
 >> B = inv(A)
```

```
B =
         ddense object: 1000-by-1000p
 >> [Brows Bcols] = size(B)
Brows =
      1000
Bcols =
      1000p
 >> ppwhos
Your variables are:
   Name       Size              Bytes      Class
   A          1000x1000p        8000000    ddense array
   B          1000x1000p        8000000    ddense array
   Bcols      1x1               258        dlayout array
   Brows      1x1               8          dlayout array
   a          1000x1000p        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          100x100p      80000     ddense array
   b          100x100p      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:

```
%function H = hilb(n)
%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 and ppevalc Functions: 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** or **ppevalc** functions allow 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).

ppeval and ppevalc

The only difference between `ppeval` and `ppevalc` is what language you want to use to write your function that runs in parallel.

- **`ppeval`** is an octave function. Octave is a language that is used for performing numerical computations. It is compatible with MATLAB. With `ppeval`, you can write a function on the client side which is then sent over to the server. ppeval specifies MATLAB functions.
- **`ppevalc`** is a C++ function. This allows you to write the function that runs in parallel in C++ rather than Octave. The main motivation for using `ppevalc` is if you have existing serial libraries in C and C++ that you want to use, it makes it easy to write wrapper functions in C++ using the ppevalc API. With `ppevalc`, you need to build the function on the server or a smaller machine and copy the module over to the server machines. ppevalc specifies C++ functions.

About ppeval

`ppeval` allows you to write a function on the client side and pass it over to the server.

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
```

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`**.

```
>> 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 =
        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 Guide**[2], 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.

About ppevalc

ppevalc is simple to use, since it has only two wrapper classes:

- **`pearg_t`** - A class to hold input and output arguments to ppevalc functions. It only supports three types of values or element types:
  - — **char** -which represents a string
  - — **double** - which means it can be an array of doubles
  - — **double complex - which** a complex double constructs complex data from real and imaginary data.
- **`ppevalc_module_t`** - This class provides the interface for ppevalc modules to interact with the starpserver runtime.

Basically, it is just a matter of getting the input arguments to your functions and passing your arguments back. You simply build your module and load it on the server.

The semantics of `ppevalc` are that the input array(s) are:

- by default, partitioned on the last dimension (i.e., columns for 2D matrices) if you do not specify anything, or
- split along any dimension as long as all arguments once it is split have the same number of rows and columns.

  For example, for a 2D matrix, you can choose to split it into rows rather than columns. All arguments must be split into the same number of slices.

The function is called on each column or each row of the input, in parallel. When you pass input arguments, you can split them in a number of ways:

- along any dimension, including along the last dimension,
- among elements where you can pass every element of an array into an individual function call,
- along broadcast arguments where the entire argument is passed along to each function invocation. In this case, at least one of the other arguments must be something other than broadcast to establish how many slices there are.

  For example, you can have one broadcast argument and one element split argument. If you were doing an element split on 1000 elements, the function would be invoked 1000 times but the broadcast argument would be passed in its entirety to each one of the invocations.

The output arguments will have the same size as the number of slices that the input argument was split into.

You can view **ppevalc** 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.

An include directory and a makefile skeleton that a user can adapt to their own module are included on the Star-P Installation CD.

Example

Let's consider a simple example.

The first example is a ppeval function:

```
ppeval c('module_function, a, b')
```

This corresponds to a C++ function that you write on the server. This `ppevalc` function for C++ has the following format:

```
void function(ppevalc_module_t &module, inargs, outargs)
```

What occurs in this C++ example is:

1. Process the input arguments.
2. The computation is done in C++.
3. Put the output into outargs (output arguments).

This C++ example corresponds to writing an .m file in MATLAB with input arguments similar to the `varg in` and the output arguments similar to `varg out`. The C++ module object gives you additional functionality. The `ppevalc` module consists of

one or more functions, similar to the example, where you write and build on the server.

`ppevalc` is useful for users with existing libraries or algorithms written in compiled languages, or, is good for functions that would take less time to write if it is done in a compiled language.

Function Example

In the following example, a cumulative sum function is implemented. This demonstrates how you can

- pass input argument data directly to an external function, and
- have the external function write directly into an output argument without making extra copies of the data.

For output arguments, this requires that the external function support being told where to write its result. In this example, the C++ standard library's partial sum function is used.

```
static void cumsum(ppevalc_module_t& module,
                   pearg_vector_t const& inargs, pearg_vector_t& outargs)
{
  // check input and output arguments
  if (inargs.size() != 1) {
    throw ppevalc_exception_t("Expected 1 input argument");
  }
  if (!inargs[0].is_vector()) {
    throw ppevalc_exception_t("Expected input argument to be a vector");
  }
  if (inargs[0].element_type() != pearg_t::DOUBLE) {
    throw ppevalc_exception_t("cumsum only supports arrays of doubles");
  }

  if (outargs.size() != 1) {
    throw ppevalc_exception_t("Expected 1 output argument");
  }

  // create an output argument of the same type and shape as the input argument
  pearg_t outarg(inargs[0].element_type(), inargs[0].size_vector());

  // call external function, telling to read its input args directly
  // from inargs, and telling it to write its result directly into the
  // outarg

  starp_double_t const* indata = inargs[0].data<starp_double_t>();
  starp_double_t *outdata = outarg.data<starp_double_t>();

  std::partial_sum(indata, indata + inargs[0].number_of_elements(),
```

```
                    outdata);

  outargs[0] = outarg;
}
```

Difference Between ppeval/ppevalc and ppevalsplit/ppevalcsplit

`ppeval/ppevalc` and `ppevalsplit/ppevalcsplit` take care of data and distribution management.

- If the data is local, Star-P will distribute.
- If the data is distributed, the type of distribution does not matter.

The result always resides on the server:

- if the output of each task has the same size, ppeval/ppevalc will collect the results, laminate them together, and distrubute the result along the highest dimension.
- if the tasks do not produce outputs of the same size use ppevalsplit/ppevalcsplit.

`ppevalsplit/ppevalcsplit` produces output of different sizes.

So a quick summary is that for both `ppeval/ppevalc` and `ppevalsplit/ppevalcsplit`, the following is true:

- `args` must be split with:
  - elements as input
  - rows as input
  - columns as input
  - Nth dimension as input
- or, `args` can be broadcast

args must be scalar, matrix (front-end to back-end), and function handles (broadcast only).

The main difference is that `ppevalsplit/ppevalcsplit` returns a dcell object, a cell array of return values from each iteration.

How to use create_starp_library

`create_starp_library` is a shell script that builds shared libraries from source files written using the ppevalc API. The shared libraries created by `generate_starp_library` can be loaded into Star-P and accessed via ppevalc function calls. `generate_starp_library` has the following usage:

```
create_starp_library [options] source1 [source2] ...
options:
 --version            show program's version number and exit
 -h, --help           show this help message and exit
 -v, --verbose        Display commands as they are executed
 -c, --compile        Compile source files, but do not link
 -lLDFLAGS            Adds library after -l to link command
 -LLDDIRS             Adds directories after -L to link command
 -ICPPFLAGS           Adds directories after -I to include command
 -mMODULENAME         Moduleodule name
 -XEXTRACOMPILEFLAGS  Extra compile flags

 -YEXTRALINKFLAGS     Extra link flags
```

`create_starp_library` is located in the bin directory under the root of the Star-P installation. Please note the following:

- `create_starp_library` requires at least one source file as input.
- If the `-m` option is not used, the name of the first source file used as the module name (the extension will be removed).
- Shared libraries are generated in the directory in which the script is invoked.

Example

```
/usr/local/starp/2.X.X/bin/create_starp_library /home/
username/src/foo.cc
```

produces `/home/username/src/libfoo.so`

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 and ppevalc Functions: 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 Internet[1] 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 x 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 MATLAB[1] and Star-P will run much more slowly when their working set exceeds the size of physical memory, though Star-P has the advantage that the size of physical memory will be bigger.

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.

INTERACTIVE supercomputing

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, excessive data transfer between the client and server can cause performance degradation, and thus the default behavior for Star-P is to leave large data on the server. One typical programming style is to move any needed data to the server at the beginning of the program (via **ppback**, **ppload**, etc.), operate on it repeatedly on the server, then save any necessary results at the end of the program (via **ppfront** (the deprecated `matlab2pp` command is equivalent to `ppfront`), **ppsave**, etc.).

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

```
 >> load imagedata a     %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          100x100p       80000      ddense array

Grand total is 20000 elements using 160000 bytes
MATLAB has a total of 10000 elements using 80000 bytes
Star-P server has a total of 10000 elements using 80000 bytes
```

The **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          100x100p       80000      ddense array
   bb          100x100p       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
```

When accessing data from disk, it may be faster to load it directly as distributed array(s) rather than loading it into the client and then moving it via **ppback** (and similarly to save it directly as distributed arrays). The **ppload/ppsave** commands are the distributed versions of the **load/save** commands. For information on **ppload** and **ppsave**, see "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 and minimize 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          100x100p       80000      ddense array
```

```
  d          1x1          8          double array
  e          1x1          8          double array
```

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

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

```
 >> f = rand(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, which often contain hundreds of millions to trillions of elements, would not be useful. Thus the Star-P behavior for a command

lacking a trailing semicolon is to print out the size of the resulting object.

```
 >> 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** (equivalent to the deprecated `pp2matlab` command) and leave off the final semicolon, MATLAB will print out the whole contents of the array.

Note that communication can happen implicitly as described in "Mixing Local and Distributed Data".

Communication Among the Processors in the Parallel Server

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

Some operations can be accomplished with no interprocessor communication on the server. For instance, if two arrays are created with the same layout (see details of layouts in "Types of

Distributions"), element-wise operators can be done with no communication, as shown in the following example.

```
>> 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.

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

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

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

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 8 processors, the first 10,000/8 == 400 rows would be on the first processor, the next 400 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.

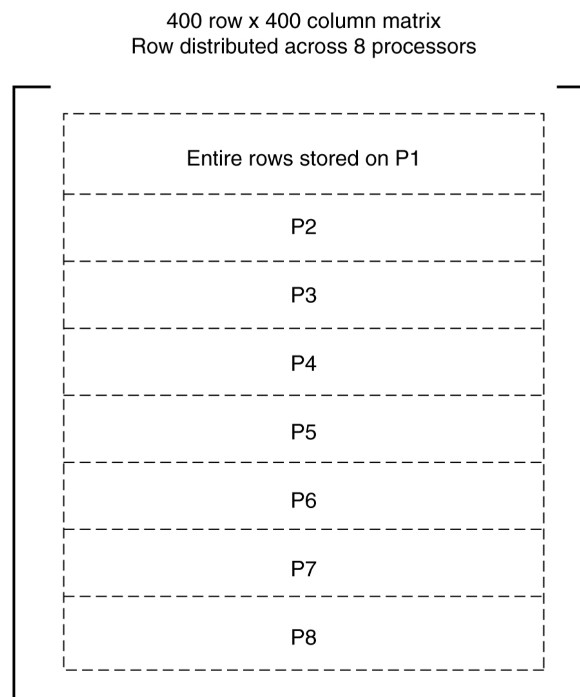


Figure 5-1 Row Distribution

Column distribution

Column-distributed works just the same as row distribution, except columns data is split over processors; `bb` is created that way above. Figure 5-2 illustrates the layout of a column distributed array. When a *p is placed in more than one dimension, the matrix or multi-dimensional array will be

distributed in the rightmost dimension containing a *p. For example, if there was a *p in both dimensions of the constructor for a two dimensional matrix, it would result in a column distribution.

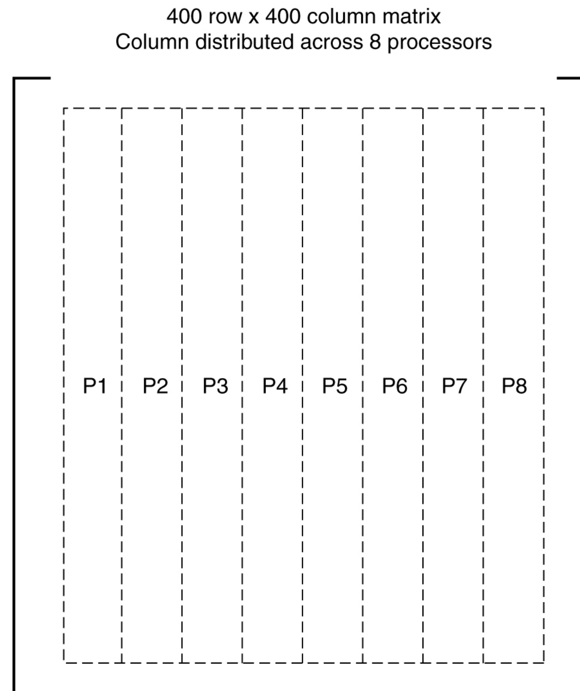


Figure 5-2 Column Distribution

Block-cyclic distribution

Though there are no explicit constructors for it, Star-P also supports a third style of distribution for two dimensional matrices. The block-cyclic distribution is used internally for certain operations, such as when calls are made to ScaLAPACK library functions. It is never necessary to create block-cyclic matrices, however, a small amount of performance improvement can result from explicitly switching to a block-cyclic distribution if many calls will be made to the `lu` or `\` functions. Block-cyclic matrices can be created by running `ppchangedist(A,3)`, where `A` is an existing distributed matrix. For information on `ppchangedist`, see "ppchangedist" on page 166.

A good example of a reason for using block-cyclic distributions can be seen by considering 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

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. 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, the resulting array will be distributed on the rightmost non-singleton requested dimension. A singleton is defined as a size of dimension is 1.

```
>> a = zeros(10*p,10,10*p,10*p,10*p)
a =
        ddensend object:10-by-10-by-10-by-10-by-10p
```

Multidimensional distributed dense arrays support a subset of operators on 2D arrays. See the full list in **Star-P Command Reference Guide**.

Distributed Sparse Matrices

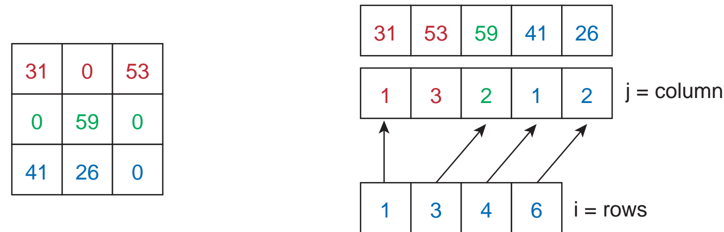
Distributed sparse matrices in Star-P use the compressed sparse row format. This format represents the nonzeros in each row by the index (in the row) of the nonzero and the value of the nonzero, as well as one per-row entry in the per-matrix data structure. This format consumes storage proportional to 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:

Star-P sparse data structure



Full:
- 2-dimensional array of real or complex numbers
- (nrows *ncols) memory

Sparse:
- compressed row storage
- about (2*nzs + nrows) memory

Figure 5-3 Star-P Sparse Data Structure

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

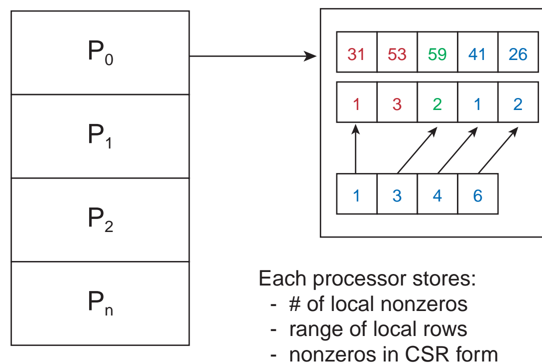


Figure 5-4 Star-P Distributed Sparse Data Structures

Figure 5-4 shows what happens when the sparse data structure from Figure 5-3 is distributed across multiple processors by Star-P. The number of rows is divided among the participating processors, and each processor gets the information about its local rows. Thus operations that occur on rows can happen

locally on a processor; operations that occur on columns require communication among the processors.

Distributed Cell Objects (dcell)

The dcell is analogous to MATLAB cells. The `dcell` type is different from the other distributed matrix or array types, as it may not have the same number of data elements per `dcell` iteration and hence doesn't have the same degree of regularity as the other distributions. This enables dcells to be used as return arguments for `ppevalsplit ()`. For more information on ppevalsplit, see "ppevalsplit" on page 178.

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

NOTE: Performance and floating point accuracy may be affected, see "Advanced Topics" on page 115 for more information.

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

Type	Distribution
ddense	row, column, block cyclic (deprecated)
ddensend	linear distribution along any dimension
dsparse	row distribution only

The distributions of the output of operations follow the "calculus of distribution". To calculate the expected distribution of the output of a given function, express the size of the output in terms of the size of the inputs. Note that matrices and multidimensional arrays are never distributed along

singleton dimensions (dimensions with a size of one), unless explicitly created that way.

Functions of One Argument

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

Examples for Functions with One Argument

Operations with one input:

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

```
>> 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:

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

As another example, consider transpose:

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

Exceptions:

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

Functions of Multiple Arguments

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

For operations for which the output size is the same as both inputs, such as elementwise operations (`A+B`, `A.*B`, `A./B`, etc), we consider the distribution of both inputs. If both inputs are

row distributed, then the output will be row distributed. If the combination of inputs has more than one distributed dimension, then the default of distributing on the rightmost dimension applies.

For example, with matrix multiplication,

```
C = A * B
size(C) = [size(A,1) size(B,2)] = rows_of_A -by- cols_of_B
```

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

For `C=A*B`, if `A` and `B` are both column distributed, similar logic forces the output to be column distributed.

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

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

Examples for Functions with Multiple Arguments

As a less trivial example, consider `C = 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 would be ambiguous, so it defaults to the standard of distributing the rightmost dimension:

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

As another example, consider transpose:

```
C = A'
size(C) = [size(A,2) size(A,1)]
```

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

Exceptions for Multiple Arguments

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

Table 5-1 Single ddense arguments (irregular cases)

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

Indexing Operations

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

Examples for Indexing

Some indexing examples:

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: 10-by-10p
>> B = A(I)
B =
ddense object: 10-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, `size(B)` is defined as `[prod([1 100p])prod([1 100])] [100p 100]` which simplifies to `[prod([10p 10p]) prod([10 10])]` and then [100p 100]. So the distribution of

- 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(find(I)), and find(I) returns a row-distributed column vector.

Summary for Propagation of Distribution

To summarize:

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

6 Application Examples

Application Example: Image Processing Algorithm

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

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.

How the Analysis Is Done

The analysis in this simplified application takes the transform of the input and target images, multiplies the elements of the transforms, and then transforms the product back. This results in correlation peaks located where the target image is located within the input image. Since the image is in color, the processing is performed within three different color spaces, correlation matches occur three times. Strong peaks exist in the image along with the possibility of some noise. To further data reduce the image, a threshold is used which reduces the information to a two dimensional (2D) binary map. The image of the 2D binary map reduces the three color space images into a single binary map indicating the locations of the target image. The location of the ones (1) indicate the position of the target image within the input image.In this example, the ones exist in four separate clusters and the centroid of each cluster indicates the center location of the target image.

Application Examples

There are three application examples given in this section:

- An example not using Star-P, see "Application Example Not Using Star-P".
- An example using `*p` to distribute the computation, see "Application Example Using Star-P".
- An example using `ppeval` to distribute the computation see "Application Example Using ppeval".

Images For Application Examples

The images used for the examples are shown in the figures below.

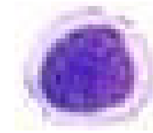


Figure 6-1 Target Image

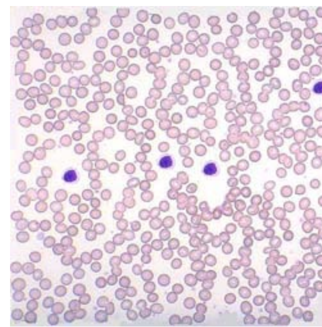


Figure 6-2 Input Image

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 `patchmatch_calc.m` file is called. A pattern matching calculation is used. This particular function is called for each of the three HSV images. A. The function correlates the input and target image by padding the target image, which is assumed to be a smaller image. It is padded with bright regions or ones (1). — 1 represents background — 0 represents lack of background Basically, the size input image is found and then the target image is padded to that size. The padded image is shifted into Fourier space assuring accuracy. B. The actual correlation is done by multiplying the Fourier transform input image times the complex conjugate of the target image. Next it takes the inverse transform of the two images and that creates the amplitude and phase of the correlation. To create the observable image, this product image is multiplied by its complex conjugate completing the correlation. C. Once the correlation calculation is complete, the correlation image is scaled between zero and one for each of the HSV components.
4	A threshold operation is performed to find the target locations within the input image. The operation is performed for each of the HSV components and is done empirically to achieve the display result through a map of the input image.
5	Displays the results which is a fully reduced, binary map of the target image location.

patmatch_color_noStarP.m File

The following is the sample file that contains the program code for the application example. The numbers on the left correspond to the table in the previous section.

1 Read in RGB data, convert to HSV

2 Image is tiled and replicated

3 Calculate the correlation on each of HSV components

4 Perform the threshold

5 Display the results

```
% Setup the variables
imr = 1; imc = 1;                         %Set number of image tiled in rows and columns
target = 'Twhite.JPG';
img = '500x500.JPG';
thres = 0.85;
% Load the data, comes in RGB, transfer to HSV 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
% 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(a));title('Input Image')
figure(2)
image(hsv2rgb(b));title('Filter Image')
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')
```

patmatch_calc.m

This is the contents of the calculation file that is called by `patmatch_color_noStarP.m`, `patmatch_color_StarP.m`, and patmatch_color_ppeval.m.

3a Set up and pad for correlation

3b Calculate the correlation

3c Scale between 0 and 1

```
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
```

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 ppfront function (equivalent to the pp2matlab 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

5 Image is transferred from 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
```

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 `ppeval` because tiling to larger images or working with larger input images on a single processor limits the performance gains achieved by single processor calculation. In other words, single processor calculations provide performance on small data sizes.
3	The correlation calculation as described for the previous two examples is performed on an individual processor on the backend.
4	The operation is the same as described for the previous two examples.
5	The operation is the same as described for the previous two examples.

patmatch_color_ppeval.m

The following is the sample file that contains the program code for the application example. The numbers on the left correspond to the table in the previous section. Only the differences from the "Application Example Not Using Star-P" are described.

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 manmade problem requiring a purchase of electrical components.

The internet is a great source for graphs. We could have started with any communications network: telegraphs, telephones, smoke signals... but let us consider the internet. The internet can be thought of as the physical links between computers. The current internet is composed of various subnetworks of connected computers that are connected at various peering points. Run traceroute from your machine to another machine and take a walk along the edges of this graph.

More exciting than the hardware connections are the virtual links. Any web page is a node; hyperlinks take us from one node to another. Web pages live on real hardware, but there is no obvious relationship between the hyperlinks connecting web pages and the wires connecting computers.

The graph that intrigues us all is the social graph: in its simplest form, the nodes are people. Two people are connected if they know each other.

A graph may be a discretization of a continuous structure. Think of the graph whose vertices are all the USGS benchmarks in North America, with edges joining neighboring benchmarks. This graph is a *mesh*: its vertices have coordinates in Euclidean space, and the discrete graph approximates the continuous surface of the continent. Finite element meshes are the key to solving partial differential equations on (finite) computers.

Graphs can represent computations. Compilers use graphs whose vertices are basic blocks of code to optimize computations in loops. The heart of a finite element computation might be the sparse matrix-vector multiplication in an iterative linear solver; the pattern of data dependency in the multiplication is the graph of the mesh.

Oftentimes graphs come with labels on their edges (representing length, resistance, cost) or vertices (name, location, cost).

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

Sparse Matrices: Representing Graphs and General Data Analysis

Consider putting everybody at a party in a circle holding hands and each person rates how well they know the person on the left and right with a number from 1 to 10.

Each person can be represented with the index i, and the rating of the person on the right can be listed as $A_{i,i+1}$ while the person on the left is listed as $A_{i,i-1}$.

As an example;

PERSON	Right	Left
1	5	6
2	3	2
3	1	9
4	2	7

In serial MATLAB

```
>> i=[1 2 3 4]; j=[2 3 4 1]; k=[4 1 2 3];
>> r=[5 3 1 2];
>> l=[6 2 9 7];
>> sparse([i i],[j k],[r l])
ans =
   (2,1)        2
   (4,1)        2
   (1,2)        5
   (3,2)        9
   (2,3)        3
   (4,3)        7
   (1,4)        6
   (3,4)        1

>> full(ans)
ans =
     0     5     0     6
     2     0     3     0
     0     9     0     1
     2     0     7     0
```

With Star-P

```
>> n=1000*p;
>> i=1:n;
>> j=ones(1,n);
>> j(1,1:end-1)=i(1,2:end); j(1,end)=i(1,1);
>> k=ones(1,n)
```

```
>> k(1,2:end)=i(1: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 x 3 array:

Route	Day	Revenue in Thousands
1	0	3
1	1	5
1	3	4
1	5	5
2	1	3
2	2	3
2	4	3
2	6	3
3	6	4
3	0	4

In Microsoft Excel, there is a little known feature that is readily available on the Data menu called PivotTable which allows for the analysis of such data.

MATLAB and Star-P users can perform the same analysis with sparse matrices.

First we define the three column array:

```
>> m=[
1       0       3
1       1       5
1       3       4
1       5       5
2       1       3
2       2       3
```

```
2       4       3
2       6       3
3       6       4
3       0       4
]
m =
     1     0     3
     1     1     5
     1     3     4
     1     5     5
     2     1     3
     2     2     3
     2     4     3
     2     6     3
     3     6     4
     3     0     4
```

Then we create the sparse matrix a:

```
>> a = sparse(m(:,1),m(:,2)+1,m(:,3))

a =

   (1,1)        3
   (3,1)        4
   (1,2)        5
   (2,2)        3
   (2,3)        3
   (1,4)        4
   (2,5)        3
   (1,6)        5
   (2,7)        3
   (3,7)        4
```

How much does each of the three routes make as revenue:

```
>> sum(a')

ans =

   (1,1)       17
   (1,2)       12
   (1,3)        8
```

Or what is the total for each day:

```
>> sum(a)

ans =
```

```
   (1,1)        7
   (1,2)        8
   (1,3)        3
   (1,4)        4
   (1,5)        3
   (1,6)        5
   (1,7)        7
```

Or what is the total revenue:

```
 >> sum(a(:))

ans =

   (1,1)       37
```

Since Star-P extends the functionality of sparse matrices to parallel machines, one can do very sophisticated data analysis on large data sets using Star-P.

Note that the `sparse` command also adds data with duplicate indices.

If the `sparse` constructor encounters duplicate (i,j) indices, the corresponding nonzero values are added together. This is sometimes useful for data analysis; for example, here is an example of a routine that computes a weighted histogram using the `sparse` constructor. In the routine, `bin` is a vector that gives the histogram bin number into which each input element falls. Notice that `h` is a sparse matrix with just one column! However, all the values of `w` that have the same bin number are summed into the corresponding element of `h`. The MATLAB function `bar` plots a bar chart of the histogram.

```
>> 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.

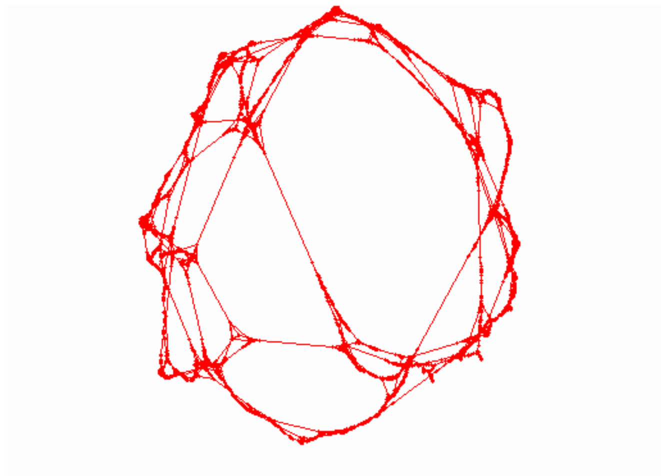


Figure 7-1 8192-vertex graph from Kern1 plotted with Fiedler coordinates

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:

```
Iij = {1 if node i is connected to node j, otherwise 0}
```

The matrix a in "Sparse Matrices: Representing Graphs and General Data Analysis" is actually an adjacency matrix. For any particular path length k, each element of I^k represents the number of paths that connect node i to node j.

```
 >> a = sprones(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 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.

The following commands will often be useful. Execute these commands on the HPC server in a terminal window.

- **`top`**: This command is often the most useful. See `man top` for details. It displays the most active processes on the system over the previous time interval, and can display all processes or just those of a specific user. It can help you understand if your Star-P server processes are being executed, if they're using the processors, if they're competing with other processes for the processors, etc. `top` also gives information about the amount of memory your processes are using, and the total amount of memory in use by all processes in the system.
- **`ps`**: 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 process IDs you need to kill the processes.

Architecture

The Star-P system is functionally composed of six modules as shown in Figure 8-1. 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** 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.

Important: The administration server is not required to use Star-P.

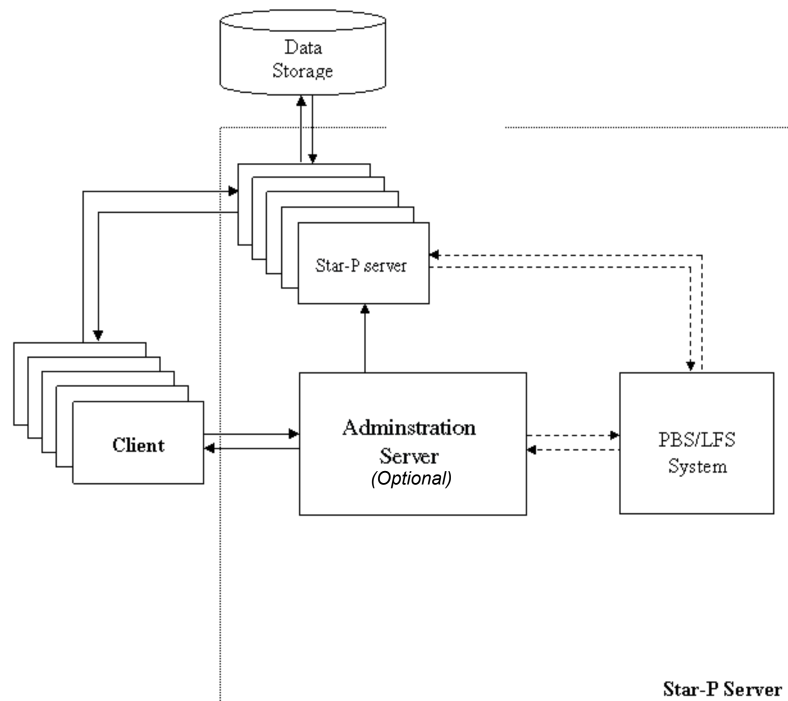


Figure 8-1 Star-P Architecture

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 web 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 not required to use Star-P. The Administration server currently works only with the SGI Altix platform.

- HPC 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 client authentication, session monitoring, data storage management, client configuration, HPC server administration, and user profile management. The Administration Server interactively provisions workgroups when required by clients.

  The HPC server maintains a persistent store of state information within the Administration server. 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

  — user session information

  — machine CPU and memory configuration

  — workgroup provisioning state and information

  — session data location in storage system and checkpointing information

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 a subsequent release.

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. By piping the contents of a MATLAB .m file into the starp command, the script will run to completion.

Example

```
cat myscript.m | starp -a server -u user -p 4 -t . -s /usr/local/starp-2.4.0
```

[1] Implementation of this feature is deferred.

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 such as the amount of network communication, number of times the distribution of a matrix is changed, etc.

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.

This section provides an overview of the SDK, see the **Star-P SDK Tutorial and Reference Guide** for detailed information.

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 (currently the GNU g++ compiler, version 3.2 and the Intel C++ compiler, version 9.1 are supported)
- ability to make the resulting shared objects accessible to StarP server

In order to use the functions created through the SDK, a user needs to know the path of the shared object file on the server and the names of the SDK functions

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 must contain real or complex data.

Distributed matrices are made accessible to the user code through smart pointers to instances of the `matrix_t` class. Each of these objects exposes metadata about a matrix such as the datatypes of its elements, the storage scheme and layout of the data among the various Star-P server processes. In addition, the matrix objects also provide access to the local portion of the matrix data on each process 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 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 must be exactly 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 cirumventing 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:

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

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.

Diagnostics and Performance

Star-P provides several diagnostic commands that help determine:

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

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

Client/Server Performance Monitoring

Coarse Timing with pptic and pptoc

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

```
 >> 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, pptic
c = a(:,1)*2 + b(:,1)*7.4;
toc, pptoc

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 **ppprofile on** and **ppprofile report**, sorted by the percentage of the execution time spent in that function. For this example, it shows you that 34% of the time is spent executing in the server routine ppfftw_fft, which calls the FFT routine in the FFTW parallel library. This report also tells you how many calls were made to each server routine, and the average time per call.

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

ppprofile 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
ppprofile report

%    Second, do it in a vectorized style

ppprofile clear, ppprofile on
tic
```

```
c = a*b;
fprintf('MxM via dgemm takes ');
toc
ppprofile 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
pppblas_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
pppblas_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 distance between the two styles of accomplishing this same computational task.

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

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

echo on
```

```
%  fft/ifft 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 **ppprofile on -detail full** option coupled with the **ppprofile 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.

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

```
>> ppeval_tic;
>> y = ppeval('inv',rand(10,10,1000*p));
>> ppeval_toc(0)

ans =

            TotalCalls: 1
            ServerInit: 0.7503
          ServerUnpack: 4.9989e-007
     ServerFunctionGen: 2.9325e-004
       ServerCallSetup: 2.4025e-004
      ServerOctaveExec: 0.0621
     ServerDataCollect: 2.9315e-004
       ServerTotalTime: 0.8132
         ClientArgScan: 0.0500
          ClientDepFun: 0.3910
```

```
          ClientEmode: 0.9740
    ClientReturnValues: 0.0500
       ClientTotalTime: 1.4650
```

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

Maximizing Performance

Maximizing performance of Star-P breaks down to

1. minimize client/server communication,
2. keep data movement between the client and server to a minimum, and
3. keep distributions of server variables aligned.

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

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

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

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

easily achieved by aligning the distribution types of all variables, i.e., all row distributed or all column distributed.

9 Supported MATLAB® Functions

This chapter lists the MATLAB[1] functions that are supported with Star-P. The tables in "Overloaded MATLAB Functions, Alphabetically" list the supported commands in alphabetical order, while the tables in "Supported MATLAB Functions, by Function Class" list all of the MATLAB functions according to their MATLAB function class.

Star-P implementation and testing is constantly evolving. Please refer to the support web page, www.interactivesupercomputing.com, for the most up-to-date function status.

A Pass-Thru entry means that the function is known to work by operating as MATLAB sourced script and explicitly implemented by Star-P.

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.

Overloaded MATLAB Functions, Alphabetically

Table 9-1 lists the MATLAB functions supported in Star-P.

Table 9-1 MATLAB functions Supported in Star-P

			Star-P Data Type				
MATLAB Function	**Function Class**	**Function Type**	**dlayout**	**ddense**	**ddenseND**	**dsparse**	**ppeval**
abs	elfun	Built-in	Available	Support		Support	Support
acos	elfun	Built-in	Available	Available		Support	Available
acosd	elfun	M-file	Available	Support	Support	Support	
acosh	elfun	Built-in	Available	Available		Support	Available
acot	elfun	M-file	Available				Available
acotd	elfun	M-file	Available			Support	
acoth	elfun	M-file	Available				Available
acsc	elfun	M-file	Available				Available
acscd	elfun	M-file	Available	Support	Support	Support	
acsch	elfun	M-file	Available				Available
airy	specfun	M-file					Available
all	ops	Built-in	Available	Support	Available	Support	Available
and	ops	Built-in	Available	Support	Support	Support	Available
angle	elfun	M-file	Available	Support		Support	A
any	ops	Built-in	Available	Support		Support	Available
asec	elfun	M-file	Available				Available
asecd	elfun	M-file	Available	Support	Support	Support	
asech	elfun	M-file	Available				Available
asin	elfun	Built-in	Available	Availablev ailable		Support	Support
asind	elfun	M-file	Available	Support	Support	Support	
asinh	elfun	Built-in	Available	Available		Support	Support
assignin	lang	Built-in					Available
atan	elfun	Built-in	Available	Available		Support	Support
atan2	elfun	Built-in	Available	Available		Support	SupportSu pport
atand	elfun	M-file	Available			Support	
atanh	elfun	Built-in	Available	Available		Support	Available
balance	matfun	Built-in					Available
base2dec	strfun	M-file	Available				Available
besselh	specfun	M-file					Available
besseli	specfun	M-file					Available
besselj	specfun	M-file					Available
besselk	specfun	M-file					Available
bessely	specfun	M-file					Available
beta	specfun	M-file					Available
betainc	specfun	M-file					Available
bin2dec	strfun	M-file	Available				Available
bitand	ops	Built-in					Available
bitcmp	ops	Built-in					Available

bitget	ops	Built-in					Available
bitmax	ops	M-file					Available
bitor	ops	Built-in					Available
bitset	ops	Built-in					Available
bitshift	ops	Built-in					Available
bitxor	ops	Built-in					Available
blanks	strfun	M-file					Available
blkdiag	elmat	M-file		Available		Available	Available
calendar	timefun	M-file					Available
cart2pol	specfun	M-file					Available
cart2sph	specfun	M-file					Available
cast	datatypes	Built-in					Available
cat	elmat	Built-in		Available		Support	Available
ceil	elfun	Built-in	Available	Available		Support	Available
cell	datatypes	Built-in		Available			Available
cell2mat	datatypes	M-file					Available
cell2struct	datatypes	Built-in					Available
cellfun	datatypes	MEX or dll					Available
cellstr	strfun	M-file					Available
char	strfun	Built-in	Available				Available
chol	matfun	Built-in		Available			Available
class	datatypes	Built-in					Available
clc	iofun	Built-in					Available
clf	graphics	M-file					Available
clock	timefun	Built-in					Available
close	graphics	M-file					Available
colamd	sparfun	M-file					Available
colon	ops	Built-in	Available				
colormap	graph3d	M-file					Available
colperm	sparfun	M-file				Support	Available
compan	elmat	M-file		Support		Support	Available
complex	elfun	Built-in	Available	Support	Support		Available
computer	general	Built-in					Available
cond	matfun	M-file		Support			Available
conj	elfun	Built-in	Available	Support		Support	Available
contour	specgraph	M-file					Available
conv	datafun	M-file					Available
conv2	datafun	Built-in		Available			
corrcoef	datafun	M-file					Available
cos	elfun	Built-in	Available	Available		Support	Support
cosd	elfun	M-file	Available	Available		Support	
cosh	elfun	Built-in	Available	Available		Support	Support
cot	elfun	M-file	Available				Available
cotd	elfun	M-file				Support	
coth	elfun	M-file	Available				Available
cov	datafun	M-file		Available		Support	Available
cputime	timefun	Built-in					Available
cross	specfun	M-file					Available

csc	elfun	M-file					Available
cscd	elfun	M-file	Available			Support	
csch	elfun	M-file	Available				Available
ctranspose (')	ops	Built-in	Available	Support		Support	
cumprod	datafun	Built-in		Available		Support	Available
cumsum	datafun	Built-in		Available		Support	Available
date	timefun	M-file					Available
datenum	timefun	M-file					Available
datestr	timefun	M-file					Available
datevec	timefun	M-file					Available
dbclear	codetools	Built-in					Available
dbstatus	codetools	Built-in					Available
dbstop	codetools	Built-in					Available
dbtype	codetools	Built-in					Available
deal	datatypes	M-file		Available		Available	Available
deblank	strfun	Built-in					Available
dec2base	strfun	M-file	Available				Available
dec2bin	strfun	M-file	Available				Available
dec2hex	strfun	M-file	Available				Available
deconv	datafun	M-file					Available
demo	demos	M-file					Available
detrend	datafun	M-file					Available
diag	elmat	Built-in		Support		Support	Available
diff	datafun	Built-in		Available			Available
disp	lang	Built-in	Available	Available		Available	Available
display	lang	Built-in	Available	Available		Available	
dmperm	sparfun	M-file					Available
dot	specfun	M-file		Available		Support	Available
double	datatypes	Built-in	Support				Available
eig	matfun	Built-in		Available			Available
eigs	sparfun	M-file				Available	
ellipke	specfun	M-file				Support	
end	lang	Built-in		Available		Available	
eomday	timefun	M-file					Available
eq	ops	Built-in	Support	Support	Support	Support	Available
erf	specfun	M-file					Available
erfc	specfun	M-file					Available
erfinv	specfun	M-file					Available
error	lang	Built-in					Available
errorbar	specgraph	M-file					Available
etime	timefun	M-file					Available
etree	sparfun	M-file					Available
etreeplot	sparfun	M-file					Available
eval	lang	Built-in					Available
evalin	lang	Built-in					Available
exist	lang	Built-in					Available
exit	general	Built-in					Available

exp	elfun	Built-in	Available	Available		Support	Available
expm	matfun	Built-in					Available
expm1	elfun	M-file	Available	Support		Support	
eye		Built-in	Support				Available
factor	specfun	M-file	Support				
factorial	specfun	M-file	Support				
false	elmat	Built-in	Available				
fclose	iofun	Built-in					Available
feof	iofun	Built-in					Available
ferror	iofun	Built-in					Available
feval	lang	Built-in		Support			Support
fft	datafun	Built-in		Available			Support
fft2	datafun	M-file		Support	Support		Support
fftn	datafun	Built-in					Available
fftshift	datafun	M-file				Support	Available
fgetl	iofun	M-file					Available
fgets	iofun	Built-in					Available
fieldnames	datatypes	Built-in					Available
figure	graphics	Built-in					Available
fileattrib	general	Built-in					Available
fileparts	iofun	M-file					Available
filter	datafun	Built-in					Available
find	elmat	Built-in	Available	Available		Support	Available
findstr	strfun	Built-in					Available
fix	elfun	Built-in	Available	Available	Available	Support	Available
flipdim	elmat	M-file				Support	Available
fliplr	elmat	M-file		Support		Support	Available
flipud	elmat	M-file		Support		Support	Available
floor	elfun	Built-in	Available	Available		Support	A
flops	elmat	M-file					Available
fopen	iofun	Built-in					Available
fprintf	iofun	Built-in	Available	Available		Available	Available
fread	iofun	Built-in					Available
freqspace	elmat	M-file	Support				
frewind	iofun	M-file					Available
fscanf	iofun	Built-in					Available
fseek	iofun	Built-in					Available
ftell	iofun	Built-in					Available
full	sparfun	Built-in	Available	Available		Available	Available
fullfile	iofun	M-file					Available
func2str	datatypes	Built-in					Available
functions	datatypes	Built-in					Available
fwrite	iofun	Built-in					Available
gamma	specfun	MEX or dll					Available
gammainc	specfun	M-file					Available
gammaln	specfun	MEX or dll					Available
gcd	specfun	M-file					Available
ge	ops	Built-in	Available	Support		Support	Available

getenv	general	Built-in					Available
gplot	sparfun	M-file					Available
grid	graph2d	M-file					Available
gt	ops	Built-in	Support	Support	Support	Support	Available
hadamard	elmat	M-file	Support				
hankel	elmat	M-file		Support		Support	Available
hess	matfun	Built-in		Available			Available
hex2dec	strfun	M-file	Available				Available
hex2num	strfun	M-file	Available				
hilb	elmat	M-file					Available
hist	datafun	M-file					Available
histc	datafun	MEX or dll		Available			
hold	graphics	M-file					Available
home	iofun	Built-in					Available
horzcat	ops	Built-in		Available		Support	Available
ifft	datafun	Built-in		Available			Available
ifft2	datafun	M-file		Available			
ifftn	datafun	Built-in					Available
ifftshift	datafun	M-file				Support	
imag	elfun	Built-in	Available	Support		Support	Available
image	specgraph	Built-in					Available
imagesc	specgraph	M-file					Available
ind2rgb	specgraph	M-file					Available
ind2sub	elmat	M-file				Support	Available
inf	elmat	Built-in	Available				
inline		M-file					Available
int2str	strfun	M-file					Available
intmax	elmat	M-file					Available
intmin	elmat	M-file					Available
inv	matfun	Built-in	Available	Support			Available
invhilb	elmat	M-file	Support				Available
ipermute	elmat	M-file		Support	Support		Available
isa	datatypes	Built-in		Available		Available	Available
iscell	datatypes	Built-in					Available
iscellstr	strfun	M-file					Available
ischar	strfun	Built-in					Available
isempty	elmat	Built-in		Available		Available	Available
isequal	elmat	Built-in	Available	Available		Available	Available
isequalwith equalnans	elmat	Built-in	Available	Available		Available	Available
isfield	datatypes	M-file					Available
isfinite	elmat	Built-in	Available	Available		Available	
isfloat	datatypes	Built-in	Available	Available		Available	
isglobal	lang	Built-in					Available
ishold	graphics	M-file					Available
isinf	elmat	Built-in	Available	Available		Available	Available
iskeyword	lang	M-file					Available
isletter	strfun	Built-in					Available

islogical	datatypes	Built-in	Available	Available		Available	
ismember	ops	M-file	Support	Available			Available
isnan	elmat	Built-in	Available	Available		Available	Available
isnumeric	datatypes	Built-in	Available	Available		Available	Available
ispc	general	M-file					Available
isprime	specfun	M-file	Available	Available		Support	
isreal	elfun	Built-in	Available	Support	Support	Support	Available
isscalar	elmat	Built-in					Available
isspace	strfun	Built-in		Available		Available	Available
issparse	sparfun	Built-in	A	Support		Support	Available
isstr	strfun	Built-in					A
isstruct	datatypes	Built-in					Available
isunix	general	M-file					Available
isvarname	lang	M-file					Available
isvector	elmat	Built-in					A
keyboard	lang	Built-in					Available
kron	ops	M-file		Support			Available
lasterr	lang	Built-in					Available
lastwarn	lang	Built-in					Available
lcm	specfun	M-file					A
ldivide (.\)	ops	Built-in	Available	Support		Support	Available
le	ops	Built-in	Available	Support		Support	Available
length	elmat	Built-in		Support	Support	Support	Available
license		Built-in					Available
lin2mu	audiovideo	M-file					Available
linspace	elmat	M-file	Support				Available
log	elfun	Built-in	Available	Available		Support	Support
log10	elfun	M-file	Available	Available		Support	Available
log1p	elfun	M-file	Available	Available		Support	
log2	elfun	Built-in	Available	Available		Support	Available
logical	datatypes	Built-in	Available	A		A	Available
loglog	graph2d	Built-in	Available	Available		Available	Available
logm	matfun	M-file					Available
logspace	elmat	M-file	Available				Available
lower	strfun	Built-in					Available
lt	ops	Built-in	Available	Support		Support	A
lu	matfun	Built-in		Available			Available
luinc	sparfun	Built-in					Available
magic	elmat	M-file	Support				
max	datafun	Built-in	Support	Support	Support	Support	Available
mean	datafun	M-file		Support	Support	Support	Available
median	datafun	M-file		Available		Available	Available
menu	uitools	M-file					Available
mesh	graph3d	M-file					Available
meshgrid	elmat	M-file		Support			Available
min	datafun	Built-in	Available	Support		Support	Available
minus (-)	ops	Built-in	Support	Support	Support	Support	Support
mislocked	lang	Built-in					Available

mkdir	general	Built-in					Available
mkpp	polyfun	M-file					Available
mldivide (\)	ops	Built-in	Available	Available		Available	Support
mlock	lang	Built-in					Available
mod	elfun	Built-in	Support	Support	Support	Support	Available
mpower (^)	ops	Built-in	Available	Support		Available	Support
mrdivide (/)	ops	Built-in	Available	Support		Available	Support
mtimes (*)	ops	Built-in	Support	Support	Support	Support	Support
mu2lin	audiovideo	M-file					Available
munlock	lang	Built-in					Available
nan	elmat	Built-in	Available				
nargchk	lang	Built-in					Available
nargin	lang	Built-in					A
nargout	lang	Built-in					Available
nchoosek	specfun	M-file	Support				
ndgrid	elmat	M-file				Support	
ndims	elmat	Built-in		Support	Support	Support	Available
ne	ops	Built-in	Support	Support	Support	Support	Available
nextpow2	elfun	M-file					Available
nnz	sparfun	M-file	Available	Support	Support	Support	Available
nonzeros	sparfun	M-file	Available	Available		Available	Available
norm	matfun	Built-in	Available	Support		Support	Available
normest	matfun	M-file				Support	
not	ops	Built-in	Available	Available		Support	Available
now	timefun	M-file					Available
null	matfun	M-file					Available
num2cell	datatypes	M-file					Available
num2hex	strfun	M-file	Available				
num2str	strfun	M-file	Available				Available
numel	elmat	Built-in		Support		Support	Support
nzmax	sparfun	M-file					Available
ones	elmat	Built-in	Available				Available
or	ops	Built-in	Support	Support	Support	Support	Available
orient	graphics	M-file					Available
orth	matfun	M-file		Support			Available
pack	general	Built-in					Available
path	general	M-file					Available
pause	timefun	Built-in					Available
pchip	polyfun	M-file					Available
permute	elmat	Built-in		Support	Support		Available
pinv	matfun	M-file		Support			Available
planerot	matfun	M-file		Support		Support	
plot	graph2d	Built-in	Available	Available		Available	Available
plot3	graph3d	Built-in	Available	Available		Available	
plus (+)	ops	Built-in	Support	Support	Support	Support	Support
pol2cart	specfun	M-file				Support	Available
polar	graph2d	M-file					Available
poly	polyfun	M-file					Available

polyder	polyfun	M-file					Available
polyfit	polyfun	M-file					Available
polyval	polyfun	M-file					Available
polyvalm	polyfun	M-file					Available
pow2	elfun	Built-in					Available
power (.^)	ops	Built-in	Available	Available		Available	Available
ppval	polyfun	M-file					Available
prod	datafun	Built-in	Available	Available		Support	Available
pwd	general	M-file					Available
qr	matfun	Built-in		Available			Available
quad	funfun	M-file					Available
quit	general	Built-in					Available
qz	matfun	Built-in					Available
rand	elmat	Built-in	Available				Available
randn	elmat	Built-in	Available				Available
randperm	sparfun	M-file					Available
rank	matfun	M-file		Support			Available
rdivide (./)	ops	Built-in	Available	Support		Support	Available
real	elfun	Built-in	Available	Available		Support	Available
regexp	strfun	Built-in					Available
regexpi	strfun	Built-in					Available
rehash	general	Built-in					Available
rem	elfun	Built-in	Support	Support	Support	Support	Available
repmat	elmat	M-file	Available	Available		Available	Support
reshape	elmat	Built-in	Available	Available		Available	Available
residue	polyfun	M-file					Available
rmdir	general	Built-in					Available
rmfield	datatypes	M-file					Available
roots	polyfun	M-file					Available
rot90	elmat	M-file		Support		Support	Available
round	elfun	Built-in	Available	Available		Support	Available
scatter	specgraph	M-file	Available	Available		Available	
schur	matfun	Built-in		Available			Available
sec	elfun	M-file	Available				Available
secd	elfun	M-file	Available			Support	
sech	elfun	M-file	Available				Available
semilogx	graph2d	Built-in	Available	Available		Available	Available
semilogy	graph2d	Built-in	Available	Available		Available	Available
setdiff	ops	M-file					Available
setstr	strfun	Built-in					Available
sign	elfun	Built-in	Available	Available		Support	Available
sin	elfun	Built-in	Available	Available		Support	Support
sind	elfun	M-file	Available	Available		Support	Available
single	datatypes	Built-in					Available
sinh	elfun	Built-in	Available	Available		Support	Support
size	elmat	Built-in		Support	Support	Support	Available
sort	datafun	Built-in	Available	Available		Available	Available
sortrows	datafun	M-file		Support		Support	Available

spalloc	sparfun	M-file					Available
sparse	sparfun	Built-in	Support	Support		Support	Available
spaugment	sparfun	M-file		Support		Support	
spconvert	sparfun	M-file					Available
spdiags	sparfun	M-file		Available		Available	
speye	sparfun	M-file	Available				Available
spfun	sparfun	M-file				Support	Available
sph2cart	specfun	M-file				Support	Available
spline	polyfun	M-file					Available
spones	sparfun	M-file				Available	Available
spparms	sparfun	M-file					Available
sprand	sparfun	M-file	Available				Available
sprandn	sparfun	M-file	Available				Available
sprandsym	sparfun	M-file					Available
sprintf	strfun	Built-in	Available	Available		Available	Available
spy	sparfun	M-file					Available
sqrt	elfun	Built-in	Available	Support		Support	Support
sqrtm	matfun	M-file		Support			Available
squeeze	elmat	M-file		Support	Support		Available
ss2tf	polyfun	M-file					Available
sscanf	strfun	Built-in					Available
stairs	specgraph	M-file					Available
std	datafun	M-file		Available		Support	Available
str2double	strfun	M-file					Available
str2func	datatypes	Built-in					Available
str2mat	strfun	M-file					Available
str2num	strfun	M-file					Available
strcat	strfun	M-file					Available
strcmp	strfun	Built-in					Available
strcmpi	strfun	Built-in					Available
strfind	strfun	Built-in					Available
strjust	strfun	M-file					Available
strmatch	strfun	M-file					Available
strncmp	strfun	Built-in					Available
strncmpi	strfun	Built-in					Available
strrep	strfun	Built-in					Available
struct	datatypes	Built-in					Available
struct2cell	datatypes	Built-in					Available
sub2ind	elmat	M-file		Available			Available
subplot	graph2d	M-file					Available
subsasgn	ops	Built-in	Available	Available		Available	
subsindex	ops	Built-in	Available	Available		Available	
subsref	ops	Built-in		Available		Available	
sum	datafun	Built-in	Available	Support	Support	Support	Available
svd	matfun	Built-in		Support			Available
svds	sparfun	M-file		Support		Support	
symamd	sparfun	M-file					Available
system	general	Built-in					Available

tan	elfun	Built-in	Available	Available		Support	Support
tand	elfun	M-file	Available	Available		Support	
tanh	elfun	Built-in	Available	Available		Support	Support
times (.*)	ops	Built-in	Support	Support	Support	Support	Available
toeplitz	elmat	M-file		Support		Support	Available
trace	matfun	M-file		Support		Support	Available
transpose (.')	ops	Built-in		Support		Support	
tril	elmat	Built-in	Available	Support		Support	Available
triu	elmat	Built-in	Available	Available		Available	Available
true	elmat	Built-in	Available				
uminus (-)	ops	Built-in	Available	Support	Support	Support	Available
union	ops	M-file		Support		Support	Available
unique	ops	M-file		Available		Available	Available
unmkpp	polyfun	M-file					Available
unwrap	elfun	M-file		Support			Available
uplus (+)	ops	Built-in	Available	Support	Support	Support	Available
upper	strfun	Built-in					Available
vander	elmat	M-file		Support		Support	Available
var	datafun	M-file		Available			Available
ver	general	M-file					Available
version		Built-in					Available
vertcat	ops	Built-in		Available		Support	Available
warning	lang	Built-in					Available
xor	ops	Built-in	Available	Available		Available	Available
zeros	elmat	Built-in	Available				Available

Supported MATLAB Functions, by Function Class

This section lists the MATLAB functions that Star-P supports, organized according their MATLAB function class.

MATLAB datafun function class

Table 9-2 lists the MATLAB functions that Star-P supports with a MATLAB function class of `datafun.`

Table 9-2 MATLAB functions Supported in Star-P of MATLAB function class datafun

			Star-P Data Type				
MATLAB Function	**Function Class**	**Function Type**	**dlayout**	**ddense**	**ddenseND**	**dsparse**	**ppeval**
conv	datafun	M-file					AvailableAvailable
corrcoef	datafun	M-file					Available
cov	datafun	M-file		Available		Support	Available
deconv	datafun	M-file					Available
detrend	datafun	M-file					Available
diff	datafun	Built-in		Available			Available
fft	datafun	Built-in		Available			Support
fft2	datafun	M-file		Support	Support		Support
fftn	datafun	Built-in					Available
fftshift	datafun	M-file				Support	Available
filter	datafun	Built-in					Available
hist	datafun	M-file					Available
histc	datafun	MEX or dll		Available			
ifft	datafun	Built-in		Available			Available
ifft2	datafun	M-file		Available			
ifftn	datafun	Built-in					Available
ifftshift	datafun	M-file				Support	
max	datafun	Built-in	Support	Support	Support	Support	Available
mean	datafun	M-file		Support	Support	Support	Available
median	datafun	M-file		Available		Available	Available
min	datafun	Built-in	Available	Support		Support	Available
prod	datafun	Built-in	Available	Available		Support	Available
sort	datafun	Built-in	Available	Available		Available	Available
sortrows	datafun	M-file		Support		Support	Available
std	datafun	M-file		Available		Support	Available
sum	datafun	Built-in	Available	Support	Support	Support	Available
var	datafun	M-file		Available			Available

MATLAB datatypes function class

Table 9-3 lists the MATLAB functions that Star-P supports with a function class of `datatypes`.

Table 9-3 MATLAB functions Supported in Star-P of MATLAB function class datatypes

			Star-P Data Type				
MATLAB Function	**Function Class**	**Function Type**	**dlayout**	**ddense**	**ddenseND**	**dsparse**	**ppeval**
cast	datatypes	Built-in					Available
cell	datatypes	Built-in		Available			Available
cell2mat	datatypes	M-file					Available
cell2struct	datatypes	Built-in					Available
cellfun	datatypes	MEX or dll					Available
class	datatypes	Built-in					Available
deal	datatypes	M-file		Available		Available	Available
double	datatypes	Built-in	Support				Available
fieldnames	datatypes	Built-in					Available
func2str	datatypes	Built-in					Available
functions	datatypes	Built-in					Available
isa	datatypes	Built-in		Available		Available	Available
iscell	datatypes	Built-in					Available
isfield	datatypes	M-file					Available
isfloat	datatypes	Built-in	Available	Available		Available	
islogical	datatypes	Built-in	Available	Available		Available	
isnumeric	datatypes	Built-in	Available	Available		Available	Available
isstruct	datatypes	Built-in					Available
logical	datatypes	Built-in	Available	Available		Available	Available
num2cell	datatypes	M-file					Available
rmfield	datatypes	M-file					Available
single	datatypes	Built-in					Available
str2func	datatypes	Built-in					Available
struct	datatypes	Built-in					Available
struct2cell	datatypes	Built-in					Available

MATLAB elfun function class

Table 9-4 lists the MATLAB functions that Star-P supports with a MATLAB function class of `elfun`

Table 9-4 MATLAB functions Supported in Star-P of MATLAB function class elfun

			Star-P Data Type				
MATLAB Function	**Function Class**	**Function Type**	**dlayout**	**ddense**	**ddenseND**	**dsparse**	**ppeval**
abs	elfun	Built-in	Available	SupportSupport		Support	Support
acos	elfun	Built-in	Available	Available		Available	Available
acosd	elfun	M-file	Available	Support	Support	Support	
acosh	elfun	Built-in	Available	Available		Support	Available
acot	elfun	M-file	Available				Available
acotd	elfun	M-file	Available			Support	
acoth	elfun	M-file	Available				Available
acsc	elfun	M-file	Available				Available
acscd	elfun	M-file	Available	Support	Support	Support	
acsch	elfun	M-file	Available				Available
angle	elfun	M-file	Available	Support		Support	Available
asec	elfun	M-file	Available				Available
asecd	elfun	M-file	Available	Support	Support	Support	
asech	elfun	M-file	Available				Available
asin	elfun	Built-in	Available	Available		Available	Support
asind	elfun	M-file	Available Available	Support	Support	Support	
asinh	elfun	Built-in	Available	Available		Support	Support
atan	elfun	Built-in	Available	Available		Support	Support
atan2	elfun	Built-in	Available	Available		Support	Support
atand	elfun	M-file	Available			Support	
atanh	elfun	Built-in	Available	Available		Support	Available
ceil	elfun	Built-in	Available	Available		Support	Available
complex	elfun	Built-in	Available	Support	Support		Available
conj	elfun	Built-in	Available	Support		Support	Available
cos	elfun	Built-in	Available	Available		Support	Support
cosd	elfun	M-file	Available	Available		Support	
cosh	elfun	Built-in	Available	Available		Support	Support
cot	elfun	M-file	Available				Available
cotd	elfun	M-file	Available			Support	
coth	elfun	M-file	Available				Available
csc	elfun	M-file	Available				Available
cscd	elfun	M-file	Available			Support	
csch	elfun	M-file	Available				Available
exp	elfun	Built-in	Available	Available		Support	Available
expm1	elfun	M-file	Available	Support		Support	
fix	elfun	Built-in	Available	Available		Support	Available
imag	elfun	Built-in	Available	Support		Support	Available

isreal	elfun	Built-in	Available	Support		Support	Available
log	elfun	Built-in	Available	Available		Support	Support
log10	elfun	M-file	Available	Available		Support	Available
log1p	elfun	M-file				Support	
log2	elfun	Built-in	Available	Available		Support	Available
mod	elfun	Built-in	Support	Support	Support	Support	Available
nextpow2	elfun	M-file					Available
real	elfun	Built-in	Available	Available		Support	Available
rem	elfun	Built-in	Support	Support	Support	Support	Available
round	elfun	Built-in	Available	Available		Support	Available
sec	elfun	M-file	Available				Available
secd	elfun	M-file	Available			Support	
sech	elfun	M-file	Available				Available
sign	elfun	Built-in	Available	Available		Support	Available
sin	elfun	Built-in	Available	Available		Support	Support
sind	elfun	M-file	Available	Available		Support	Available
sinh	elfun	Built-in	Available	Available		Support	Support
sqrt	elfun	Built-in	Available	Support		Support	Support
tan	elfun	Built-in	Available	Available		Support	Support
tand	elfun	M-file	Available	Available		Support	
tanh	elfun	Built-in	Available	Available		Support	Support
unwrap	elfun	M-file		Support			Available

MATLAB elmat function class

- Table 9-5 lists the MATLAB functions that Star-P supports with a MATLAB function class of `elmat`.

Table 9-5 MATLAB functions Supported in Star-P of MATLAB function class elmat

			Star-P Data Type				
MATLAB Function	**Function Class**	**Function Type**	**dlayout**	**ddense**	**ddenseND**	**dsparse**	**ppeval**
blkdiag	elmat	M-file		Support		Support	Available
cat	elmat	Built-in		Available		Support	Available
compan	elmat	M-file		Support		Support	Available
diag	elmat	Built-in		Support		Support	Available
find	elmat	Built-in	Available	Available		Support	Available
flipdim	elmat	M-file				Support	Available
fliplr	elmat	M-file		Support		Support	Available
flipud	elmat	M-file		Support		Support	Available
flops	elmat	M-file					Available
freqspace	elmat	M-file	Support				
hadamard	elmat	M-file	Support				
hankel	elmat	M-file		Support		Support	Available
hilb	elmat	M-file					Available
ind2sub	elmat	M-file				Support	Available
inf	elmat	Built-in	Available				
intmax	elmat	M-file					Available
intmin	elmat	M-file					Available
invhilb	elmat	M-file	Support				Available
ipermute	elmat	M-file		Support	Support		Available
isempty	elmat	Built-in		Available		Available	Available
isequal	elmat	Built-in	Available	Available		Available	Available
isequalwith equalnans	elmat	Built-in	Available	Available		Available	Available
isfinite	elmat	Built-in	Available	Available		Available	
isinf	elmat	Built-in	Available	Available		Available	Available
isnan	elmat	Built-in	Available	Available		Available	Available
isscalar	elmat	Built-in					Available
isvector	elmat	Built-in					Available
length	elmat	Built-in		Support	Support	Support	Available
linspace	elmat	M-file	Support				Available
logspace	elmat	M-file	Available				Available
magic	elmat	M-file	Support				
meshgrid	elmat	M-file		Support		Support	Available
nan	elmat	Built-in	Available				
ndgrid	elmat	M-file				Support	
ndims	elmat	Built-in		Support	Support	Support	Available
numel	elmat	Built-in		Support		Support	Support
ones	elmat	Built-in	Available				Available

permute	elmat	Built-in		**Support**	**Support**		**Available**
rand	elmat	Built-in	**Available**				**Available**
randn	elmat	Built-in	**Available**				**Available**
repmat	elmat	M-file	**Available**				**Support**
reshape	elmat	Built-in	**Available**	**Available**		**Available**	**Available**
rot90	elmat	M-file		**Support**		**Support**	**Available**
size	elmat	Built-in		**Support**	**Support**	**Support**	**Available**
squeeze	elmat	M-file		**Support**	**Support**		**Available**
sub2ind	elmat	M-file		**Available**			**Available**
toeplitz	elmat	M-file		**Support**		**Support**	**Available**
trace	matfun	M-file		**Support**		**Support**	**Available**
transpose	ops	Built-in		**Support**		**Support**	
tril	elmat	Built-in		**Support**		**Support**	**Available**
triu	elmat	Built-in	**Available**	**Available**		**Available**	**Available**
true	elmat	Built-in	**Available**				
vander	elmat	M-file		**Support**		**Support**	**Available**
zeros	elmat	Built-in	**Available**				**Available**

MATLAB iofun function class

Table 9-6 lists the MATLAB functions that Star-P supports with a MATLAB function class of `iofun`.

Table 9-6 MATLAB functions Supported in Star-P of MATLAB function class iofun

			Star-P Data Type				
MATLAB Function	**Function Class**	**Function Type**	**dlayout**	**ddense**	**ddenseND**	**dsparse**	**ppeval**
clc	iofun	Built-in					AvailableAvailable
fclose	iofun	Built-in					Available
feof	iofun	Built-in					Available
ferror	iofun	Built-in					AvailableAvailable
fgetl	iofun	M-file					Available
fgets	iofun	Built-in					Available
fileparts	iofun	M-file					AvailableAvailable
fopen	iofun	Built-in					Available
fprintf	iofun	Built-in	Available	Available		Available	Available
fread	iofun	Built-in					Available
frewind	iofun	M-file					Available
fscanf	iofun	Built-in					Available
fseek	iofun	Built-in					Available
ftell	iofun	Built-in					Available
fullfile	iofun	M-file					Available
fwrite	iofun	Built-in					Available
home	iofun	Built-in					Available

MATLAB lang function class

Table 9-7 lists the MATLAB functions that Star-P supports with a MATLAB function class of `lang`.

Table 9-7 MATLAB functions Supported in Star-P of MATLAB function class lang

			Star-P Data Type				
MATLAB Function	**Function Class**	**Function Type**	**dlayout**	**ddense**	**ddenseND**	**dsparse**	**ppeval**
assignin	lang	Built-in					Available
disp	lang	Built-in	Available	Available		Available	Available
display	lang	Built-in	Available	Available		Available	
end	lang	Built-in		Available		Available	
error	lang	Built-in					Available
eval	lang	Built-in					Available
evalin	lang	Built-in					Available
exist	lang	Built-in					Available
feval	lang	Built-in		Support			Support
isglobal	lang	Built-in					Available
iskeyword	lang	M-file					Available
isvarname	lang	M-file					Available
keyboard	lang	Built-in					Available
lasterr	lang	Built-in					Available
lastwarn	lang	Built-in					Available
mislocked	lang	Built-in					Available
mlock	lang	Built-in					Available
munlock	lang	Built-in					Available
nargchk	lang	Built-in					Available
nargin	lang	Built-in					Available
nargout	lang	Built-in					AvailableAvailable
warning	lang	Built-in					Available

MATLAB matfun function class

Table 9-8 lists the MATLAB functions that Star-P supports with a MATLAB function class of `matfun`.

Table 9-8 MATLAB functions Supported in Star-P of MATLAB function class matfun

			Star-P Data Type				
MATLAB Function	**Function Class**	**Function Type**	**dlayout**	**ddense**	**ddenseND**	**dsparse**	**ppeval**
balance	matfun	Built-in					Available
chol	matfun	Built-in		Available			Available
cond	matfun	M-file		Support			Available
eig	matfun	Built-in		Available			Available
expm	matfun	Built-in					Available
hess	matfun	Built-in		Available			Available
inv	matfun	Built-in	Available	Support			Available
logm	matfun	M-file					Available
lu	matfun	Built-in		Available			Available
norm	matfun	Built-in	Available	Support		Support	Available
normest	matfun	M-file				Support	
null	matfun	M-file					Available
orth	matfun	M-file		Support			Available
pinv	matfun	M-file		Support			Available
planerot	matfun	M-file		Support		Support	
qr	matfun	Built-in		Available			Available
qz	matfun	Built-in					Available
rank	matfun	M-file		Support			Available
schur	matfun	Built-in		Available			Available
sqrtm	matfun	M-file		Support			Available
svd	matfun	Built-in		Support			Available
trace	matfun	M-file		Support		Support	Available

MATLAB ops function class

Table 9-9 lists the MATLAB functions that Star-P supports with a MATLAB function class of `ops`.

Table 9-9 MATLAB functions Supported in Star-P of MATLAB function class ops

MATLAB Function	Function Class	Function Type	Star-P Data Type				
			dlayout	ddense	ddenseND	dsparse	ppeval
all	ops	Built-in	Available	Support		Support	Available
and	ops	Built-in	Available	Support	Support	Support	Available
any	ops	Built-in	Available	Support		Support	Available
bitand	ops	Built-in					Available
bitcmp	ops	Built-in					Available
bitget	ops	Built-in					Available
bitmax	ops	M-file					Available
bitor	ops	Built-in					Available
bitset	ops	Built-in					Available
bitshift	ops	Built-in					Available
bitxor	ops	Built-in					Available
colon	ops	Built-in	Available				
ctranspose (')	ops	Built-in		Support		Support	
eq	ops	Built-in	Support	Available	Available	Support	Available
ge	ops	Built-in	Available	Support		Support	Available
gt	ops	Built-in	Support	Support	Support	Support	Available
horzcat	ops	Built-in		Available		Support	Available
kron	ops	M-file		Support			Available
ldivide (.\)	ops	Built-in	Available	Support		Support	Available
le	ops	Built-in	Available	Support		Support	Available
lt	ops	Built-in	Available	Support		Support	Available
minus (-)	ops	Built-in	Support	Support	Support	Support	Support
mldivide (\)	ops	Built-in	Available	Support		Available	Support
mpower (^)	ops	Built-in	Available	Support		Available	Support
mrdivide (/)	ops	Built-in	Available	Support		Available	Support
mtimes (*)	ops	Built-in	Support	Support	Support	Support	Support
ne	ops	Built-in	Support	Support	Support	Support	Available
not	ops	Built-in	Available	Available		Support	Available
or	ops	Built-in	Support	Support	Support	Support	Available
plus (+)	ops	Built-in	Support	Support	Support	Support	Support
power (.^)	ops	Built-in	Available	Available		Available	A
rdivide (./)	ops	Built-in	Available	Support		Support	Available
setdiff	ops	M-file					Available
subsasgn	ops	Built-in	Available	Available		Available	
subsindex	ops	Built-in	Available	Available		Available	
subsref	ops	Built-in		Available		Available	
times (.*)	ops	Built-in	Support	Support	Support	Support	Available

transpose (.')	ops	Built-in		Support		Support	
uminus (-)	ops	Built-in	Available	Support	Support	Support	Available
union	ops	M-file		Support		Support	Available
unique	ops	M-file		Available		Available	Available
uplus (+)	ops	Built-in	Available	Support	Support	Support	Available
vertcat	ops	Built-in		Available		Support	Available
xor	ops	Built-in	Available	Available		Available	Available

MATLAB sparfun function class

Table 9-10 lists the MATLAB functions that Star-P supports with a MATLAB function class of `sparfun`.

Table 9-10 MATLAB functions Supported in Star-P of MATLAB function class sparfun

			Star-P Data Type				
MATLAB Function	**Function Class**	**Function Type**	**dlayout**	**ddense**	**ddenseND**	**dsparse**	**ppeval**
colamd	sparfun	M-file					Available
colperm	sparfun	M-file				Support	Available
dmperm	sparfun	M-file					Available
eigs	sparfun	M-file				AvailableAvailable	
etree	sparfun	M-file					Available
etreeplot	sparfun	M-file					Available
full	sparfun	Built-in	Available	Available		Available	Available
gplot	sparfun	M-file					Available
issparse	sparfun	Built-in	Available	Support		Support	Available
luinc	sparfun	Built-in					Available
nnz	sparfun	M-file		Support	Support	Support	Available
nonzeros	sparfun	M-file					Available
nzmax	sparfun	M-file					Available
randperm	sparfun	M-file					Available
spalloc	sparfun	M-file					Available
sparse	sparfun	Built-in	Available	Available		Available	Available
spaugment	sparfun	M-file		Support		Support	
spconvert	sparfun	M-file					Available
spdiags	sparfun	M-file		Available		Support	
speye	sparfun	M-file	Available				Available
spfun	sparfun	M-file				Support	Available
spones	sparfun	M-file				Available	Available
spparms	sparfun	M-file					Available
sprand	sparfun	M-file	Available				Available
sprandn	sparfun	M-file	Available				Available
sprandsym	sparfun	M-file					Available
spy	sparfun	M-file					Available
svds	sparfun	M-file		Support		Support	
symamd	sparfun	M-file					Available

MATLAB specfun function class

Table 9-11 lists the MATLAB functions that Star-P supports with a MATLAB function class of `specfun`.

Table 9-11 MATLAB functions Supported in Star-P of MATLAB function class specfun

			Star-P Data Type				
MATLAB Function	Function Class	Function Type	dlayout	ddense	ddenseND	dsparse	ppeval
airy	specfun	M-file					AvailableAvailableAvailable
besselh	specfun	M-file					Available
besseli	specfun	M-file					Available
besselj	specfun	M-file					Available
besselk	specfun	M-file					Available
bessely	specfun	M-file					Available
beta	specfun	M-file					Available
betainc	specfun	M-file					Available
cart2pol	specfun	M-file					Available
cart2sph	specfun	M-file					Available
cross	specfun	M-file					Available
dot	specfun	M-file		Available		Support	Available
ellipke	specfun	M-file				Support	
erf	specfun	M-file					Available
erfc	specfun	M-file					Available
erfinv	specfun	M-file					Available
factor	specfun	M-file	Support				
factorial	specfun	M-file	Support				
gamma	specfun	MEX or dll					Available
gammainc	specfun	M-file					Available
gammaln	specfun	MEX or dll					Available
gcd	specfun	M-file					Available
isprime	specfun	M-file	Available	Available		Support	
lcm	specfun	M-file					Available
nchoosek	specfun	M-file	Support				
pol2cart	specfun	M-file				Support	Available
sph2cart	specfun	M-file				Support	Available

MATLAB strfun function class

Table 9-12 lists the MATLAB functions that Star-P supports with a MATAB function class of `strfun`.

Table 9-12 MATLAB functions Supported in Star-P of MATLAB function class strfun

			Star-P Data Type				
MATLAB Function	**Function Class**	**Function Type**	**dlayout**	**ddense**	**ddenseND**	**dsparse**	**ppeval**
base2dec	strfun	M-file					Available
bin2dec	strfun	M-file					Available
blanks	strfun	M-file					Available
cellstr	strfun	M-file					Available
char	strfun	Built-in					Available
deblank	strfun	Built-in					Available
dec2base	strfun	M-file					Available
dec2bin	strfun	M-file					Available
dec2hex	strfun	M-file					Available
intmin	elmat	M-file					Available
iscellstr	strfun	M-file					Available
ischar	strfun	Built-in					Available
isletter	strfun	Built-in					Available
isspace	strfun	Built-in		Available		Available	Available
isstr	strfun	Built-in					Available
lower	strfun	Built-in					Available
num2str	strfun	M-file	Available				Available
regexp	strfun	Built-in					Available
regexpi	strfun	Built-in					Available
setstr	strfun	Built-in					Available
sprintf	strfun	Built-in	Available	Available	Available	Available	Available
sscanf	strfun	Built-in					Available
str2double	strfun	M-file					Available
str2mat	strfun	M-file					Available
str2num	strfun	M-file					Available
strcat	strfun	M-file					Available
strcmp	strfun	Built-in					Available
strcmpi	strfun	Built-in					Available
strfind	strfun	Built-in					Available
strjust	strfun	M-file					Available
strmatch	strfun	M-file					Available
strncmp	strfun	Built-in					Available
strncmpi	strfun	Built-in					Available
strrep	strfun	Built-in					Available
upper	strfun	Built-in					Available

10 Star-P Functions

This chapter summarizes the Star-P functions that are not part of standard MATLAB and describes their implementation. It also describes the syntax of the * Star-P functions.

Basic Server Functions Summary

The following table lists the types of functions available.

Table 10-1 Star-P Function

Function	Description
General Functions	
np	Returns the number of processes in the server.
p	Creates an instance of a `dlayout` object.
pp	Is useful for users who wish to use the variable `p` for another purpose.
ppgetoption	Returns the value of Star-P properties.
ppsetoption	Sets the value of Star-P properties.
ppgetlog	Get the Star-P server log file.
`ppinvoke`	Invoke a function contained in a previously loaded user library via the Star-P Software Development Kit (SDK).
`pploadpackage`	Load a compiled user library on the server.
`ppunloadpackage`	Unload a user library from the server.
`ppfopen`	Open a distributed server-side file descriptor. The syntax is similar to that of the regular fopen() but the file is accessed on the server.

Table 10-1 Star-P Function

Function	Description
ppquit	Disconnects from the server and causes the server to terminate.
ppwhos	Gives information about distributed variables and their sizes (similar to **`whos`**).
pph5whos	Print information about variables in a HDF5 file.
Data Movement Functions	
ppback	Transfers local matrix to the server and stores the handle to the server matrix.
ppfront	Retrieves distributed matrix from the server and stores it in local matrix.
ppchangedist	Allows you to explicitly change the distribution of a matrix in order to avoid implicit changes in subsequent operations.
pph5write	Writes variables to a HDF5 file on the server.
pph5read	Reads distributed variables from a HDF5 file on the server
ppload	Loads a data set from the server filesystem to the backend.
ppsave	Saves backend data to the server filesystem.
Task Parallel Functions	
bcast	Broadcasts an array section where the entire argument is passed along to each function invocation.
split	Splits an array for each iteration of a `ppeval` function.
ppeval	Executes a specified function in parallel on sections of input array(s)
ppevalc	Executes a specified function in parallel on sections of input array(s) in C++

Table 10-1 Star-P Function

Function	Description
NOTE: The `ppevalc` and `ppevalcsplit` MATLAB functions have the same functionality as ppeval and ppevalsplit. The main difference is that the function name passed as the first input argument refers to a native function from a `ppevalc` module implemented as a loadable library on the server, rather than a MATLAB function defined on the client. To learn more, see "About ppevalc" on page 56. The function name argument to ppevalc or ppevalcsplit is a string with the format `MODULENAME:FNAME`, where `MODULENAME` is the module name as returned by an earlier call to `ppevalcloadmodule`, and, `FNAME` is the function name registered in that module. For example, the call: `ppevalc('solverlib:polyfit', arg1, arg2);` invokes the `polyfit` function in the `imsl` ppevalc module with input arguments `arg1` and `arg2`.	
ppevalsplit	Returns a dcell object, a cell array of return values from each iteration
ppevalcsplit	Returns a dcell object, a cell array of return values from each iteration
ppevalcloadmodule	Loads a ppevalc module on the server.
ppevalcunloadmodule	Removes a previously loaded ppevalc module.
Performance Functions	
ppprofile	Collects and display performance information on Star-P
pptic/toc	Provides information complementary to the MATLAB `tic/toc` command

Known Differences Between MATLAB and Octave Functions

This section lists the known differences between MATLAB and Octave.

- If an **`inf`** value is present in a matrix that is used as an argument to **`eig`** in ppeval, Star-P may hang, while MATLAB returns an error.
- The evaluation of the '++' and '--' auto-increment/decrement operators differs between ppeval and MATLAB. For example, x=7;++x returns 8 in `ppeval`, but returns 7 in MATLAB.

General Functions

np

`n = np`

Function Syntax Description

- `n` (`double`) - number of processes
- `np` returns the number of processes in the server. This is the argument that was passed to the `-p` switch to `starp`.

NOTE: This number should be less than or equal to the number of processors.

p

`z = p`

Function Syntax Description

- `z` (`dlayout`) - a `dlayout` object
- `p` creates an instance of a `dlayout` object. `p` by itself is a dumb 'symbolic variable'. Variables of type `dlayout` are used to tag dimensions as being distributed.

pp

`z = pp`

Function Syntax Description

- `z` (`dlayout` object) - a `dlayout` object
- `pp` is an alias to `p`. `pp` is useful for users who wish to use the variable `p` for another purpose.

Reference

- See `z = p`.

ppgetoption

Returns the value of the Star-P properties.

ppsetoption

Sets the value of the Star-P properties.

```
ppsetoption('option','value')
```

Function Syntax Description

- `ppsetoption('SparseDirectSolver', 'value')` where `value` can be `SuperLU`. We expect other options in the future.
- `ppsetoption('log', 'value')` where `value` can be one of `on` (default) or `off`. This controls whether information about the steps executed by the server is written to the log.
- `ppsetoption('pp2ml_msg','value')` where `value` can be one of `on` (default) or `off`. This controls whether the warning message from `pp2matlab` and `matlab2pp` about large transfers and from `ppchangedist` about large redistributions is emitted or not.
- `ppsetoption('pp2ml_size',size)` where `size` is the threshold above which the `pp2matlab/matlab2pp/ppchangedist` warning message will be emitted. The default is 100 megabytes.

ppgetlog

Get the Star-P server log file.

Function Syntax Description

- `f = ppgetlog`

  Returns the filename of a local temporary file containing the Star-P server log. The temporary file is deleted when MATLAB exits.

- `ppgetlog(FILENAME)`

  Stores a copy of the Star-P server log file in `FILENAME`.

ppinvoke

Invoke a function contained in a previously loaded user library via Star-P SDK.

Function Syntax Description

```
[varargout] = ppinvoke (function, varargin)
```

NOTE: See the ***Star-P Software Development Kit Tutorial and Reference Guide*** for more information on this function.

ppLoadpackage

Load a compiled user library on the server.

Function Syntax Description

```
pploadpackage (package_name)
```

NOTE: See the ***Star-P Software Development Kit Tutorial and Reference Guide*** for more information on this function.

ppunloadpackage

Unload a user library from the server.

Function Syntax Description

```
ppunloadpackage (package_name)
```

NOTE: See the ***Star-P Software Development Kit Tutorial and Reference Guide*** for more information on this function.

ppfopen

Open a distributed server-side file descriptor. The syntax is similar to that of the regular fopen() but the file is accessed on the server.

Function Syntax Description

```
FID = ppfopen('F')
```

Opens file ‘F’ in read-only mode.

```
FID = ppfopen('F', MODE)
```

Opens file F in the mode specified by MODE. MODE can be: '

Table 10-2 ppfopen MODE

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

Return Values

The return value FID is a distributed file identifier. Passing this value to the following MATLAB functions: `fopen()`, `fread()`, `fwrite()`, `frewind()` and `fclose()` will operate on distributed matrices on the server with the same semantics as with regular file id on the client.

ppquit

Disconnects from the server and causes the server to terminate.

ppwhos

`ppwhos` lists the variables in the caller's Star-P workspace. `ppwhos` is aware of distributed matrices that exist on the server so it will return the correct dimensions and sizes for those matrices, as well as returning the distribution information.

`ppwhos` is the Star-P equivalent of the MATLAB `whos` command. It provides detailed information about the distribution of the server side variables (2nd column), their size (3rd column), and their types (4th column).

All distributed variables will also show up in the MATLAB `whos` command, but the information displayed for these variables does not accurately represent their size and distribution properties. The `ppwhos` output helps align the distributions of the variables; in general having similar distributions for all variables provides the best performance. It also allows identifying variables that should be distributed, since they are large, which variables are not, and variables that should not be distributed, since they are small, but are distributed. A typical `ppwhos` output looks something like this:

```
>> a = rand(1000,1000*p);
>> b = rand(1000*p,1000);
>> c = rand(1000,1000);
>> ppwhos
Your variables are:
  Name      Size            Bytes       Class
  a         1000x1000p      8000000     ddense array
  b         1000px1000      8000000     ddense array
  c         1000x1000       8000000     double array

Grand total is 3000000 elements using 24000000 bytes
MATLAB has a total of 1000000 elements using 8000000 bytes
Star-P server has a total of 2000000 elements using 16000000 bytes
```

pph5whos

Print information about variables in a HDF5 file.

```
pph5whos('FILE')
```

Prints size and type information of variables in an HDF5 FILE on the server. The format is similar to the MATLAB `WHOS` function.

```
S = pph5whos ('FILE')
```

Returns the dataset names in an HDF5 FILE along with the corresponding size and type information in a structure array, S.

NOTE: `pph5whos` is able to parse an arbitrary HDF5 file, but will return accurate size and type information only for datasets that consist of double or double complex dense and sparse data. In all other cases, the type field is marked 'unknown'.

Reference

See also: `pph5write`, `pph5read`.

Data Movement Functions

ppback

```
B = ppback(A)
B = ppback(A, d)
```

`Transfer the MATLAB matrix A to the backend server and stores the result in B. A can be dense or sparse.`

NOTE: Replaces the matlab2pp command.

Function Syntax Description

- Input:
  - `A` (dense/sparse matrix) - MATLAB matrix to be transferred
  - `d` (optional) - distribution
- Output

  `B` (`ddense/ddensend/dsparse` matrix) - distributed matrix

  Transfer the MATLAB matrix `A` to the backend server and store the result in `B`.

  If `A` is dense and two-dimensional:
  - If `d` is not specified, then `B` is column distributed unless it is a column vector of length > 1, in which case it is row distributed.
  - If `d` is 1, then `B` is row distributed.
  - If `d` is 2, then `B` is column distributed.
  - If `d` is 3, then `B` is block cyclic distributed.

If A is dense and greater than two-dimensional:

— If d is not specified, then B is distributed along the last dimension, else.

— B is distributed along the dimension specified.

— Block cyclical distributions of B are not defined for A having greater than two dimensions

If A is sparse:

— B is row distributed.

IMPORTANT: A warning message is displayed if the transfer is over a threshold size (currently 100MB), to avoid silent performance losses. Emission of the message or the value of the threshold can be changed by use of the ppsetoption command

Reference:

See also ppfront, ppsetoption.

ppfront

Transfers the distributed matrix A from the server to the MATLAB client.

NOTE: Replaces B = pp2matlab(A)

```
B = ppfront (A)
```

Function Syntax Description

- Input: A - distributed matrix
- Output: B (dense/sparse MATLAB matrix) - local copy of A

ppfront transfers the distributed matrix A from the server to the MATLAB client.

- If A is a distributed dense matrix, then B is a dense MATLAB matrix.
- If A is a distributed sparse matrix, then B is a sparse matrix.

dlayout objects are converted to double and other non-distributed objects are preserved.

IMPORTANT: A warning message is emitted if the transfer is over a threshold size (currently 100MB), to avoid silent performance losses. Displays the warning message or the value of the threshold can be changed by use of the ppsetoption command.

Reference

See also ppback, ppsetoption.

ppchangedist

The ppchangedist command allows you to explicitly change the distribution of a matrix in order to avoid implicit changes in subsequent operations. This is especially important to do when performing operations within loops. In order to maximize performance, operands should have conformant distributions. `ppchangedist` can be used before and/or after the loop to prepare for subsequent operations.

Function Syntax Description

```
ppchangedist(a,dist)
```

- a is input ddense
- dist is the desired distribution
  - 1 for ROW
  - 2 for COL
  - 3 for CYCLIC

NOTE: A warning message is emitted if the transfer is over a threshold size (currently 100MB), to avoid silent performance losses. Emission of the message or the value of the threshold can be changed by use of the ppsetoption command.

pph5write

Write variables to a HDF5 file on the server. .

Function Syntax Description

```
PPH5WRITE('FILE', VARIABLE1, 'DATASET1', VARIABLE2, 'DATASET2',
...)
```

Writes `VARIABLE1` to `DATASET1` in the `FILE` specified on the server in the HDF5 format.

- If the `FILE` already exists, it is overwritten.
- Similarly if one of the dataset already exists, it is also overwritten with the new variable.

```
PPH5WRITE('FILE', 'MODE', ...)
```

Specifies the output mode which can either be 'clobber' or 'append'.

- If the mode is 'clobber' and `FILE` already exists, it is overwritten.
- If the mode is 'append' and `FILE` already exists, the variables specified in the `PPH5WRITE` call are appended to the `FILE`. If `FILE` does not exist, it is newly created.

Example 1

```
% Write matrix_a to the dataset /my_matrices/a and matrix_b to the
% dataset /my_matrices/workspaces/temp/matrix_b to the file
% /tmp/temp.h5 on the server, overwriting it if it already exists
pph5write('/tmp/temp.h5', matrix_a, '/my_matrices/a', matrix_b, '/
          my_matrices/workspace/temp/matrix_b');
```

Example 2

```
% Append matrix_c to the existing file in the location /
% my_matrices/workspace2/temp/matrix_c
pph5write('/tmp/temp.h5', 'append', matrix_c,'/my_matrices/
          workspace2/temp/matrix_c');
```

NOTE: Currently, only writing double and double complex dense and sparse matrices is supported.

Reference

See also: `pph5read`, `pph5whos`.

pph5read

```
[VARIABLE1, VARIABLE2, ...] = pph5read ('FILE', 'DATASET1',
'DATASET2', ...)
```

Read distributed variables from a HDF5 file on the server.

Reads from `FILE`, the contents of `DATASET1` into `VARIABLE1`, `DATASET2` into `VARIABLE2`, etc.

- If any of the datasets is missing or invalid, or the `FILE` is not a valid HDF5 file, the function returns an error.

Example

```
% Read the contents of the dataset /my_matrices/workspace/temp/
% matrix_b from the file /tmp/temp.h5 into the distributed
% variable matrix_d
matrix_d = pph5read('/tmp/temp.h5', '/my_matrices/workspace/temp/
                    matrix_b');
```

NOTE: Only the contents of datasets which contain double or double complex dense or sparse data can currently be read. In the latter case, the sparse matrix must be stored in a specific format outlined in the *Star-P User Guide*.

Reference

See also: PPH5WRITE, PPH5WHOS.

ppload

```
ppload('f', 'v1', 'v2', ..., 'dist')
```

Loads the distributed objects named `v1`, `v2`, ... from the file `f` into variables of the same names. Specify the distribution to use with dist.

Function Syntax Description

- `ppload('f', dist)`

  Loads all variables out of mat file `f`, retaining their original names. All loaded matrices will be distributed the same way, given by `dist`. A `dist` value of 1 denotes a row-distributed object, a value of 2 denotes a column-distributed object, and a value of 3 denotes a block-cyclic-distributed object.

- `ppload('f',' v1', 'v2', ...)`
- `ppload('f')`

  If `dist` is omitted, the `ddense` objects will be column-distributed by default.

- `S = ppload('f', 'v1', 'v2', ..., 'dist')`

  Defines `S` to be a `struct` containing fields that match the returned variables.

- `ppload f v1, v2, ...`

  Alternate syntax description

ppsave

```
ppsave('f', 'v1', 'v2', ...)
```

Saves the distributed objects `v1`, `v2`, ... directly to the server file `f`, each under its own name.

Function Syntax Description

- `ppsave('f')`

  If no variables are listed, saves all distributed objects currently assigned to variable names in the workspace.

- `ppsave('f', 'v1', 'v2', ..., -append)`

  Appends the variables to the end of file `f` instead of overwriting.

- `ppsave('f', 'v1', 'v2', ..., -multi)`

  Splits the variable data into one file per processor, each containing the local data for that processor.

- `ppsave f v1, v2, ...`

  Alternate syntax description

  IMPORTANT: `ppsave` will not save the contents of any local (client) objects.

Task Parallel Functions

bcast

Tag distributed object `x` as being broadcast to all of the **`ppeval`** calls.

```
y = bcast(x)
```

References

Also, see `ppeval`, and `split`.

split

Split a distributed object x along dimension `dim`. Used as input to **`ppeval`**.

```
y = split(x,dim)
```

Function Syntax Description

If `dim` == 0, then `x` is split into its elements

Example

`split(A,1)` splits A by rows

Each row is then an input to the function specified in the **`ppeval`** call.

References

Also, see `ppeval` and `bcast`.

ppeval

Execute a function in parallel on distributed data. `ppeval` is just another way of specifying iteration.

```
[o1,o2,...,on] = ppeval(f,in1,in2,...,inl)
```

Function Syntax Description

Two pieces of information are required for the call:

- The function to be executed. This is the `f` argument. It is a string containing the function name.
- The specification of the set of inputs to `f`. These are the `in1` arguments. If `f` is a function of `k` arguments then `k in1` arguments are needed. Each of these arguments are split into inputs to `f` by the following rules:
  - — If class(INl) = 'ddense', then it is split by columns = 'double', then it is broadcast (each invocation of F gets the entire argument)
    - `inl = split(ddense,d)`, then it is split along dimension `d`
    - `in1 = split(ddense,0)`, then it is split into its constituent elements
    - `in1 = bcast(a)`, then a is broadcast

NOTE: The arguments must be “conformable” in the sense that the size of each split (excluding broadcasts, of course) must be the same for all the arguments that are split. In this way we can determine the total number of calls to `f` that will be made.

The output arguments, o1, o2, ..., on are ddense or ddensend arrays representing the results of calling f. Each output argument is created by concatenating the result of each iteration along the next highest dimension; for example, if `K` iterations of `f` are performed and the output of each

iteration is a matrix of size MxN, then the corresponding output after the ppeval invocation will be a MxNxK matrix.

NOTE: Note that prior versions of Star-P * a version of ppeval that did not reshape the output arguments to ddense objects. For backward compatibility, this function is ** as ppevalsplit.

ppeval is only defined for arguments that are dense.

If f returns n output arguments then there will be n output arguments. See also `split` and `bcast`.

ppevalc

`ppevalc` is similar to `ppeval` on the client side. On the server side, the difference is that server side functions, the code which runs on the server, is written in C++ for `ppevalc`.

When writing a `ppevalc` function, everything that is placed between the input arguments and output arguments is up to the user.

Example

This example implements the cumulative sum function. It demonstrates how you can

- pass input argument data directly to an external function, and,
- have the external function write directly into an output argument without making extra copies of the data.

For output arguments, this requires that the external function in question, support being told where to write its result. In this example, the C++ standard library's partial sum function is used.

```
static void cumsum(ppevalc_module_t& module, pearg_vector_t const&
                   inargs, pearg_vector_t& outargs)
{
   // check input and output arguments
   if (inargs.size() != 1)
   {
     throw ppevalc_exception_t("Expected 1 input argument");
   }
   if (!inargs[0].is_vector())
   {
     throw ppevalc_exception_t("Expected input argument to be a
                               vector");
   }
   if (inargs[0].element_type() != pearg_t::DOUBLE)
   {
      throw ppevalc_exception_t("cumsum only supports arrays of
                                doubles");
   }
```

```
    if (outargs.size() != 1)
    {
      throw ppevalc_exception_t("Expected 1 output argument");
}
```

ppevalc Function Classes

`ppevalc` uses the following classes:

- **pearg_t**
  A class to hold input and output arguments to ppeval C++ functions. This class contains the following **constructors**:
  - scalar constructors
  - Uninitialized and initialized array constructors

  This class also contains a **data accessor** which returns a pointer to the underlying data.
- **ppevalc_module_t**
  A class that provides the interface for ppevalc modules to interact with the starpserver runtime. This class contains constructors and deconstructors.

Header Files

There are two header files included with ppevalc. The header file is used to described the main body of code to other modules.

- pearg.h
- ppevalc_module.h

pearg_t class functions

A class to hold input and output arguments to ppeval C++ functions.

Instances of this class have shallow copy, reference counted semantics, so it can be passed around by value. The data held is only freed when the last copy goes out of scope. To make a deep copy, use the **`clone()`** method.

Input Arguments

`pearg_t`

`pearg_t` `()`

Creates a null pearg_t.

`pearg_t` `clone () const`

Create a deep copy of this pearg_t.

`void` **`disown_data`** `()`

Make this pearg disown data it holds.

Public Functions

Scalar constructors

Creates an object initialized from the input argument.

```
pearg_t (starp_double_t v)
        (starp_dcomplex_t v)
        (std::string const &v)
```

The following table describes the parameters for the scalar constructors.

Table 2-3 Scalar Constructor Parameters

Type	Description
starp_double_t	An array of doubles.
starp_complex_t	A complex double constructs complex data from real and imaginary data.
std:: string const &v	An alphanumeric character string.

Uninitialized Array Constructors

Create an array object of the given dimensions and type. Memory for the array is allocated and managed by this object. The memory is uninitialized. Use `data ()` to get a pointer to the memory.

The following table describes the parameters for the uninitialized array constructors.

Table 2-4 Uninitialized Array Constructors

Type	Description
element_type	• CHAR • DOUBLE DOUBLE_COMPLEX
starp_size_t	• length • num_rows - number of rows • num_col - number of columns
starp_size_vector_t	Dimension size

Initialized Array Constructors

Create an array object backed by the array provided in argument v.

- If `is_owner` is true, then this object takes ownership of the memory pointed to by argument v and frees up the memory by using `delete []`.
- If `is_owner` is false, then it is the caller's responsibility to free the memory.

The following table describes the parameters for initialized array constructors.

Table 2-5 Initialized Array Constructor Parameters

Type	Description
element_type	• CHAR • DOUBLE DOUBLE_COMPLEX
starp_size_t	• length • num_rows - number of rows • num_col - number of columns
starp_size_vector_t	Dimension size
bool	TRUE if the pearg_t is null, meaning it was constructed with no argument constructor, therefore, it has no value.

Returns

`bool is_null () const`

Return true if this pearg_t is null. It means that it was constructed with the no argument constructor and, therefore, has no value.

`element_type_t element_type () const`

Return the object element type.

`starp_size_t element_size () const`

Return the size of one element in bytes of element_type.

`starp_size_vector_t const & size_vector () const`

Return vector containing the size of each dimension.

`starp_size_t number_of_elements () const`

Return the total number of elements in the argument.

`bool is_vector () const`

Return true if this argument is one-dimensional, or, two-dimensional with one dimension size equal to 1.

`std::string const & string_data () const`

Return a string value if this is type STRING.

Data Accessor

Returns a point to the underlying data. The template type parameter must match the `element_type` for the object.

```
template<class ElementType> ElementType * data ()
```

```
template<class ElementType> ElementType const * data () const
```

Example:

```
starp_double_t *p = arg.data<starp_double_t();
```

Constructors and Deconstructors: Input

`pearg_t::pearg_t ()` `[inline]`

Creates a null `pearg_t`. The `is_null` method will return true if this constructor is used.

Member Function: Input

```
peart_t pearg_t::clone () const
```

Create a deep copy of this pearg_t.

The copy constructor and assignment operator for this class make shallow copies, which internally refer to the same data. Use this method when you want to make a separate copy of a pearg_t.

```
template<class ElementType> ElementType const* pearg_t::data ()
const
template<class ElementType> ElementType* pearg_t::data ()
void pearg_t::disown_data () [inline]
```

Make this pearg disown the data it holds. It will not free up memory for its argument data in its destructor. This is useful if you want to get a pointer to the data using `data ()`, and, hold on to the pointer after `pearg_t` object has gone out of scope.

Returns

```
starp_size_t pearg_t::element_size () const
```

Return the size of one element in bytes of element_type.

```
element_type_t pearg_t::element_type () const [inline]
```

Return the element type of this object.

```
bool pearg_t::is_null () const [inline]
```

Return true if this `pearg_t` is null, meaning it was constructed with the no argument constructor and therefore, has no value.

```
bool pearg_t::is_vector () const [inline]
```

Return true if this argument is one dimensional, or, two dimensional with one dimension size equal to 1.

```
starp_size_t pearg_t::number_of_elements () const [inline]
```

Return the total number of elements in this argument. For strings, this will be the string length.

```
starp_size_vector_t const& pearg_t:: size_vector () const [inline]
```

Return vector containing the size of each dimension.

```
std::string const& pearg_t::string_data () const
```

Return string value if this is of type STRING.

NOTE: `ppevalc_exception_t` if `element_type` is not STRING.

ppevalc_module_t Function Class

This class provides the interface for ppevalc modules to interact with the starpserver runtime.

```
#include <ppevalc_module_h>
```

Public Member Functions: Input

```
void add_function (std::string const &func_name,
ppevalc_user_function_t func)
```

Register a new function with the module.

Public Member Functions: Return

```
std::ostream & log
```

Returns a reference to an ostream which should be used by user functions to log messages.

Member Functions

Function

```
void ppevalc_module_t::add_function (std::string const & func_name
ppevalc_user_function_t func)
```

Description

Register a new function with the module. The function pointer is accessible using the `get_function` method.

Parameters

The following table describes the parameters:

Table 2-6 Function Parameters

Type	Description
`func_name`	A string to retrieve the associated function pointer. Later it can be used as a look-up key with the `get_function` method. If another function pointer was previously registered with the same name, it will be replaced.
`func`	The function pointer to associate with "name".

Function

```
std::ostream& ppevalc_module_t::log() [inline]
```

Returns

Returns a reference to an ostream which should be used by user functions to log messages.

Example

In the following example, a cumulative sum function is implemented. This demonstrates how you can

- pass input argument data directly to an external function, and
- have the external function write directly into an output argument without making extra copies of the data.

For output arguments, this requires that the external function support being told where to write its result. In this example, the C++ standard library's partial sum function is used.

```
static void cumsum(ppevalc_module_t& module,
                   pearg_vector_t const& inargs, pearg_vector_t& outargs)
{
  // check input and output arguments
  if (inargs.size() != 1) {
    throw ppevalc_exception_t("Expected 1 input argument");
  }
  if (!inargs[0].is_vector()) {
    throw ppevalc_exception_t("Expected input argument to be a vector");
  }
  if (inargs[0].element_type() != pearg_t::DOUBLE) {
    throw ppevalc_exception_t("cumsum only supports arrays of doubles");
  }
```

```
  if (outargs.size() != 1) {
    throw ppevalc_exception_t("Expected 1 output argument");
  }

  // create an output argument of the same type and shape as the input argument
  pearg_t outarg(inargs[0].element_type(), inargs[0].size_vector());

  // call external function, telling to read its input args directly
  // from inargs, and telling it to write its result directly into the
  // outarg

  starp_double_t const* indata = inargs[0].data<starp_double_t>();
  starp_double_t *outdata = outarg.data<starp_double_t>();

  std::partial_sum(indata, indata + inargs[0].number_of_elements(),
                   outdata);

  outargs[0] = outarg;
}
```

ppevalsplit

`ppevalsplit ()`

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

Because of this potential irregularity, a `dcell` object cannot be used for much of anything until it is converted into a "normal" distributed object via the `reshape` operator. The only operators that will work on a `dcell` are those that help you figure out what to convert it into, e.g., `size`, `numel`, `length`, and `reshape`, which converts it, in addition to `ppwhos`. Luckily, you will almost never need to be aware of `dcell` arrays or manipulate them.

ppevalcsplit

`ppevalcsplit ()`

Same function as `ppevalsplit` except for the `ppevalc` modules.

ppevalcloadmodule

`NAME = ppevalcloadmodule (FNAME, NAME)`

Loads a ppevalc module on the server.

Function Syntax Description

- Input
  - FNAME refers to a valid filename on the server machine of a ppevalc module.
  - NAME refers to the modules' symbolic name. It is used when invoking module functions from the client.
- Output: The return value is the symbolic module name.

`NAME = ppevalcloadmodule (FNAME)`

Loads a ppevalc module on the server.

Function Syntax Description

- Input
  - FNAME - NAME defaults to FNAME with any leading path, and, filename extension stripped off.
- Output: The return value is the symbolic module name.

ppevalcunloadmodule

`ppevalcunloadmodule (NAME)`

Remove a previously loaded ppevalc module.

Function Syntax Description

- NAME is the symbolic name of the previously loaded module.

ppprofile

The **`ppprofile`** command collects and displays performance information for Star-P. `ppprofile` is a profiler for the Star-P server. It allows you to examine which function calls the Star-P server makes and how much time is spent in each call.

Function Syntax Description

- `ppprofile on` starts the collection of performance data about each call from the Star-P client to the Star-P server
- `ppprofile on -detail basic` has the same effect as `ppprofile on`
- `ppprofile on -detail full` also collects information about the number of changes of distribution that occur on the server, and the amount of time spent executing in the server
- `ppprofile off` stops gathering data without clearing the data that's already been collected

- `ppprofile clear` clears the collected data
- `ppprofile display` displays the data about each server call as it occurs
- `ppprofile nodisplay` displays the immediate display of data about each server call
- `ppprofile report` generates a report of the data collected so far

See “Summary and Per-Server-Call Timings with ppprofile” in the **Star-P User Guide** for examples of the usage of `ppprofile`.

Example

To turn profiling on, issue the following command:

```
>> ppprofile on
```

Then follow with the commands or scripts of interest and end with `ppprofile` report:

```
>> ppprofile on
>> a = rand(1000*p);
>> b = inv(a);
>> d = inv(a);
>> c = eig(b);
>> ppprofile report
function                    calls      time  avg time    %calls     %time
ppscalapack_eig                 1    99.295    99.295    8.3333   93.7134
ppscalapack_inv                 2     6.008     3.004   16.6667    5.6703
ppbase_createMatrixCopyRe       2     0.342     0.171   16.6667   0.32278
pppblas_ctrans                  2     0.161    0.0805   16.6667   0.15195
pppblas_gemm                    1      0.08      0.08    8.3333  0.075503
ppdense_rand                    1      0.04      0.04    8.3333  0.037752
ppdense_diag                    1      0.02      0.02    8.3333  0.018876
ppdense_binary_op               1      0.01      0.01    8.3333 0.0094379
ppdense_sumv                    1         0         0    8.3333         0
Total                          12   105.956    8.8297
```

The `ppprofile` information is ordered in columns and displays, from left to right, the server function called, the number of function calls, the time spent inside the function, the average time spent inside the function per function call, the percentage of function calls, and the percentage of time spend inside the function. For the full range of functionality of ppprofile please consult the Command Reference Guide or type help ppprofile in Star-P.

pptic/toc

`pptic/toc` provides information complementary to the MATLAB `tic/toc` command. The latter provides the wall-clock time of the instructions enclosed by the `tic/toc` statement and the former provides information on the communication between the client and the server.

The pptic/pptoc output displays:

- the number of messages and the number of bytes received by the server from the client and
- the number of messages and the number of bytes sent from the server to the client.

```
>> pptic; d = inv(a); pptoc;
Client/server communication info:
 Send msgs/bytes    Recv msgs/bytes     Time spent
 0e+000 / 0.000e+000B  3e+000 / 4.130e+002B   9.800e-002s
Server info:
  execution time on server: 5.875e+000s
  #ppchangedist calls: 1
```

In addition to the number of message and bytes received and sent, pptic/toc shows the time spent on communication and calculation and the number of distribution changes needed to accomplish the instructions enclosed by the pptic/toc statement.

The two important pieces of information contained in pptic/toc that affect performance are the bytes received and sent and the number of distribution changes.

Combining client variables and server variables in the expression will result in the movement of the client variable to the server, which will show up in the bytes received field. Since data movement is expensive, this is a possible place to enhance performance, especially if the expression happens to be located inside a looping construct. For example, compare the following two calculations:

Example 1

```
% Multiply client matrix and server matrix
>> A = rand(1000);
>> B = rand(1000*p);
>> tic; pptic; C=A*B; pptoc;toc
Client/server communication info:
  Send msgs/bytes       Recv msgs/bytes        Time spent
  0e+000 / 0.000e+000B  3e+000 / 8.000e+006B   1.160e+000s
Server info:
  execution time on server: 1.250e-001s
  #ppchangedist calls: 0
Elapsed time is 1.704369 seconds.
```

Example 2

```
% Multiply two server matrices
>> A = rand(1000*p);
>> B = rand(1000*p);
>> tic;pptic; C=A*B; pptoc;toc
Client/server communication info:
  Send msgs/bytes       Recv msgs/bytes        Time spent
  0e+000 / 0.000e+000B  2e+000 / 2.120e+002B   0.000e+000s
Server info:
  execution time on server: 1.250e-001s
```

```
  #ppchangedist calls: 0
Elapsed time is 0.234176 seconds.
```

In the first example, you see that number of bytes received by the server is exactly the size of `A`, 1000*1000*8 bytes = 8 MB, and that the communication took 1.16 sec.

In the second example, the number of bytes received is 212 or 37,000 times smaller. These 212 bytes contain the instructions to the server that specify what operations need to be performed. The penalty you pay in the first example is 1.16 sec of data transfer, which could have been prevented by creating the variable A on the server instead of on the client.

The number of distribution changes reported by `pptic/toc` indicates how often Star-P needed to make a temporary change to the distribution of a variable, for example, from row to column distributed, in order to perform a set of instructions. Distribution changes cost time and should be avoided whenever possible when optimizing code for performance (note that distribution changes become more expensive for slower interconnects between the processors, e.g., clusters). In general, keeping the distributions of all variables aligned, i.e., all row distributed or all column distributed, prevents distribution changes and improve performance.

11 Star-P User Commands

Star-P can often be used without any options, but advanced users may wish to override some of the defaults to control Star-P behavior more precisely. The main command you will probably use is the **`starp`** command, which provides the following options:

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

Hostname or address of HPC to connect to. Implies -nomcm

If optional arguments are passed, a Machine File will be generated using the list of nodes.

- `-c config config_file`
  Config file to load
- `-d,--usedefault`
  Use default HPC session instead of prompting for one
- `-f,--filter`
  Run MATLAB in filter mode so it reads from stdin and writes to stdout, for testing
- `-h,--help`
  Print this help message
- `-m,--machine machine_file_path`
  Path to a machine file to be used for this instance of starp. The file format is one machine name per line, with no empty lines at end of file. node specified by -a argument must be included in file. Example contents of this file would be: node1 node2 ... nodeN
- `-n,--nomcm`
  Run without using Administration server
- `-p,--numprocs numprocs`
  Number of processes to request. Implies -nomcm
- `-s,--starppath starp_path`
  Path to starp installation in HPC
- `-t,--datadir data_path`
  Path on HPC where distributed data will be stored
- `-u,--hpcuser`
  hpcuser username used to SSH to HPC

- `-x,--exclude [node] or [node1,node2,...,nodeN] or [node2-nodeN]`
  Exclude a node, a set of nodes or a range of nodes from the current instance of starp. This argument will be used as a modifier against either a machine file passed in using the -m argument, or against either the user's or the system's default machine file. This flag is mutually exclusive with -z.
- `-z,--use [node] or [node1,node2,...,nodeN] or [node2-nodeN]`
  Use a node, a set of nodes or a range of nodes for the current instance of starp. This argument will be used as a modifier against either a machine file passed in using the -m argument, or against either the user's or the system's default machine file. This flag is mutually exclusive with `-x`.

By providing command-line options, you can override some of the information normally supplied by the Administration Server or the starpd.properties file. This also provides the ability to run without having an Administration Server available. The following example shows the minimal set of command-line options required for running without an Administration Server. In this case, the command would cause MATLAB to start up, running eight Star-P Server processes on a machine with the hostname **altix** as the user **joe**:

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

Examples

The following is not intended to represent an exhaustive guide to all the possible command line option combinations you could choose to use, but just to provide a few examples that ISC thinks you might find useful.

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

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

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

NOTE: If node3 or node7 are not members of the default machine file, they will be ignored.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

It is also useful to understand the precedence order of potential machine files.

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

A Additional Information for ppevalc

ppevalc Application Programming Interface (API)

pearg_t Function Class

Member Enumeration

enum pearg_t::element_type_t

The user-defined enumerators are:

- CHAR
- DOUBLE
- DOUBLE_COMPLEX

Input Arguments

pearg_t

pearg_t ()

Creates a null pearg_t.

pearg_t clone () const

Create a deep copy of this pearg_t.

void **disown_data** ()

Make this pearg disown data it holds.

Public Functions

Scalar constructors

```
pearg_t (starp_double_t v)
        (starp_dcomplex_t v)
        (std::string const &v)
```

Uninitialized Array Constructors

```
pearg_t (pearg_t::element_type_t element_type,
         starp_size_t length)
pearg_t (pearg_t::element_type_t element_type,
          starp_size_t num_rows, starp_size num_cols)
```

```
pearg_t (pearg_t::element_type_t element_type,
        starp_size_vector_t const &dim_sizes)
```

Initialized Array Constructors

```
pearg_t (starp_double_t *v, starp_size_t length,
        bool is_owner=true)
pearg_t (starp_double_t *v, starp_size_t num_rows,
        starp_size_t num_cols, bool
        is_owner=true)
pearg_t (starp_double_t *v, starp_size_vector_t
        const &dim_sizes, bool is_owner=true)

pearg_t (starp_dcomplex_t *v, starp_size_t length,
        bool is_owner=true)

pearg_t (starp_dcomplex_t *v, starp_size_t
        num_rows, starp_size_t num_cols, bool
        is_owner=true)
pearg_t (starp_dcomplex_t *v, starp_size_vector_t
        const &dim_sizes, bool is_owner=true)
```

Returns

`bool is_null () const`

Return true if this pearg_t is null. It means that it was constructed with the no argument constructor and, therefore, has no value.

`element_type_t element_type () const`

Return the object element type.

`starp_size_t element_size () const`

Return the size of one element in bytes of `element_type`.

`starp_size_vector_t const & size_vector () const`

Return vector containing the size of each dimension.

`starp_size_t number_of_elements () const`

Return the total number of elements in the argument.

`bool is_vector () const`

Return true if this argument is one-dimensional, or, two-dimensional with one dimension size equal to 1.

`std::string const & string_data () const`

Return a string value if this is type STRING.

Data Accessor

```
template<class ElementType> ElementType * data ()

template<class ElementType> ElementType const * data () const
```

Example:

```
starp_double_t *p = arg.data<starp_double_t();
```

Constructors and Deconstructors: Input

```
pearg_t::pearg_t (starp_double_t v)
pearg_t::pearg_t (starp_dcomplex_t v)
pearg_t::pearg_t (std::string const & v)
pearg_t::pearg_t (pearg_t::element_type_t element_type,
                  starp_size_t length)
pearg_t::pearg_t (pearg_t::element_type_t element_type,
                  starp_size_t num_rows, starp_size_t num_cols)
pearg_t::pearg_t (pearg_t::element_type_t element_type,
                  starp_size_vector_t const & dim_sizes)
pearg_t::pearg_t (starp_double_t * v, starp_size_t length, bool
                  is_owner=true)
pearg_t::pearg_t (starp_double_t * v, starp_size_t num_rows, bool
                  is_owner=true)
pearg_t::pearg_t (starp_double_t * v, starp_size_vector_t const &
                  dim_sizes, bool is_owner=true)
pearg_t::pearg_t (starp_dcomplex_t * v, starp_size_t length, bool
                  is_owner=true)
pearg_t::pearg_t (starp_dcomplex_t * v, starp_size_t num_rows,
                  bool is_owner=true)
pearg_t::pearg_t (starp_dcomplex_t * v, starp_size_vector_t const
                  & dim_sizes, bool is_owner=true)
```

Member Function: Input

```
peart_t pearg_t::clone () const

template<class ElementType> ElementType const* pearg_t::data ()
const
template<class ElementType> ElementType* pearg_t::data ()
void pearg_t::disown_data () [inline]
```

Returns

```
starp_size_t pearg_t::element_size () const
```

Return the size of one element in bytes of element_type.

```
element_type_t pearg_t::element_type () const [inline]
```

Return the element type of this object.

```
bool pearg_t::is_null () const [inline]
```

Return true if this `pearg_t` is null, meaning it was constructed with the no argument constructor and therefore, has no value.

```
bool pearg_t::is_vector () const [inline]
```

Return true if this argument is one dimensional, or, two dimensional with one dimension size equal to 1.

```
starp_size_t pearg_t::number_of_elements () const [inline]
```

Return the total number of elements in this argument. For strings, this will be the string length.

```
starp_size_vector_t const& pearg_t:: size_vector () const [inline]
```

Return vector containing the size of each dimension.

```
std::string const& pearg_t::string_data () const
```

Return string value if this is of type STRING.

NOTE: `ppevalc_exception_t` if `element_type` is not STRING.

ppevalc_module_t Function Class

Public Member Functions: Input

```
ppevalc_module_t (std::string const &name)
```

Constructs a new module with the name “name”.

```
virtual ~ppevalc_module_t ()
std::string const & get_name () const
```

Get the module name as passed to the constructor.

```
void add_function (std::string const &func_name,
ppevalc_user_function_t func)
```

Register a new function with the module.

```
ppevalc_user_function_t get_function (std::string const
&func_name) const
```

Retrieve a function pointer by name.

Public Member Functions: Return

```
std::ostream & log
```

Returns a reference to an ostream which should be used by user functions to log messages.

Static Public Member Functions: Input

```
static smart_ptr<ppevalc_module_t> load_module (std::string const
&module_name, std:: string const &file_name
```

Load a module from a shared library file.

Constructor and Destructor

```
ppevalc_module_t::ppevalc_module_t (std::string const & name)
```

```
virtual ppevalc_module_t::~ppevalc_module_t () [virtual]
```

Member Functions

Function

```
void ppevalc_module_t::add_function (std::string const & func_name
ppevalc_user_function_t func)
```

Description

Register a new function with the module. The function pointer is accessible using the `get_function` method.

Function

```
ppevalc_user_function_t ppevalc_module_t::get_function
(std::string const & func_name) const
```

Description

Retrieve a function pointer by name.

Returns

The function pointer associated with `func_name`, or, NULL if no function was registered with that name.

Function

```
std::string const& ppevalc_module_t::get_name () const [inline]
```

Description

Get the module name as passed to the constructor.

Function

```
static smart_ptr<ppevalc_module_t>  ppevalc_module_t::load_module
(std::string const & module_name, std::string const & file_name)
[static]
```

Load a module from a shared library file.

The shared library must contain an exported function named `ppevalc_module_init`, with a signature matching `ppevalc_module_init_function_t`. A new ppevalc_module_t object will be created and passed to the module init function, and, the `init` function can register module functions with the module object.

Returns A pointer to the newly created module.

Exception `ppevalc_exception_t` on error loading the module.

Function `std::ostream& ppevalc_module_t::log() [inline]`

Returns Returns a reference to an ostream which should be used by user functions to log messages.

Let's consider a simple example. This function returns the arguments, whatever arguments are passed and then returned. All the functions in this example are staticfunctions so they should not be seen outside the module.

```
#include "ppevalc_module.h"
#include <numeric>
static void identity(ppevalc_module_t& module,
                     pearg_vector_t const& inargs, pearg_vector_t&
outargs)
{
  for (size_t i = 0; i < inargs.size(); i++)
  {
    outargs[i] = inargs[i].clone();
  }
}
```

- All module functions need to have the `ppevac_module_t` object. In this example, the module object is not used for any purpose.
- Second object is the input arguments. Each input argument is just a typedef for a standard library C++ vector. So you can use the size and the method on the input arguments. In this example, the method is clone. It is important you know the module and peart classes. See the **Star-P Command Reference Guide** for details.
- Third object is the output arguments. This argument is where you put your output or your return. This is also a vector and its already the right size. You do not append arguments to it, you just assign them. It is important to check that the output arguments are the size you expected them to be.

In this next example, an array object is being created using initialized array constructors. These constructors are used to assign values to the array. Generated values are supplied from a list of scalar values.

```
{
  if (inargs.size() != 1) {
    throw ppevalc_exception_t("Expected 1 input argument");
  }
  if (outargs.size() != 1) {
    throw ppevalc_exception_t("Expected 1 output argument");
```

```
    }

    module.log() << "Here is how to send messages to the server log: "
                 << "plusone called with " << inargs.size() << " args"
                 << std::endl;

    pearg_t inarg = inargs[0];

    pearg_t outarg(inarg.element_type(), inarg.size_vector());

    switch (inarg.element_type())
    {
    case pearg_t::DOUBLE:
      {
        double const* indata = inarg.data<double>();
        double *outdata = outarg.data<double>();

        for (size_t i = 0; i < inarg.number_of_elements(); i++)
        {
          outdata[i] = indata[i] + 1;
        }
      }
      break;

    case pearg_t::DOUBLE_COMPLEX:
      {
        starp_dcomplex_t const* indata = inarg.data<starp_dcomplex_t >();
        starp_dcomplex_t *outdata = outarg.data<starp_dcomplex_t >();

        for (size_t i = 0; i < inarg.number_of_elements(); i++)
        {
          outdata[i] = indata[i] + 1.0;
        }
      }
      break;

    default:
      throw ppevalc_exception_t("Unsupported input data type");
    }

    outargs[0] = outarg;
  }
```

- The input arguments first check the size of the array. It throws an exception if the input size does not match the output size.

- Uses standard vectors of peargs which are thin wrapper classes so you can make copies of them just by value. It does not actually copy the data. These vectors are similar to pointers.

In the following example, a cumulative sum function is implemented. This demonstrates how you can

- pass input argument data directly to an external function, and
- have the external function write directly into an output argument without making extra copies of the data.

For output arguments, this requires that the external function in question, supports being told where to write its result. In this example, the C++ standard library's partial sum function is used.

```
static void cumsum(ppevalc_module_t& module,
                   pearg_vector_t const& inargs, pearg_vector_t& outargs)
{
  // check input and output arguments
  if (inargs.size() != 1) {
    throw ppevalc_exception_t("Expected 1 input argument");
  }
  if (!inargs[0].is_vector()) {
    throw ppevalc_exception_t("Expected input argument to be a vector");
  }
  if (inargs[0].element_type() != pearg_t::DOUBLE) {
    throw ppevalc_exception_t("cumsum only supports arrays of doubles");
  }

  if (outargs.size() != 1) {
    throw ppevalc_exception_t("Expected 1 output argument");
  }

  // create an output argument of the same type and shape as the input argument
  pearg_t outarg(inargs[0].element_type(), inargs[0].size_vector());

  // call external function, telling to read its input args directly
  // from inargs, and telling it to write its result directly into the
  // outarg

  starp_double_t const* indata = inargs[0].data<starp_double_t>();
  starp_double_t *outdata = outarg.data<starp_double_t>();

  std::partial_sum(indata, indata + inargs[0].number_of_elements(),
                   outdata);

  outargs[0] = outarg;
}
```

The following example is another implementation of cumsum. This time it demonstrates how to return output of functions which do their own memory allocation for output arguments.

```
static void cumsum_copy(ppevalc_module_t& module,
                        pearg_vector_t const& inargs, pearg_vector_t& outargs)
{
```

In a continuation of this example, the input and output arguments are checked.

```
  if (inargs.size() != 1) {
    throw ppevalc_exception_t("Expected 1 input argument");
  }
  if (!inargs[0].is_vector()) {
    throw ppevalc_exception_t("Expected input argument to be a vector");
  }
  if (inargs[0].element_type() != pearg_t::DOUBLE) {
    throw ppevalc_exception_t("cumsum only supports arrays of doubles");
  }

  if (outargs.size() != 1) {
    throw ppevalc_exception_t("Expected 1 output argument");
  }
```

The next step in this example is to call an external function, but time, allocate the data for the returned data locally, to simulate a function which does its own allocation.

```
  starp_double_t const* indata = inargs[0].data<starp_double_t>();
  starp_double_t *outdata = new starp_double_t[inargs[0].number_of_elements()];

  std::partial_sum(indata, indata + inargs[0].number_of_elements(),
                   outdata);
```

Also, in this example, the outdata pointer is passed to the `pearg_t` constructor. The `pearg_t` object now takes ownership of the memory pointed to by outdata. The memory will be freed up when it is done by calling delete []. This only works if the memory was allocated with the “new” operator. In particular, no pointers to memory on the stack or static data.

```
   outargs[0] = pearg_t(outdata, inargs[0].size_vector());
```

If outdata had been statically allocated memory, which would persist after this function exits, but which `pearg_t` should not try to delete, you could pass an optional 3rd bool argument to `pearg_t`, telling it not to free the memory, as shown in the next section of this example:

```
  // outargs[0] = pearg_t(outdata, inargs[0].size_vector(), false);
}


static void my_strlen(ppevalc_module_t& module,
                      pearg_vector_t const& inargs, pearg_vector_t& outargs)
{
  if (inargs.size() != 1) {
    throw ppevalc_exception_t("Expected 1 input argument");
  }
  if (outargs.size() != 1) {
    throw ppevalc_exception_t("Expected 1 output argument");
  }
  if (inargs[0].element_type() != pearg_t::CHAR) {
    throw ppevalc_exception_t("Expected string input argument");
  }
  std::string arg = inargs[0].string_data();
  outargs[0] = pearg_t((double)arg.size());
}


extern "C"
void ppevalc_module_init(ppevalc_module_t& module)
{
  module.add_function("identity", identity);
  module.add_function("plusone", plusone);
  module.add_function("cumsum", cumsum);
  module.add_function("strlen", my_strlen);
}
```

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 *Star-P processor pool*

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 *administration server*.

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, 27

A

Administration Database, 117
Administration Server, 116, 117
architecture, 116
array bounds, 29
arrays, distributed, 82

B

bcast function, 169
block cyclic distribution, 82
block-cyclic distribution, 84

C

Client module, 116
client-server model, 2
column distribution, 83
combinatorial problems, 105
communication in Star-P, 77
  implicit, 78
  interprocessor, 80
complex numbers, 26

D

data creation routines, 28
data management, 9
Data Movement Functions, 164
ddense class, 42
dense arrays, distributed, 86
dense matrices, distributed, 82
distributed array bounds, 29
distributed arrays, creating, 27
distributed classes, 42
distributed data, 21
distributed data creation routines, 28
distributed dense arrays, 82, 86
distributed dense matrices, 82
distributed matrices, examining, 49
distributed partitioned objects (dpart), 88, 178
distributed sparse matrices, 86
dlayout class, 44

F

features, Star-P, 7
Fourier analysis, 95

G

General Functions, 160

H

hardware requirements, 10
Hilbert matrix, 49

I

image processing algorithm, 95
indexing, 30

L

Laplacian matrix, 111
library management, 7, 9
Linux client system, startup, 15
loading data, 33

M

MATLAB and Star-P, 3
MATLAB features, 60
matlab2pp function, 33, 38, 164

N

naming conventions, 21
np, 160
np function, 23, 160

P

p function, 23, 160
path counting, 112
performance monitoring, 123
pivot tables, 108
pp function, 160
pp2matlab function, 33, 50, 165
ppback, 164
ppeval application example, 102
ppeval function, 50, 102, 170
ppgetlog function, 161
ppload function, 33, 168
ppprofile function, 125, 179
ppsave function, 33, 180
ppsetoption function, 161
pptic function, 123
pptoc function, 123
ppwhos function, 163
propogation, distributed attribute, 45

R

requirements, Star-P system, 10
row distribution, 83

S

serial vectorization, 73
sparse matrices, 105
sparse matrices, distributed, 86
sparse matrices, representation in Star-P, 86
split function, 170
Star-P
  architecture, 116
  configuration, 19
  documentation support, 10
  naming conventions, 21
  server, 116
  system requirements, 10
  user commands, 183
  whose command, 22
Star-P functions
  bcast, 169
  np, 160
  p, 160

pp, 160
ppback, 164
ppchangedist, 166
ppeval, 170
ppevalc, 171
ppevalcloadmodule, 179
ppevalcsplit, 178
ppevalcunloadmodule, 179
ppevalsplit, 178
ppfront, 165
ppgetlog, 161
ppgetoption, 160
pph5read, 167
pph5whos, 163
pph5write, 166
ppload, 168
ppprofile, 179
ppquit, 163
ppsave, 168
ppsetoption, 161
pptic/toc, 180
ppwhos, 163
split, 170
startup
Linux client system, 15
Windows client system, 16
system requirements, 10

T

Task Parallel Functions, 169

U

user commands, 183

V

vectorization
serial, 73

W

whose command, 22
Windows client system, startup, 16