MPInside Reference Guide

007–5780–002

COPYRIGHT

© 2011, 2013, SGI. All rights reserved; provided portions may be copyright in third parties, as indicated elsewhere herein. No permission is granted to copy, distribute, or create derivative works from the contents of this electronic documentation in any manner, in whole or in part, without the prior written permission of SGI.

LIMITED RIGHTS LEGEND

The software described in this document is "commercial computer software" provided with restricted rights (except as to included open / free source) as specified in the FAR 52.227-19 and / or the DFAR 227.7202, or successive sections. Use beyond license provisions is a violation of worldwide intellectual property laws, treaties and conventions. This document is provided with limited rights as defined in 52.227-14.

TRADEMARKS AND ATTRIBUTIONS

SGI, Altix, and the SGI logo are are trademarks or registered trademarks of Silicon Graphics International Corp. or its subsidiaries in the United States and other countries.

Intel and Xeon are trademarks or registered trademarks of Intel Corporation. Excel, Microsoft, and Windows are registered trademarks or trademarks of Microsoft Corporation in the United States and / or other countries.

All other trademarks mentioned herein are the property of their respective owners.

Record of Revision

Version	Description
001	June 2011 Original Printing.
002	November 2013 Rewrite to support MPInside 3.6.3.

Contents

Figures

About This Manual

This publication describes SGI MPInside, which is an MPI profiling tool.

The default MPInside report, `mpinside_stats`, includes information about the time spent cumulatively in each MPI routine. This report also contains size histograms that show the amount of data transferred between ranks, in terms of both size and time.

MPInside includes many environment variables that enable you to retrieve different types of data about your application. For example:

- An MPI program's performance problems often stem from a lack of synchronization during sends and receives. MPInside can help you determine which of the MPI send/receive pairs are not executing synchronously. MPInside measures this unsynchronized time for all of the MPI ranks and for all the MPI functions involved in the application. Its reports include information about the actual speeds the MPI engine attained during send/receive communication.
- MPInside reports can include information on a branch basis. A *branch* is an MPI function with all its ancestors in the calling sequence. MPInside provides the routine name and the source file line number for all the routines that define a branch.

Related Publications

The release notes for the SGI Foundation Suite and the SGI Performance Suite list SGI publications that pertain to the specific software packages in those products. The release notes reside in a text file in the `/docs` directory on the product media. For example, `SGI-MPI-1.`*x*`-readme.txt`. After installation, the release notes and other product documentation reside in the `/usr/share/doc/packages/`*product* directory.

You might also find the following documentation to be useful:

- *Message Passing Toolkit (MPT) User's Guide*

  This manual describes the industry-standard message passing protocol as optimized for SGI computers.

- `MPInside`(3)

  This man page lists all MPInside environment variables.

Obtaining Publications

You can obtain SGI documentation in the following ways:

- You can access the SGI Technical Publications Library at the following website:

  http://docs.sgi.com

  Various formats are available. This library contains the most recent and most comprehensive set of online books, release notes, man pages, and other information.

- You can view man pages by typing `man` *title* at a command line.

Conventions

The following conventions are used throughout this document:

Convention	**Meaning**
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
`manpage`(*x*)	Man page section identifiers appear in parentheses after man page names.
variable	Italic typeface denotes variable entries and words or concepts being defined.
`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. (Output is shown in nonbold, fixed-space font.)
[ ]	Brackets enclose optional portions of a command or directive line.

...	Ellipses indicate that a preceding element can be repeated.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this publication, contact SGI. Be sure to include the title and document number of the publication with your comments. (Online, the document number is located in the front matter of the publication. In printed publications, the document number is located at the bottom of each page.)

You can contact SGI in either of the following ways:

- Send e-mail to the following address:

  techpubs@sgi.com

- Contact your customer service representative and ask that an incident be filed in the SGI incident tracking system:

  http://www.sgi.com/support/supportcenters.html

SGI values your comments and will respond to them promptly.

Chapter 1

About MPInside

This chapter contains the following topics:

About Analyzing Program Performance With MPInside

MPInside is a Message Passing Interface (MPI) profiling tool. You can use MPInside to analyze the performance of an MPI application. MPInside provides information about the MPI communications. MPInside analyzes the process send actions and the process receive actions and generates data that reveals how closely the send and receive actions are synchronized. When you analyze the data, you can determine the areas in the application upon which you want to focus your optimization efforts.

As with any performance analysis tool, remember that MPInside provides data, hints, and clues. You need an initial hypothesis of your own to guide your path of inquiry. The statistics that MPInside generates can back up or disprove your hypothesis. Your own hypothesis combined with the data from multiple MPInside runs can guide you toward analyzing and modifying your program. When you run MPInside multiple times with different settings, the different output can guide your tuning.

The information that MPInside generates includes the following:

- The size of each data request
- The number of data requests

- The size of the communicator used for MPI collective functions
- The number of times that each rank was the root of a collective function

For the preceding statistics, MPInside reports the size as the sum total for the full run.

MPInside can also provide more advanced information. For example, you can use MPInside to answer "what if" questions, such as what if the MPI environment (library and hardware) were perfect? That is, what if bandwidth were infinite and latency were zero? MPInside can also provide information about the relative lateness of a send posting with regard to the receive posting.

MPInside reports its timings on a *call stack branch*, or simply *branch*, basis. A branch is a sequence of calls. Specifically, a branch is an MPI function and all of its ancestors in the calling sequence. To analyze a program's branches, use the MPInside post processor called `MPInside_post`. For each rank, MPInside generates reports named `mpinside_clstk.`*rank*, where *rank* is the number of the rank. When you run the `MPInside_post` command, it generates reports named `mpinside_clstk_post.`*rank*, where *rank* is the number of the rank. The MPInside reports include the routine name for all the routines that define a branch. If you compile the program with `-g`, the reports also includes source line numbers.

For each CPU's branch that had a send / receive partnership with another CPU's branches, MPInside generates information about each send / receive partnership. MPInside defines each partner set with the following four numbers:

- Sending rank number
- Sending CPU branch identification
- Percentage of time accounted to the partnership, in relation to the total execution wait time of the receiving branch
- Percentage of execution wait time attributed to the lack of synchronization

About MPInside Overhead

As with all profiling tools, MPInside generates some overhead when it runs. The overhead incurred with MPInside is negilible.

For example, problems occur if the application calls the `MPI_Wait` function billions of times with a null `MPI_REQUEST_NULL` request. With a null request, the MPI library returns to the application in about 0.2 microseconds, so these calls add approximately

200 seconds. Even when MPInside runs as lightly as possible, MPInside calls the timer upon entry and exit. MPInside updates the counter based on these two calls. For one instance, it takes about 0.3 microseconds to update the counter, so this action adds approximately 300 seconds. In this case, the action of updating the counter is more intrusive and more complicated than checking if the request is null, and that is what the MPI library is doing. In cases such as this, the program incurs approximately 500 seconds for the `MPI_Wait` function, and MPInside overhead is bigger than just the MPI function itself.

Check the size and request statistics gathered during the basic run, and use that information to find the problems in the application. When you examine your program in light of the existing statistics, make sure that the program does not call `MPI_Wait` billions of times with a null request.

Obtaining Additional MPInside Information

In addition to this manual, you might want to examine the online information about MPI and MPInside.

The MPI and MPInside `man`(1) pages are as follows:

- `MPI`(1), which introduces the Message Passing Interface (MPI). This `man`(1) page is available in the SGI MPT package. This `man`(1) page is not included in the MPI standard.
- `MPInside`(3), which introduces the MPInside tool and explains the enviroment variables that you can set when you use MPInside.
- `mpiplace`(1), which is a data placement tool.

In addition, the command help ouput contains some feature and usage information.

You can type the following commands on an SGI system to retrieve extended help output:

- `pram -h`

- `MPInside_post -h`

About Using a Spreadsheet Program with MPInside

Because of the large amount of statistics that MPInside generates, SGI recommends that you use a spreadsheet program as an aid to understanding. SGI does not endorse or recommend any particular spreadsheet program, but the examples in this documentation use Microsoft Excel from the Microsoft Office 2003 program suite.

Installing MPInside and Establishing the Computing Environment

SGI distributes MPInside as part of the SGI Performance Suite. The SGI Performance Suite installation process installs MPInside along with the rest of the SGI Performance Suite software.

The following procedure establishes the computing environment and ensures that you can retrieve the MPInside `man`(1) pages.

Procedure 1-1 To establish the computing environment

1. Log into your SGI system.
2. Ensure that MPInside, at its current release level, is included in the list of directories that include executable programs.

   The following example command displays the content of the `$PATH` variable and shows that `MPInside/`*version* is not in the path:

```
% echo $PATH
/usr/lib64/mpi/gcc/openmpi/bin:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/bin/X11:
/usr/X11R6/bin:/usr/games:/opt/kde3/bin:/usr/lib/mit/bin:/usr/lib/mit/sbin
```

   The following example commands add `MPInside/3.6.3` to the `$PATH` variable and verify success:

```
% module load MPInside/3.6.3
% echo $PATH
/opt/sgi/MPInside/3.6.3/bin:/usr/lib64/mpi/gcc/openmpi/bin:/usr/bin:/bin:/usr/sbin:/sbin:
/usr/local/bin:/usr/bin/X11:/usr/X11R6/bin:/usr/games:/opt/kde3/bin:/usr/lib/mit/bin:/usr/lib/mit/sbin
```

To use a different release level, replace `3.6.3` with the release level you want to use.

3. Ensure that you can retrieve `man`(1) pages related to MPInside.

   For example, the following commands show the correct paths to MPInside `man`(1) pages in the `$LD_LIBRARY_PATH` variable and in the `$MANPATH` variable:

```
% echo $LD_LIBRARY_PATH
/opt/sgi/MPInside/3.6.1/lib:/usr/lib64/mpi/gcc/openmpi/lib64
% echo $MANPATH
/opt/sgi/MPInside/3.6.1/man:/usr/lib64/mpi/gcc/openmpi/man:/usr/share/man:/usr/local/man:
/opt/man:/usr/share/catman:/usr/catman:/usr/man
```

About the Commands that Start MPInside

You can generate an MPInside report without recompiling or relinking your program. By default, MPInside creates a report called `mpinside_stats`. When you open the `mpinside_stats` report from inside a spreadsheet program, you can see the data displayed as a series of tables. When you convert the tables to graphics, it is easy to see the amount of time the program spends on communication.

When you use SGI MPT, you can use either of the following commands to start an MPI program and generate statistics with MPInside:

- The `mpirun`(1) command, which is designed for use in interactive environments. This manual typically uses this command to show how to start program runs that request MPInside analysis. This command has the following format:

  `mpirun -np` *processes* `MPInside` *program_name* [*program_args*]

- The `mpiexec_mpt`(1) command, which is designed for use in batch environments. The `mpiexec_mpt`(1) command accepts the same arguments as the `mpirun`(1) command and has the following format:

  `mpiexec_mpt -np` *processes* `MPInside` *program_name* [*program_args*]

The arguments to the preceding commands are the same and are as follows:

- For *processes*, specify the number of ranks used by the application.
- For *program_name*, specify the name of the binary program you want to analyze. The program must have been compiled. For example: `a.out`.

- For *program_args*, specify any optional arguments that the program requires.

For example, to generate default MPInside statistics, run your compiled program and include the `MPInside` parameter on the `mpirun` command line. For example:

```
# mpirun -np 128 MPInside ./a.out arg1
```

The `mpirun`(1) and the `mpiexec_mpt`(1) commands that this topic shows are part of the SGI MPT package. Your implementation might have different commands, but you can still use the `MPInside` argument to generate MPInside statistics.

If you use the SGI MPT package, the system loads the MPI libraries by default. The example programming runs in this manual show how to use the `MPINSIDE_LIB` environment variable to load a nondefault, package-specific library.

About MPInside Terminology

The MPInside documentation and the MPInside reports use the following terms:

Function time	The time before the call to the MPI function minus time when returning to the application. This time is equal to send late time + transfer time in Figure 1-1 on page 7.
Transfer time	The time when the data is actually being transferred (see Figure 1-1 on page 7).
Function waiting time	In Figure 1-1 on page 7, this time is equal to the function time because `MPI_Recv` is a blocking function. For a nonblocking function, such as `MPI_Irecv`, function wait time is the time of the `MPI_Wait` function that "finished" the request (in the MPI sense) corresponding to this function.
Send late time (SLT)	Figure 1-1 on page 7 shows unsynchronized communication between a send / receive pair. In this communication, Rank 0 issues an `MPI_Recv` request at T_1, but Rank 1 does not begin to send the data until T_2. The time difference between T_2 and T_1 is called *send late time* (SLT). The actual send time is $T_3 – T_2$. Network latency can affect SLT and send time. In theory, there is a small amount of time between when Rank 1 starts the `MPI_Send` and when Rank 0

receives the first data item, but because this amount of time is negligible, this documentation does not address or acknowledge this time.

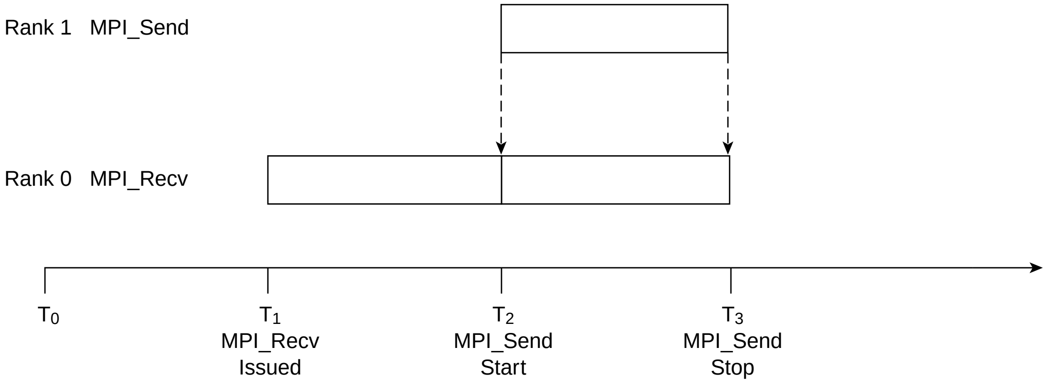


Figure 1-1 Send Late Time

Branch	A *branch* is a sequence of function calls in a user application that is terminated by an MPI function. A branch has a unique identification number. Such a number could differ from one CPU to the other even if both refer to exactly the same sequence of calls. The identification depends on the order they are encountered in the MPInside library.

For more information about branches, run the `MPInside_post` command after an MPInside run, and examine the `mpinside_clstk_post.`*rank* reports from an MPInside run.

About MPInside Environment Variables

MPInside supports many environment variables that you can use to modify the default MPInside behavior. The chapters in this manual contain examples that use these environment variables to generate different types of MPInside reports. The `MPInside`(3) man page lists all the MPInside environment variables that are available to you.

Note: Depending on the shell you use, you might need to export the environment variables after you declare them. The examples in this manual do not show an export step. Consult your shell documentation for information about how to export variables.

The following list summarizes the environment variables that are used in the examples in this documentation:

- `MPINSIDE_CALLSTACK DEPTH` *integer_number*

  Unwinds the branch call stack to a depth of *integer_number*, and writes branch information and statistics to files named `mpinside_clstk.`*rank*.

  A *branch* is an MPI function and all of its ancestors in the calling sequence. The additional reports, one for each rank number, contain information about all the branches that end with a call to an MPI routine.

- `MPINSIDE_CROSS_REFERENCE`

  When set along with `MPINSIDE_CALLSTACK DEPTH`, this variable specifies that call stack branches include data about the ranks that participated on the other end of the MPI routine. The additional cross referencing output includes the following:

  - The total communication time attributable to each partner on the other end of a given branch
  - The total send late time for each branch / partner pair

- `MPINSIDE_EVAL_COLLECTIVE_WAIT`

  Specifies that MPInside perform the following tasks:

  - Insert an `MPI_Barrier` function before all MPI collective operations
  - Record the time elapsed for the `MPI_Barrier` function

- `MPINSIDE_EVAL_SLT`

  Directs MPInside to measure the time for all send actions that are late (send late time (SLT)) compared to the `MPI_Recv`-`MPI_Wait` arrivals.

- `MPINSIDE_LITE`

  Reduces MPInside overhead to the absolute minimum. MPInside overhead is minimal in most environments, but when you use the `MPINSIDE_LITE` environment variable, MPInside reduces its overhead to very minimal levels. This mode of operation can be useful for programs that perform many, small-sized function calls. The resulting report include timings but not size or request information.

- `MPINSIDE_MATRICES`

  Directs MPInside to print the transfer topology matrix files.

- `MPINSIDE_MODEL`

  Instructs MPInside to generate statistics that model how the program would perform in an environment with zero latency, infinite bandwidth, and no time spent in the MPI routines. This output is useful because it shows how much faster a program could run if each rank did the same computational work but adopted a more efficient communication pattern or ran on a system with better networking hardware.

- `MPINSIDE_OUTPUT_PREFIX`

  Enables you to specify a custom prefix for the MPInside output report. By default, MPInside writes its report to `mpinside_stats` in the run directory. When you specify this environment variable, you can specify a full path to a different directory, or you can specify a prefix other than `mpinside`. This environment variable is useful if you want to run MPInside several times and write the report to a differently named report each time.

- `MPINSIDE_PRINT_ALL_COLUMNS`

  Prints columns of MPInside statistics with a value of zero (0) when zero values are generated. By default, MPInside suppresses columns that contain all zeros. When this variable is set, MPInside output includes all columns of statistics that pertain to the environment variables that you set. Use this variable if you want to make sure that a particular column is printed. For example, if you want to run MPInside more than once, use this variable for each run. When you set this variable, you ensure that the output contains the same columns of data for all runs.

- `MPINSIDE_PRINT_DIRTY`

  Prints data with full precision but no formatting. The report appears poorly formatted if you open it in an editor such as `vi`(1), but you can import the report into a spreadsheet with readable results.

- `MPINSIDE_SHOW_READ_WRITE`

Generates additional columns in the MPInside report. These columns show the time, number of characters, and number of calls to `libc` I/O functions such as `read()`, `write`, `open`, and `fread` that the program calls directly. If the application calls one of the MPI I/O functions, such as `MPI_File_read_at()`, setting this variable causes MPInside to include information about the `MPI_File_`*xxx* functions in an additional set of five arrays.

To set this environment variable, specify a `1` as its argument or, for extended I/O reporting, specify a string of file names. For information about how to specify the string, see `MPInside`(3).

- `MPINSIDE_SIZE_DISTRI`

  Generates a table that shows the total number of requests that the program generated at a given size for each type of MPI communication. Options to this environment variable enable you to generate a table that shows the total time spent per each request size and type of communication.

The preceding list defines each of the environment variables very briefly. For more information, see the `MPInside`(3) man page.

Chapter 2

Getting Started and Generating Default MPInside Reports

This chapter contains the following topics:

- "About Getting Started" on page 11
- "About MPInside Example Programs" on page 11
- "Analyzing a Program Using MPInside Defaults" on page 12

About Getting Started

The MPInside reports, `mpinside_stats`, `mpinside_clstk.`*rank*, and `mpinside_clstk_post.`*rank*, contain many statistics. MPInside writes these files as tab-separated text files.

Because the reports contain so many statistics, SGI recommends that you open the reports from within a spreadsheet. The example in this chapter explains how to generate a default `mpinside_stats` report and open it from within a spreadsheet.

The examples in this manual use a Windows operating system and Microsoft Excel from the Microsoft Office 2003 product suite, but you can use any spreadsheet program. On Linux platforms, you can open the reports in some spreadsheet programs by dragging and dropping the output file into an open spreadsheet program.

About MPInside Example Programs

SGI includes MPI examples and a script in the following directory:

`/usr/share/doc/packages/MPInside-`*version_number*`/examples`

The `examples` directory contains the following:

- A `README` file, which contains information about the example test provided and the output that it generates.

- The `osu_allgather` binary, which is an open source MPI latency microbenchmark.
- The `osu_allgather.c`, which is the source file for the `osu_allgather` binary.
- The `collect_osu_allgather_statistics.sh` script, which generates four files. The first is the `mpinside_stats` report. The second file contains rank-to-rank communication matricies. The script also generates two call stack reports, one for each rank.

  You can run the script interactively or you can submit it to a batch scheduler that is compatible with PBS. You can use this file to generate an MPI report for your own applications by replacing `osu_allgather` with the name of your application and setting the number of ranks appropriately.

  When you run the script, it generates output files in the current working directory.

Analyzing a Program Using MPInside Defaults

The following procedures show how to run MPInside with your program and how to obtain a simple set of output statistics:

- "Generating MPInside Statistics" on page 12
- "Opening the `mpinside_stats` Report Within a Spreadsheet" on page 13
- "Creating Graphics Within the Spreadsheet" on page 14

Generating MPInside Statistics

The MPInside analysis does not require you to recompile or relink your program. The following procedure explains how to run an MPI program and request MPInside statistics.

Procedure 2-1 To generate MPInside statistics

1. Type the following command to load the MPInside module:

   ```
   % module load MPInside
   ```

2. (Optional) Set environment variables.

MPInside supports many environment variables. The environment variables affect the MPInside output with regard to formatting, comprehensiveness, file naming, and other aspects of performance analysis.

For information about the MPInside environment variables, see the `MPInside`(3) man page.

3. Type the `mpirun` command in the following format:

   `mpirun -np` *processes* `MPInside` *program_name* `[`*program_args*`]`

   For *processes*, specify the number of ranks used by the application.

   For *program_name*, specify the name of the binary program you want to analyze. The program must have been compiled. For example: `a.out`.

   For *program_args*, specify any arguments that the program requires. These arguments are optional and are not required by MPInside.

4. Verify that the `mpirun` command finished, and locate the following file in the `mpinside_stats` working directory.

   By default, the report is named `mpinside_stats`. You can use the `MPINSIDE_OUTPUT_PREFIX` environment variable to specify a prefix other than `mpinside`.

5. Copy the `mpinside_stats` report to the computer that hosts the spreadsheet program you want to use.

Opening the `mpinside_stats` Report Within a Spreadsheet

The following procedure explains how to open the `mpinside_stats` report within Microsoft Excel 2003.

Procedure 2-2 To open the `mpinside_stats` report

1. Open the spreadsheet program.
2. Within the spreadsheet program, click **Data** > **Import External Data** > **Import Data ...** .
3. Navigate to the folder in which `mpinside_stats` resides.
4. In the **Files of type** field, select **All Files (*.*)**.
5. Select the `mpinside_stats` report, and click **Open**.

6. On the **Text Import Wizard — Step 1 of 3** pop-up window, select **Delimited**, and click **Next**.
7. On the **Text Import Wizard — Step 2 of 3** pop-up window, select **Tab**.

   Make sure that all the other boxes in this window are clear.
8. On the **Text Import Wizard — Step 3 of 3** pop-up window, accept the defaults, and click **Finish**.
9. On the **Import Data** pop-up window, make sure that **Existing Worksheet** is selected, and click **OK**.

Creating Graphics Within the Spreadsheet

MPInside creates a large volume of data. It is easier to detect problem areas in your program if you create graphics within the spreadsheet. You can create a graphic from any data set in the spreadsheet. The example in this topic creates a graphic from the data for the first array.

The following procedure explains how to create graphics from the data in the `mpinside_stats` report within a spreadsheet program.

Procedure 2-3 To create graphics

1. Locate the data for the first array.

   For example, in Microsoft Excel, press `CTRL-f`, and type **`CPU`** in the pop-up window's search field.

   This action positions the cursor in the **CPU** cell for the first array.

   Within the spreadsheet, each column head is an abbreviation for an MPI function. At the top of the report, you can see an explanation for each column head.
2. Select all the data for the first array and create a graph.

   In this step, your goal is to select all the columns from the **Compute** column through the rightmost column. The orientation of the resulting graphic should be rectangular (horizontal). For example:

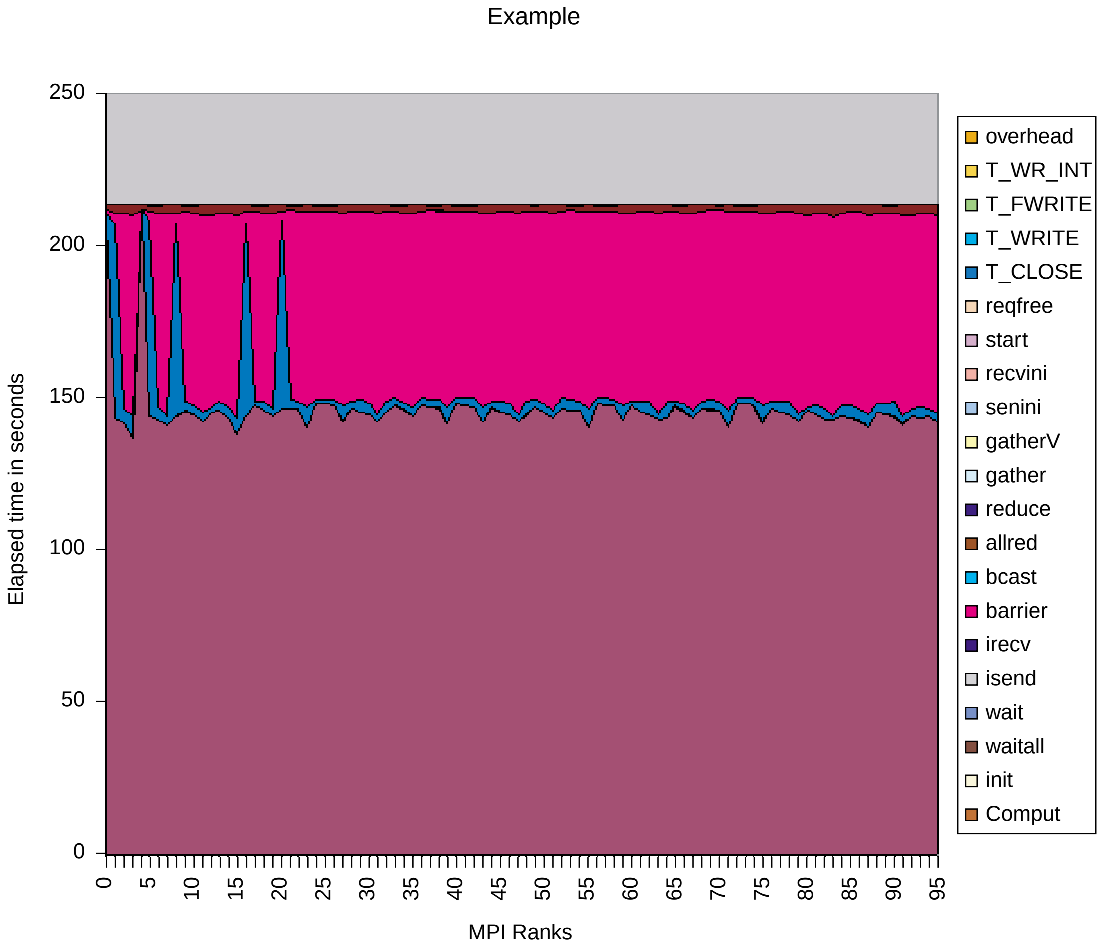


Figure 2-1 Stacked Area Plot — Running Times Per Rank of Various MPI Routines

You can drag your mouse to highlight all the rows and columns of data, but if you have a lot of data, this can be error-prone and tedious.

To highlight all the data automatically, ensure that your cursor is still in the **Compute** cell, and press the following keys simultaneously: `Shift` + `End` + `down arrow`.

If this key combination does not work automatically, press `Shift` + `End` + `down arrow` to highlight all the rows, and then press `Shift` + `End` + `right arrow` to highlight all the columns.

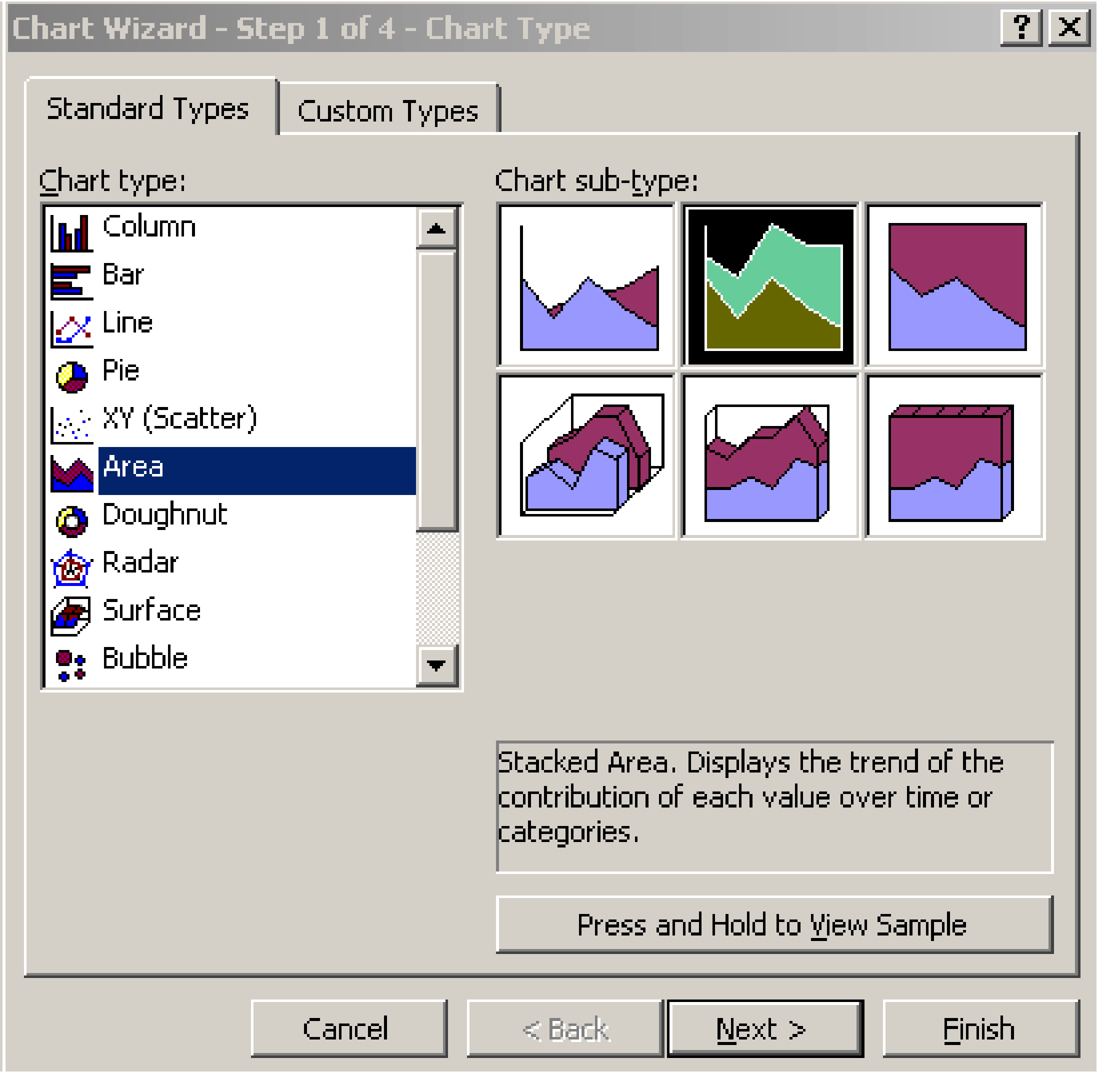
3. Click **Insert > Chart**.

   Alternatively, click the chart icon in the menu bar to create the graphic.

4. On the **Chart Wizard — Step 1 of 4 — Chart Type** pop-up window, on the **Standard Types** tab, complete the following steps:

   - In the **Chart type** field, select **Area**.
   - In the **Chart sub-type** area of the screen, click the Stacked Area icon and click **Next**.

     Figure 2-2 on page 17 shows the Stacked Area icon in the middle of the first row with a black background.



Stacked Area. Displays the trend of the contribution of each value over time or categories.

Figure 2-2 Stacked Area Selection

5. On the **Chart Wizard — Step 2 of 4 — Chart Source Data** pop-up window, under **Series in**, select **Columns** and click **Next**.

6. On the **Chart Wizard — Step 3 of 4 — Chart Options** pop-up window, complete the following fields and click **Next**:
   - In the **Chart title** field, type a name for the histogram.
   - In the **Category (X) axis** field, type a label. For example, **`MPI Ranks`**.
   - In the **Category (Y) axis** field, type a label. For example, **`Elapsed Time in Seconds`**.
7. On the **Chart Wizard — Step 4 of 4 — Chart Location** pop-up window, accept the defaults and click **Finish**.

Chapter 3

Comparing MPInside Statistics from Multiple Program Runs

This chapter contains the following topics:

About Using Statistics From Multiple Program Runs

MPInside includes environment variables that you can use to modify MPInside's default output. For example, you can use these environment variables to model a different execution environment, to generate additional statistics, or to rename the report. When you use different environment variables for each program run, each program run generates slightly different statistics. When all the program runs are complete, you can compare the statistics.

This chapter contains a large example that shows how to use different environment variables for different programming runs to generate a suite of statistics for you to examine.

Gathering Data From Multiple Program Runs

The following topics each show one part of a large example that collects statistics from an MPI program over several program runs:

- "Run 1 — Gathering Baseline Statistics" on page 19
- "Run 2 — Simulating a Perfect Interconnect Environment" on page 23
- "Run 3 — Analyzing the Amount of Time Spent Waiting" on page 25

Run 1 — Gathering Baseline Statistics

In this initial run, your goal is to gather statistics from a typical run in your typical programming environment.

The following procedure explains the environment variables to use when you run MPInside for the first time.

Procedure 3-1 To run MPInside

1. Type the following command to load the MPInside module:

   ```
   % module load MPInside
   ```

2. Rename the MPInside report.

   By default MPInside writes to the `mpinside_stats` report. If you want to run MPInside only once, there is no need to rename the report. However, in this example, you want to run MPInside multiple times and compare the results. In this case, if you permit MPInside to use the default report name, MPInside overwrites the `mpinside_stats` report in its successive runs. To preserve each successive run in a separate file, use different names for the report in each run.

   Type the following command to rename the MPInside report to `mpinside_baseline_stats`:

   ```
   % setenv MPINSIDE_OUTPUT_PREFIX mpinside_baseline
   ```

3. Type the following command to generate statistics on the request sizes:

   ```
   % setenv MPINSIDE_SIZE_DISTRI T+12:0-11
   ```

   By default, MPInside runs with the following specification: `MPINSIDE_SIZE_DISTRI 12:0--0`, and the statistics that MPInside generates show the accumulated total request sizes for all calls for rank zero (0). In this step, you specify the following:

`T+`	Generates additional statistics that describe how much time each type of MPI communication spent in transmitting or receiving a given request size. The statistics show the timings of the MPI functions, split by size. These statistics can expose bottlenecks in the program that occur for particular request sizes. The resulting statistics appear in a table with two axes, one for the communication type and one for the request size. You can look at both axes to determine the cause of an application's slowness. These statistics appear in the report after the request sizes. In the transmission time report, time spent in `MPI_Wait`, `MPI_Waitall`, `MPI_Waitany`, and `MPI_Waitsome` is added into

the row and column that corresponds to the previous nonblocking communication request, such as an `MPI_Isend`, `MPI_Irecv` call.

If the `T+` option is not supplied, the MPInside statistics do not include the time spent in calls to `MPI_wait` or `MPI_waitall`.

`12`	Specifies the number of rows in the report, excluding the row at the bottom that tabulates requests of size zero.
`0-11`	Includes statistics for ranks *first*-*last* in the report. In this example, you request statistics for 12 ranks. If you have 16 ranks and you want statistics for all ranks, specify `0-15`.

4. Type the following commands to set additional environment variables:

```
% setenv MPINSIDE_SHOW_READ_WRITE
% setenv MPINSIDE_PRINT_ALL_COLUMNS
% setenv MPINSIDE_PRINT_DIRTY
```

5. (Conditional) Set the `MPINSIDE_LIB` environment variable to your MPI implementation.

   Perform this step if your MPI implementation is something other than SGI's MPT MPI. The default setting is `MPINSIDE_LIB MPT`, which assumes that SGI MPT is your MPI implementation.

   If you use an implementation that is not SGI's MPT MPI implementation, type the one command from the following list that pertains to your implementation:

Command	**MPI Implementation**
`%` **`setenv MPINSIDE_LIB IMPI`**	X86 Intel MPI
`%` **`setenv MPINSIDE_LIB HPMPI`**	X86 HP MPI
`%` **`setenv MPINSIDE_LIB OPENMPI`**	OpenMPI

6. Type the `mpirun` command in the following format:

   `mpirun -np` *processes* `MPInside` *program_name* [*program_args*]

   For *processes*, specify the number of ranks used by the application.

   For *program_name*, specify the name of the binary program you want to analyze. The program must have been compiled. For example: `a.out`.

   For *program_args*, specify any arguments that the program requires. These arguments are optional and are not required by MPInside.

7. (Conditional) Repeat this programming run with a smaller set of environment variables.

   Perform this step only if the preceding run completed in an excessively long period of time and you suspect that MPInside introduced overhead.

   In most cases, MPInside incurs negligble overhead. However, if you notice that your program's run took noticeably longer to complete when MPInside was invoked, you might want to get an additional run. In this additional run, invoke MPInside with only minimal environment variables.

   Make sure to retain the `mpinside_baseline_stats` report file. You do not want to overwrite `mpinside_baseline_stats` because it includes important information about the size and number of requests for which overhead does not matter. Type the following commands to repeat the programming run and request minimal MPInside operations:

   - Load the MPInside module:

     ```
     % module load MPInside
     ```

   - Specify the report name:

     ```
     % setenv MPINSIDE_OUTPUT_PREFIX mpiniside_lite
     ```

   - Specify minimal overhead:

     ```
     % setenv MPINSIDE_LITE
     ```

   - (Conditional) Specify your MPI library:

     ```
     % setenv MPINSIDE_LIB lib
     ```

     This step is not needed if your library is SGI MPT MPI. The default *lib* is `MPT`. An earlier step in this procedure shows the nondefault *lib* specifications.

   - Run the program with MPInside:

     ```
     % mpirun -np processes MPInside program_name [program_args]
     ```

   If this run, with `MPINSIDE_LITE` specified, is still noticeably longer than a run without MPInside involvement, you need to consider programming problems.

   With a null request, the MPI library could return to the application in tens or hundreds of nanoseconds. For such calls, MPInside's accounting can take more processing time than the actions of the MPI library that you wanted to track. If these calls make up a substantial amount of the total MPI calls in your program, you might end up with an unrealistically long running time due to MPInside overhead, even when running in lite mode.

Run 2 — Simulating a Perfect Interconnect Environment

If the programming environment had a perfect network and perfect hardware, you might expect all message passing to occur perfectly, with no waiting. When you complete this run, you simulate a perfect environment. This run simulates the amount of waiting that occurs because of unbalanced loads, and that is independent of the MPI engine.

The following procedure explains the environment variables to use when you run MPInside for the second time.

Procedure 3-2 To run MPInside

1. Type the following command to load the MPInside module:

   ```
   % module load MPInside
   ```

2. Type the following command to rename the MPInside report to `mpinside_perfect_stats`:

   ```
   % setenv MPINSIDE_OUTPUT_PREFIX mpinside_perfect
   ```

3. Type the following command to specify the modeling of a perfect execution envionment:

   ```
   % setenv MPINSIDE_MODEL PERFECT+1.0
   ```

   This environment variable uses the following parameter:

`PERFECT+1.0`	The `1.0` specifies that you want the CPU to run in its typical mode, as you would expect it to run. If you set this higher, for example, to `1.2`, MPInside simulates a CPU that is 20% faster.

If you set this lower, for example, to `0.8`, MPInside simulates a CPU that is 20% slower, or 80% of its typical speed.

4. Type the following command to direct MPInside to print the transfer topology matrix files:

   ```
   % setenv MPINSIDE_MATRICES EXA:-B:S
   ```

   This environment variable uses the following parameters:

`EXA`	Includes the exact point-to-point transfers implied by the collective functions in the matrix files. Specify this parameter only when running with the SGI MPT MPI library. If you load your program with libraries other than the SGI MPT MPI library, use the `PLA` parameter.
`-B`	Include matrix files in the output in binary format only.
`S`	Separates the collective functions and the point to point matrices in the binary output.

5. (Conditional) Direct MPInside to merge collectives and point-to-point matrices into binary files.

   Perform this step if you did not use the SGI MPT MPI libraries to compile your program.

   The command is as follows:

   ```
   % setenv MPINSIDE_MATRICES P2P:-B:M
   ```

6. Type the following command to set additional environment variables:

   ```
   % setenv MPINSIDE_SHOW_READ_WRITE
   % setenv MPINSIDE_PRINT_ALL_COLUMNS
   % setenv MPINSIDE_PRINT_DIRTY
   ```

7. (Conditional) Set the `MPINSIDE_LIB` environment variable to your MPI implementation.

   Perform this step if your MPI implementation is something other than SGI's MPT MPI. The default setting is `MPINSIDE_LIB MPT`, which assumes that SGI MPT is your MPI implementation.

   If you use an implementation that is not SGI's MPT MPI implementation, type the one command from the following list that pertains to your implementation:

Command	**MPI Implementation**
`% **setenv MPINSIDE_LIB IMPI**`	X86 Intel MPI
`% **setenv MPINSIDE_LIB HPMPI**`	X86 HP MPI
`% **setenv MPINSIDE_LIB OPENMPI**`	OpenMPI

8. Type the `mpirun` command in the following format:

   `mpirun -np` *processes* `MPInside` *program_name* [*program_args*]

   For *processes*, specify the number of ranks used by the application.

   For *program_name*, specify the name of the binary program you want to analyze. The program must have been compiled. For example: `a.out`.

   For *program_args*, specify any arguments that the program requires.

   Do not try to compare the statistics in this programming run's `mpinside_perfect_stats` report with the true time elapsed during the programming run. The MPInside times are simulated times, and MPInside performs more computations when it simulates a perfect environment

   For example, if you pass the entire `mpirun` command in this step to the `time`(1) command, the `time`(1) command returns the time elapsed during this MPInside programming run. Do not compare the timings in `mpinside_perfect_stats` with the output from the `time`(1) command.

Run 3 — Analyzing the Amount of Time Spent Waiting

In a perfect environment, CPUs would work constantly. The CPUs would pass data to each other as smoothly as a runner in a relay race passes a baton to the next runner in line. However, a CPU sometimes has to wait until the information in another CPU is available to be transferred. The run in this topic enables you to analyze the amount of time the CPUs spend waiting.

The following procedure explains the environment variables to use when you run MPInside for the wait analysis.

Procedure 3-3 To run MPInside

1. Type the following command to load the MPInside module:

```
% module load MPInside
```

2. Type the following command to rename the MPInside report to `mpinside_slt_stats`:

   ```
   % setenv MPINSIDE_OUTPUT_PREFIX mpinside_slt
   ```

3. Type the following commands to set environment variables:

   ```
   % setenv MPINSIDE_EVAL_COLLECTIVE_WAIT
   % setenv MPINSIDE_EVAL_SLT
   % setenv MPINSIDE_SHOW_READ_WRITE
   % setenv MPINSIDE_PRINT_ALL_COLUMNS
   % setenv MPINSIDE_PRINT_DIRTY
   ```

4. (Conditional) Set the `MPINSIDE_LIB` environment variable to your MPI implementation.

   Perform this step if your MPI implementation is something other than SGI's MPT MPI. The default setting is `MPINSIDE_LIB MPT`, which assumes that SGI MPT is your MPI implementation.

   If you use an implementation that is not SGI's MPT MPI implementation, type the one command from the following list that pertains to your implementation:

Command	**MPI Implementation**
`% setenv MPINSIDE_LIB IMPI`	X86 Intel MPI
`% setenv MPINSIDE_LIB HPMPI`	X86 HP MPI
`% setenv MPINSIDE_LIB OPENMPI`	OpenMPI

5. Type the `mpirun` command in the following format:

   `mpirun -np` *processes* `MPInside` *program_name* [*program_args*]

   For *processes*, specify the number of ranks used by the application.

   For *program_name*, specify the name of the binary program you want to analyze. The program must have been compiled. For example: `a.out`.

   For *program_args*, specify any arguments that the program requires.

Chapter 4

Using MPInside to Analyze Only Parts of a Program

This chapter contains the following topics:

- "About Analyzing Subsets of a Program" on page 27
- "Analyzing Subsets of a Program" on page 27

About Analyzing Subsets of a Program

In some cases, you do not need to, or want to, run MPInside against the entire application from `MPI_Init()` to `MPI_Finalize()`. This might be the case if you have already used MPInside to isolate programming problems and have determined the parts of your program that you want to change. You can also use the procedure in this chapter if you are responsible for only parts of a program, not the entire program.

To create a targeted MPInside analysis, insert MPInside calls into your program and recompile your program. When the program runs, MPInside generates statistics for only the parts of the program that require deeper analysis. The set of calls you need to use depends on how much of your program you want MPInside to analyze. This chapter includes an example that shows you how to enable and disable MPInside analysis.

Analyzing Subsets of a Program

The following procedure explains how to run MPInside with a reduced amount of analysis.

Procedure 4-1 To reduce the amount of MPInside analysis

1. Determine the scope of the analysis that is needed.

   Does one large part of your program require analysis? You can divide a program like this into three areas, as follows:

   Initialization — no analysis needed
   Computation — analysis needed
   Finalization — no analysis needed

Do many small parts of your program require analysis? You can divide a program like this into two or more small areas, as follows:

Initialization — no analysis needed
Computation phase 1 — analysis needed
Computation phase 2 — no analysis needed
Computation phase 3 — analysis needed
Computation phase 4 — no analysis needed
Computation phase 5 — analysis needed
Finalization — no analysis needed

2. Open your program file and insert function calls to MPInside.

   For a program with one large part that needs analysis, insert calls as follows:

   - For a C program:

     Initialization
     `(void) mpinside_start();`
     Computation — analysis needed
     `(void) mpinside_end();`
     Finalization

   - For a Fortran program:

     Initialization
     `Call mpinside_start`
     Computation — analysis needed
     `Call mpinside_end()`
     Finalization

   For a program with many small parts that need analysis, insert calls as follows:

   - For a C program:

     Initialization — no analysis needed
     `(void) mpinside_start();`
     Computation phase 1 — analysis needed
     `mpinside_suspend()`
     Computation phase 2 — no analysis needed
     `mpinside_resume()`
     Computation phase 3 — analysis needed
     `mpinside_suspend()`
     Computation phase 4 — no analysis needed

```
mpinside_resume()
Computation phase 5 — analysis needed
(void) mpinside_end();
Finalization — no analysis needed
```

- For a Fortran program:

```
Initialization — no analysis needed
Call mpinside_start
Computation phase 1 — analysis needed
Call mpinside_suspend()
Computation phase 2 — no analysis needed
Call mpinside_resume()
Computation phase 3 — analysis needed
Call mpinside_suspend()
Computation phase 4 — no analysis needed
Call mpinside_resume()
Computation phase 5 — analysis needed
Call mpinside_end()
Finalization — no analysis needed
```

Note the following regarding the function calls:

- The `mpinside_start()`, `mpinside_end()`, `mpinside_suspend()`, and `mpinside_resume()` calls must involve all ranks. That is, these calls must be `MPI_COMM_WORLD` collective calls, or you may experience unexpected results.
- MPInside's analysis ends when it encounters an `mpinside_end()` call. If the program calls `MPI_Finalize()` before it calls `mpinside_end()`, the program ends as expected, and the MPInside report contains statistics through the `MPI_Finalize()` call.
- If your program includes an `mpinside_suspend()` call toward the end, but does not includes an `mpinside_end()` call, the analysis continues from the last `mpinside_suspend()` call through to the `MPI_Finalize()` call.

3. Compile the program.
4. Link the program.

   Make sure that the MPInside environment is properly set in order to be able to link your program with `libMPInside_stub.so`.

   The example that follows this procedure contains a link step.

5. Type the following command to set the `MPINSIDE_PARTIAL_EXPERIMENT` environment variable:

   ```
   % setenv MPINSIDE_PARTIAL_EXPERIMENT
   ```

   This environment variable ensures that MPInside starts its analysis after it encounters the `mpinside_start()` call.

   For information about environment variables, see the `MPInside`(3) man page.

6. (Conditional) Set the `MPINSIDE_LIB` environment variable to your MPI implementation.

   Perform this step if your MPI implementation is something other than SGI's MPT MPI. The default setting is `MPINSIDE_LIB MPT`, which assumes that SGI MPT is your MPI implementation.

   If you use an implementation that is not SGI's MPT MPI implementation, type the one command from the following list that pertains to your implementation:

Command	MPI Implementation
`% setenv MPINSIDE_LIB IMPI`	X86 Intel MPI
`% setenv MPINSIDE_LIB HPMPI`	X86 HP MPI
`% setenv MPINSIDE_LIB OPENMPI`	OpenMPI

7. Type the following command to run your program:

   ```
   % mpirun -np processes program_name program_args
   ```

   Example. The following commands link and run a C++ program that uses the SGI MPT MPI library:

   - Type the following command to load SGI's MPT module:

     ```
     % module load mpt
     ```

   - Type the following command to load the MPInside module:

     ```
     % module load MPInside
     ```

     This step ensures that `LD_LIBRARY_PATH` includes the MPInside library's directory.

- Type the following command to invoke the Intel C++ Compiler:

  ```
  % icc -o prog_with_stub prog.c -l mpi -l MPInside_stub
  ```

- Type the following command to run the program:

  ```
  % mpirun -np 128 ./prog_with_stub args
  ```

- Type the following command to run the program with MPInside:

  ```
  % mpirun -np 128 MPInside ./prog_with_stub args
  ```

Chapter 5

Analyzing Call Stack Branches and Program Stiffness

This chapter contains the following topics:

- "About Call Stack Branches and Program Stiffness" on page 33
- "Interpreting the Call Stack Branch Output" on page 33
- "Communication Stiffness" on page 39
- "Generating Statistics to Analyze Call Stack Branches and Program Stiffness" on page 42

About Call Stack Branches and Program Stiffness

Certain environment variables enable you to generate call stack information and to generate information about program *stiffness.* A program is said to be *stiff* when it contains serialized communication dependencies, and MPInside output can help you analyze these dependencies.

This chapter explains how to interpret the statistics that describe call stack branches and stiffness. This chapter also contains an example that shows the programming runs used when gathering these statistics.

Note: Some of the output examples in this chapter are very wide. To accommodate inclusion in this documentation, some have been wrapped, and column alignment in examples might differ from your output.

Interpreting the Call Stack Branch Output

When you use the `MPINSIDE_CALLSTACK_DEPTH` environment variable, the MPInside report contains call stack information. Each `mpinside_clstk.`*rank* file is a call stack branch report. These reports list call stack branch information. Each branch consists of an MPI function, followed by all of its call stack ancestors. The report sorts the branches and provides data about the time spent in a particular MPI function.

The reports can also contain information about branch partners. The following topics explain the call stack branch reports:

- "Opening the Call Stack Branch Report" on page 34
- "Branch Statistics" on page 34
- "Ancestor Information" on page 35
- "Partner Information" on page 35

Opening the Call Stack Branch Report

Like the MPInside statistics report (`mpinside_stats`), the call stack branch report (`mpinside_clstk.`*rank*) is a tab-separated report that contains a very large amount of data. SGI recommends that you open the `mpinside_clstk.`*rank* file from a spreadsheet. For information about how to open this file in a spreadsheet, use the information in the following topic:

"Opening the `mpinside_stats` Report Within a Spreadsheet" on page 13

Branch Statistics

The call stack branch contains columns of data for each function. You can interpret the column headings as follows:

Column Heading	**Meaning**
`MPI_FUNCTION`	The function name. For example, `MPI_Allreduce` or `MPI_Send`.
`Branch ID`	The unique branch identification number.
`Receive Time(s)`	The time spent on the receive function itself.
`Self%`	The percentage of the total execution time that is accountable to this branch. Specifically, this is a percentage accounting of how much time was spent on a certain branch out of the run time of the whole program.
`Self totals`	The sum of the `Self%` for all earlier branches plus the `Self%` for the current branch.

`#Send reqs`	The number of send requests from this branch.
`#Recv reqs`	The number of receive requests from this branch.
`Ave MBs sent`	Average data amount sent, in Mbytes, from this branch. For this column, and for the `Ave MBs received` column, the exact meaning depends on the specific MPI function. See the preamble of the MPInside report for information specific to the MPI function.
`Ave MBs received`	Average data amount received, in Mbytes, from this branch. For this column, and for the `Ave MBs sent` column, the exact meaning depends on the specific MPI function.
`Ave partner wait time(s)`	The `Receive Time(s)` column shows the time spent for the receive function itself. The `Ave partner wait time(s)` column shows the wait time associated with a request. For a function like `MPI_Recv()`, these two times are equal. For a function like `MPI_Irecv()`, these two times are not equal.

Ancestor Information

If the `MPINSIDE_CROSS_REFERENCE` environment variable is set, the call stack branch output contains an additional line that appears after the columns of timing data. This additional line is headed by the keyword `Ancestors`. If the application was compiled with the `-g` option, MPInside prints the line number for each function at the end of the line for each routine in the `Ancestors` section.

Partner Information

Some branches have *partners*. Branch partners consist of complementary pairs of operations. The partner for a particular call stack is the corresponding call stack branch in one or more other ranks. The call stack in the other rank performs the corresponding action in that other rank. The partners are two halves of a communication pair.

For example, assume that one application uses three ranks, and the ranks contain the following calls:

- Rank 0

  `Main > func_a > MPI_Recv`

  `Main > func_b > MPI_Recv`
- Rank 1
- Rank 2

MPInside tracks the following for these calls:

- Two call stacks, one for each `MPI_Recv` that is issued from Rank 0.
- Two, three, or four branch partners. Rank 0 issues the `MPI_Recv` functions. Rank 0's `MPI_Recv` calls could receive data from Rank 1, from Rank 2, or from both Rank 1 and from Rank 2. MPInside reports all branch partners, so the statistics report could show the following branch partners:
  - `Main > func_a > MPI_Recv` and the returned data from Rank 1
  - `Main > func_a > MPI_Recv` and the returned data from Rank 2
  - `Main > func_b > MPI_Recv` and the returned data from Rank 1
  - `Main > func_b > MPI_Recv` and the returned data from Rank 2

If rank 0 sends data to rank 1, then rank 0 has a call stack that ends with an `MPI_Send` that partners with an `MPI_Recv` call in rank 1.

The MPInside call stack branch output contains partner information, where appropriate, in the following format:

A:#*B*:*C*:*D*

A	The rank number of the partner that initiated the `MPI_Send` or `MPI_Isend` for this branch.
B	The `MPI_Send` or `MPI_Isend` branch identifier (ID) of the partner. You can find this ID in the `mpinside_clstk.`*rank* report.
C	The percentage of receive or wait time that MPInside can attribute to rank *A* and MPI send branch #*B*.

D	The percentage of `MPI_Recv` time for which the corresponding send arrived late (send late time) in *C*.

The partners connect the wait/receive branches to their corresponding request/send branches. MPInside generates partner information for the `MPI_Recv`, `MPI_Wait`, and `MPI_Test` functions. Partners for these functions are always `MPI_Send`, `MPI_Isend`, and so on. It is possible for an `MPI_Recv`, `MPI_Wait`, or `MPI_Test` branch to have several partners.

The following topics describe partners in more detail:

- "Branches With Partners" on page 37
- "Branches Without Partners" on page 37
- "Examples" on page 38

Branches With Partners

Wait branches connect, as partners, the send/recieve branches that initiate MPI requests. For all the send/receive branches that were connected to it, each wait branch reports the percentage of function waiting time to account to a particular send/receive branch in regard to the total execution time of a particular wait branch. For example, an `MPI_Wait` branch is a wait branch as well as a `MPI_Recv` branch.

Receive branches have send partners and are targets of wait branches. Each receive branch reports, for all the send branches that were with it, as follows:

- The ranks of the sends
- The send branch IDs
- The percentage of execution time (function waiting time) to account to this particular send branch in regard to the total wait time of this `Recv` branch.
- The percentage of time (send late time) such send branches were arriving late in regard to the matching receive posting.

Branches Without Partners

Some call stacks do not have partners.

Ordinary branches do not have partners nor are they targets of another branch. Collective function branches are of this type.

Send branches do not have partners. Send branches are targets of receive branches or wait branches.

Examples

The following two examples show MPInside output and include partner information after the ancestor information. For more information about this kind of output, see "Run 4 — Examining the Call Stack Branches" on page 55.

Note: The following output examples are very wide. The rightmost two columns are wrapped and shown below the main body of the output for inclusion in this documentation.

Example 1. In the following output, in the `Partners` line at the end, the first `100.00` indicates that the branch spent 100% of its time partnering with rank `0`, branch `2`. The second `100.00` in this line indicates that 100% of the time was spent waiting on a late send.

```
MPI_FUNCTION  Branch ID  Receive Time(s)  Self%  Self totals  #Send reqs  #Recv reqs  Ave MBs sent
   MPI_Recv   #258       2.003            39.72  39.7         0           1           0
Ancestors: level_2  /home/bryce6/dthomas/MPInside/TESTS/example.c:20
            level_1  /home/bryce6/dthomas/MPInside/TESTS/example.c:38
             main  /home/bryce6/dthomas/MPInside/TESTS/example.c:56
              __libc_start_main  ??:?
Partners_1_0:   0:#2:100.00:100.00

                                                  ## The last two columns of output are as follows:
                                                   Ave MBs received  Ave partner wait time(s)
                                                   0                 2.003513
```

Example 2. In the `MPI_Recv` in following output, the report shows that this branch, with the `level_2()` call on line `20`, was a partner to the matching `MPI_Send` from rank `0`, branch `1`. This `MPI_Send` was executed following the second call (line `20`) of the `level_2()` routine. This partnership accounted for 100% of the `MPI_Recv` branch, and 99.88% of the time was just wait.

```
MPI_FUNCTION  Branch ID  Receive Time(s)  Self%  Self totals  #Send reqs  #Recv reqs  Ave MBs sent
   MPI_Recv   #257       1.026            20.34  60.1         0           1           0
```

```
Ancestors: level_2  /home/bryce6/dthomas/MPInside/TESTS/example.c:20
            level_1  /home/bryce6/dthomas/MPInside/TESTS/example.c:37
             main  /home/bryce6/dthomas/MPInside/TESTS/example.c:56
              __libc_start_main  ??:?
Partners_1_0:   0:#1:100.00:99.88

                                                            ## The last two columns of output are as follows:
                                                             Ave MBs received  Ave partner wait time(s)
                                                             0                 1.027300
```

Communication Stiffness

Communication stiffness measures an application's sensitivity to point-to-point communication dependency chains. Stiffness is a conceptual rating of how sensitive the running time of each rank is to waiting for other ranks to send the data that the rank needs to proceed.

MPInside measures the communication stiffness as a proxy for the performance effect of dependent communication chains in an MPI program. The stiffness rating is not an absolute timing; stiffness does not describe, in terms of wall clock time, how fast a given application can run. An application's communication strategy is *stiff* if the ranks spend an unnecessary amount of time waiting for data before the ranks can proceed.

For example, consider a physical simulation that involves a rectangular region of space. The rectangle is a region of space composed of a one-dimensional set of *n* cubes, such that each of *n* ranks is responsible for the communication and computation that pertains to the particles in one cube. The following are different communication strategies:

- A communication strategy with an optimal (low) stiffness rating is one in which (1) a rank containing a particle moving out of its physically contained region sends that particle's data directly to the other rank that will contain the particle in the next time step and (2) each rank receives data only from ranks that will send them particles in this time step.
- In a less optimal strategy, rank 0 sends data about all of its outgoing particles to rank 1 before rank 1 can send data about its particles to rank 2, and then from rank 2 to rank 3, and so on. In this case, rank *n*-1 sends data to rank 0 before the end of the time step. With this programming strategy, each time step involves a

chain of *n*–dependent MPI communications, which gives the application a higher stiffness rating.

Applications that include dependency chains that affect performance have the following characteristics:

- Poor scalability
- Transfers that introduce load imbalances

MPInside uses two numbers to calculate the stiffness rating. The first number is the SDC counter, which is the *Size of the Dependency Chain*. MPInside keeps an SDC counter for each rank. The second number is the TNSR counter. The TNSR counter is the sum of the *Total Number of Sends and Receives*. The TNSR number reflects the number of point-to-point operations performed by the rank. These numbers form the stiffness rating, as follows:

- When a send occurs, MPInside increments the SDC counter for that rank by one and includes the new value in the message header.
- When a receive completes, MPInside increments the SDC counter for the receiving rank by one. If the new value for the receiving rank is lower than the SDC value of the sending rank, MPInside assigns the SDC value of the sending rank to the receiving rank.
- The stiffness rating is the ratio of SDC/TNSR.

  When you run the program with the following environment variables, the MPInside report includes a stiffness rating:

  – `MPINSIDE_EVAL_SLT`

  – `MPINSIDE_MODEL PERFECT+1.0`

  A stiffness rating of 1 is good and signifies very little program stiffness. Higher numbers indicate a higher probability of dependency chains in the program and worsening program stiffness.

Figure 5-1 on page 41 shows a program with an acceptable, or good, stiffness rating of 1. The value within each box shows SDC changes for each send/receive operation. The TNSR for each rank is 3.

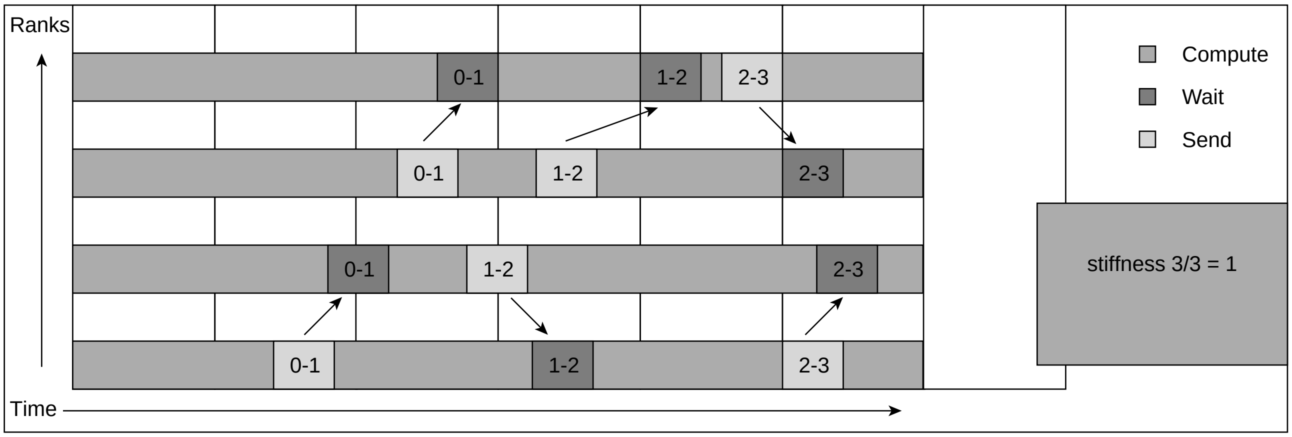


Figure 5-1 Output for a Program With a Low Stiffness Rating

Figure 5-2 on page 41 shows a program with scalability problems. This program includes a bottleneck in that a token is passed back and forth. This program has a higher stiffness rating of 3.

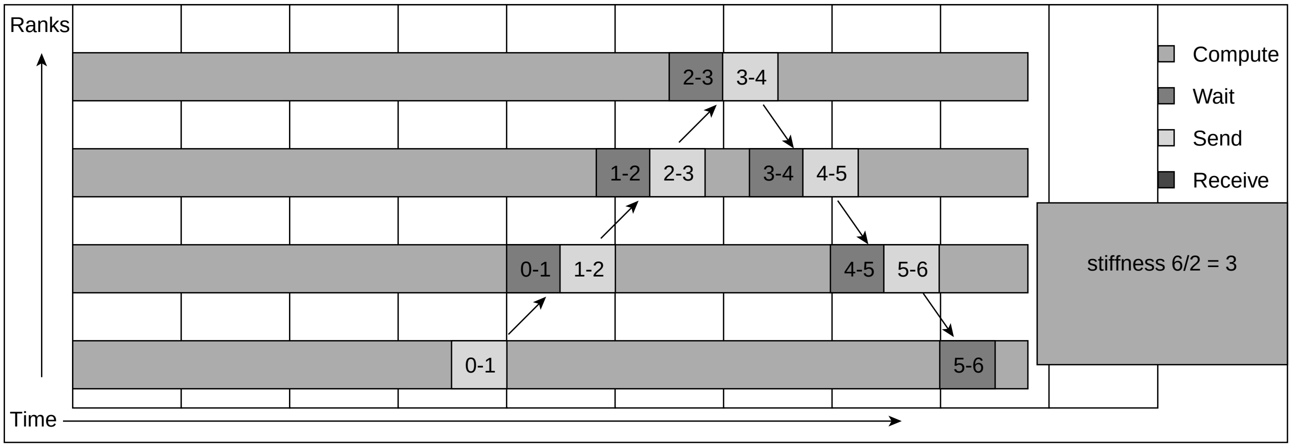


Figure 5-2 Output for a Program With a High Stiffness Rating

Generating Statistics to Analyze Call Stack Branches and Program Stiffness

The following topics contain procedures that explain how to use the stiffness data in the MPInside report. The topics are as follows:

- "Run 1 — Obtaining Baseline Statistics" on page 42
- "Run 2 — Simulating a Perfect Interconnect Environment" on page 50
- "Run 3 — Evaluating Send Late Time" on page 52
- "Run 4 — Examining the Call Stack Branches" on page 55

Run 1 — Obtaining Baseline Statistics

The first run collects baseline statistics about how the program performs in a typical programming environment.

The following procedure explains the MPInside output for a particular C program called `example`.

Procedure 5-1 To create the program and generate output

1. Use a text editor to create the following C program, called program `example`:

```
#include
#include
#include
#include

#define BUFF_SZ 1024
int buff_S[BUFF_SZ], buff_R[BUFF_SZ];
int dest, src,me, World_size;


void level_2(int load_unbalance)
{
   MPI_Status status;

   if(me % 2 == 0 ) {
     (void) sleep(load_unbalance);
     (void) MPI_Send (buff_S, BUFF_SZ/2, MPI_INT,
            dest, 13, MPI_COMM_WORLD);
```

```
   }
   else { // note Receive size is 2 time send size
     (void) MPI_Recv(buff_R, BUFF_SZ,MPI_INT,
            src, 13, MPI_COMM_WORLD, &status);
     (void) sleep(load_unbalance);
   }

   (void) MPI_Allreduce(buff_S, buff_R, BUFF_SZ,
            MPI_INT, MPI_SUM, MPI_COMM_WORLD);

   if(me == 0) (void) sleep(1);
   (void) MPI_Bcast(buff_S, BUFF_SZ/4, MPI_INT,
           0, MPI_COMM_WORLD);

}

void level_1()
{
   level_2(1);
   level_2(2);
}

int
main(int argc, char **argv)
{

    (void) MPI_Init(&argc,&argv);

    (void) MPI_Comm_size(MPI_COMM_WORLD, &World_size);
    (void) MPI_Comm_rank(MPI_COMM_WORLD, &me);

    bzero(buff_S, BUFF_SZ * sizeof(int));

    dest = me + 1;
    src = me - 1;

    level_1();

    (void) MPI_Finalize();
}
```

Note the following about the preceding program:

- This program runs a cascade of calls and then calls several MPI functions in the `level_2()` routine.
- After the program runs, MPInside writes the `mpinside_stats` report. The information in the `mpinside_stats` report about the work in routine `level_2(int load_unbalance)` shows that the `sleep()` routine calls introduce a huge load imbalance to the computation and add to the MPI function time.

2. Type the following commands to compile the program and to generate baseline statistics:

```
% gcc -g -o example example.c -lmpi
% setenv MPINSIDE_ECHO_INPUT
% setenv MPINSIDE_PRINT_DIRTY
% setenv MPINSIDE_ADD_COLUMN_MEANING
% setenv MPINSIDE_SIZE_DISTRI T+12:0-3
```

3. (Conditional) Set the `MPINSIDE_LIB` environment variable to your MPI implementation.

   Perform this step if your MPI implementation is something other than SGI's MPT MPI. The default setting is `MPINSIDE_LIB MPT`, which assumes that SGI MPT is your MPI implementation.

   If you use an implementation that is not SGI's MPT MPI implementation, type the one command from the following list that pertains to your implementation:

Command	**MPI Implementation**
`% setenv MPINSIDE_LIB IMPI`	X86 Intel MPI
`% setenv MPINSIDE_LIB HPMPI`	X86 HP MPI
`% setenv MPINSIDE_LIB OPENMPI`	OpenMPI

4. Type the following command to run the program with MPInside:

```
% mpirun -np 4 MPInside ./example
```

5. Open the `mpinside_stats` report within a spreadsheet.

   For information about how to open an `mpinside_stats` report from within a spreadsheet, see the following:

"Opening the `mpinside_stats` Report Within a Spreadsheet" on page 13

6. (Optional) Scan the preamble.

   The preamble in the `mpinside_stats` report contains metadata about the programming run and explains how MPInside calculates some of its statistics.

   In `example`'s case, you specified the `MPINSIDE_ECHO_INPUT` environment variable, so the preamble lists the environment variables you used during this program run. Otherwise, however, it does not contain a lot of program-specific information. The preamble for `example` is as follows:

```
MPInside 3.6.3 standard(Sep  6 2013 14:06:40) Input variables:
MPINSIDE_PRINT_DIRTY                  : Set
MPINSIDE_ADD_COLUMN_MEANING           : Set
MPINSIDE_SIZE_DISTRI                  : T+12:0-3

 >>> column meanings <<<<
MPI_Init:  MPI_Init
Recv:      MPI_Recv
Send:      MPI_Send
Bcast:     MPI_Bcast:     Calls sending data+=comm_sz,Calls receiving data++;Root:Bytes sent++:Bytes received+=count
Allreduce: MPI_Allreduce: Calls sending data+=comm_sz;Bytes received+=count,Calls receiving data++

#calls:  Number of calls to the MPI function
#count:  Number of bytes transfered (P2P functions)
#comm_sz:        Sum of the communicator size for collective functions;Remote group size if intercomm
#recvcnt:        Bytes received parameter(collective functions, average for V functions)
#sendcnt:        Bytes_sent_parameter(collective functions,average for V functions)
#root:   Number of time rank was root of the collective function
```

   For information about how MPInside calculates the `Bcast` and `Allreduce` data, see Appendix A, "MPInside Calculations" on page 61.

7. Analyze the `Communication time totals` area of the report.

   The statistics in this part of the report summarize timings, in seconds, for the whole program. The following is an example of this area:

```
>>>> Communication time totals (s) 0 1<<<<
CPU   Compute   MPI_Init  Recv      Send      Bcast     Allreduce
0000  5.002315  0.000218  0.000000  0.000025  0.000033  3.013441
0001  3.004462  0.000201  3.004253  0.000000  2.000165  0.008992
```

```
0002  3.007882  0.000219  0.000000  0.000048  2.000170  3.009974
0003  3.004205  0.000195  3.007667  0.000000  2.000175  0.005770
```

Take a look at this area and ask yourself the following questions:

- *Do the statistics seem balanced?*

  The rows show the number of seconds that each rank spends in the following specific activities:

  - Total compute time
  - Processing `MPI_Init` functions
  - Processing `MPI_Recv` functions
  - Processing `MPI_Send` functions
  - Processing `MPI_Bcast` functions
  - Processing `MPI_Allreduce` functions

  In the case of program `example`, the `mpinside_stats` report shows that rank 0's computing time is much higher (at 5+ seconds) than the computing time for the other ranks (at 3+ seconds, each). In most cases, you want to design your program to balance computing time evenly across all ranks. In this case, the additional time is not expected, so this is something that you want to investigate.

- *What kind of communication traffic occurs between the ranks?*

  Generally, you want to balance the send times among the ranks. In program `example`, the send times are balanced between ranks 0 and 2, and the receive times are balanced between ranks 1 and 3. If, for example, ranks 0 and 2 sent the same amount of data but rank 0's send took longer to complete than rank 2's send, that might indicate network contention or load balancing problems.

In program `example`, note the `sleep()` routine that occurs before the `MPI_Send()` function in the `level_2()` routine for rank 0 and rank 2. The `sleep()` routine forces rank 1 and rank 3 to wait 3 seconds for the matching `MPI_Recv()`. A clearly asymmetric set of timings for the sends and receives suggests that the receivers might be spending a long time waiting for sends. In a larger program of, for example, 1000 ranks and more subtle imbalances, you could not conclude so quickly that a large `MPI_Recv()` time contributes so greatly to the load imbalance problem or to the performance of the pure transfers.

Additional programming runs that use more advanced MPInside features can help you to find the causes of performance problems more easily.

8. Analyze the `Bytes sent` area of the report, which is as follows in program `example`:

```
>>>> Bytes sent <<<<
CPU   Compute  MPI_Init  Recv      Send      Bcast   Allreduce
---   ------   --------  --------  #bytes    #root   --------
0000  ------   0         0         4096      2       0
0001  ------   0         0         0         0       0
0002  ------   0         0         4096      0       0
0003  ------   0         0         0         0       0
```

   The preceding output conveys the following information:

   - Ranks 0 and 2 sent 4096 bytes using a call to `MPI_Send`.
   - Rank 0 sent out 2 bytes during calls to `MPI_Allreduce`.
   - Ranks 1 and 3 did not send any data during any calls to `MPI_Send`.

   With regard to the data for your program, does this match your expectations?

9. Analyze the `Calls sending data` area of the report, which is as follows in program `example`:

```
>>>> Calls sending data <<<<
CPU   Compute  MPI_Init  Recv      Send    Bcast     Allreduce
---   ------   --------  --------  #calls  #comm_sz  #comm_sz
0000  ------   1         0         2         8       8
0001  ------   1         0         0         8       8
0002  ------   1         0         2         8       8
0003  ------   1         0         0         8       8
```

   The preceding output conveys the following information:

   - Each rank called `MPI_init` once.
   - Ranks 0 and 2 called `MPI_Send` two times. These are point-to-point calls. No other ranks called `MPI_Send`.

- From the listing, we know that program `example` performs collective operations with `MPI_COMM_WORLD` with four ranks. The 8s in the last two columns are the product of 2 X 4. In the `MPI_Bcast` column, this represents 2 ranks multiplied by 4 `MPI_Bcast` calls. In the `MPI_Allreduce` column, this represents 2 ranks multiplied by 4 `MPI_Allreduce` calls.

With regard to the data for your program, does this match your expectations?

10. Analyze the `Bytes received` area of the report, which is as follows in program `example`:

```
>>>> Bytes received <<<<
CPU   Compute  MPI_Init  Recv    Send      Bcast   Allreduce
---   ------   --------  #bytes  --------  #bytes  #bytes
0000  ------   0         0       0         2048    8192
0001  ------   0         4096    0         2048    8192
0002  ------   0         0       0         2048    8192
0003  ------   0         4096    0         2048    8192
```

The preceding output conveys the following information:

- Ranks 1 and 3 received 4096 bytes of data.
- Ranks 0–4 (all ranks) received 2048 bytes from `MPI_Bcast` calls, which is the result of 1024 bytes received from each of the two `MPI_Bcast` calls that Rank 0 initiated.
- The 8192 bytes that each rank received came from the two `MPI_Allreduce` calls. 4 ranks participated in each call, each contributing 1024 bytes of data.

11. Analyze the `Calls receiving data` area of the report, which is as follows in program `example`:

```
>>>> Calls receiving data <<<<
CPU   Compute  MPI_Init  Recv    Send      Bcast   Allreduce
---   ------   --------  #calls  --------  #calls  #calls
0000  ------   0         0       0         2       2
0001  ------   0         2       0         2       2
0002  ------   0         0       0         2       2
0003  ------   0         2       0         2       2
```

Ranks 1 and 3 each called `MPI_Recv` twice.

Each rank received data from two `MPI_Broadcast` calls and two `MPI_Allreduce` calls.

12. Examine the output in the `SIZE HISTOGRAMS` area of the report.

    The `MPINSIDE_SIZE_DISTRI` environment variable generates the `SIZE HISTOGRAMS` area of the report. All times are in seconds. All units for non-timing tables are expressed as the number of calls. The following is the MPInside output for program `example` for Rank 0 and Rank 1:

```
>>> Rank 0 Sizes distribution <<<
     Sizes          Send       Bcast     Allreduce
     65536          0          0         0
     32768          0          0         0
     16384          0          0         0
      8192          0          0         0
      4096          0          0         2
      2048          2          0         0
      1024          0          2         0
       512          0          0         0
       256          0          0         0
       128          0          0         0
        64          0          0         0
        32          0          0         0
         0          0          0         0
>>> Rank 0 Size distribution times (in s) <<<
     Sizes          Send       Bcast     Allreduce
     65536          0.000000   0.000000  0.000000
     32768          0.000000   0.000000  0.000000
     16384          0.000000   0.000000  0.000000
      8192          0.000000   0.000000  0.000000
      4096          0.000000   0.000000  3.013441
      2048          0.000025   0.000000  0.000000
      1024          0.000000   0.000033  0.000000
       512          0.000000   0.000000  0.000000
       256          0.000000   0.000000  0.000000
       128          0.000000   0.000000  0.000000
        64          0.000000   0.000000  0.000000
        32          0.000000   0.000000  0.000000
         0          0.000000   0.000000  0.000000
>>> Rank 1 Sizes distribution <<<
```

```
     Sizes          Recv       Bcast     Allreduce
     65536          0          0         0
     32768          0          0         0
     16384          0          0         0
      8192          0          0         0
      4096          0          0         2
      2048          2          0         0
      1024          0          2         0
       512          0          0         0
       256          0          0         0
       128          0          0         0
        64          0          0         0
        32          0          0         0
         0          0          0         0
>>> Rank 1 Size distribution times (in s) <<<
     Sizes          Recv       Bcast     Allreduce
     65536          0.000000   0.000000  0.000000
     32768          0.000000   0.000000  0.000000
     16384          0.000000   0.000000  0.000000
      8192          0.000000   0.000000  0.000000
      4096          0.000000   0.000000  0.008992
      2048          3.004253   0.000000  0.000000
      1024          0.000000   2.000165  0.000000
       512          0.000000   0.000000  0.000000
       256          0.000000   0.000000  0.000000
       128          0.000000   0.000000  0.000000
        64          0.000000   0.000000  0.000000
        32          0.000000   0.000000  0.000000
         0          0.000000   0.000000  0.000000
```

In the preceding output, look for numbers that are uneven. The block of information called `Rank 0 Size distribution times` shows an `MPI_Allreduce` of 4096 bytes that took 3+ seconds to complete, compared with `MPI_Send` and `MPI_Bcast` times of much less than a second. This indicates that you should look in your program for an `MPI_Allreduce` with a 4K data payload.

Run 2 — Simulating a Perfect Interconnect Environment

If the programming environment had a perfect network and perfect hardware, you might expect all message passing to occur perfectly, with no waiting. When you complete this run, you simulate a perfect environment. This run simulates the amount of waiting that occurs because of unbalanced loads and that is independent of the MPI engine.

The following procedure explains the environment variables to use when you run MPInside for the second time.

Procedure 5-2 To simulate a perfect interconnect environment

1. Type the following command to load the MPInside module:

   ```
   % module load MPInside
   ```

2. Type the following command to rename the MPInside report to `mpinside_perfect_stats`:

   ```
   % setenv MPINSIDE_OUTPUT_PREFIX mpinside_perfect
   ```

3. Type the following commands to specify that you want to run MPInside in a way that simulates a perfect computing environment:

   ```
   % setenv MPINSIDE_ECHO_INPUT
   % setenv MPINSIDE_PRINT_DIRTY
   % setenv MPINSIDE_ADD_COLUMN_MEANING
   % setenv MPINSIDE_MODEL PERFECT+1
   ```

4. (Conditional) Set the `MPINSIDE_LIB` environment variable to your MPI implementation.

   Perform this step if your MPI implementation is something other than SGI's MPT MPI. The default setting is `MPINSIDE_LIB MPT`, which assumes that SGI MPT is your MPI implementation.

   If you use an implementation that is not SGI's MPT MPI implementation, type the one command from the following list that pertains to your implementation:

Command	**MPI Implementation**
`% setenv MPINSIDE_LIB IMPI`	X86 Intel MPI
`% setenv MPINSIDE_LIB HPMPI`	X86 HP MPI
`% setenv MPINSIDE_LIB OPENMPI`	OpenMPI

5. Type the following command to run the program with MPInside:

   ```
   % mpirun -np 4 MPInside ./example
   ```

6. Compare the `mpinside_perfect_stats` report from this run to the `mpinside_stats` report from the previous program run.

   The preamble is nearly identical in both reports. The only difference is the echoed environment variables.

   The `Communication time totals` area shows the time spent by the MPI function if the MPI engine operated in a perfect environment with zero latency and infinite bandwidth. The timings in this perfect environment are very similar to the baseline run. This simulation running time is a little shorter because of the huge load imbalance that is introduced with the `sleep()` function. The true transfer for the run is almost nothing in regard to the load imbalance. The statistics are as follows:

```
>>>> Communication time totals (s) 0 1<<<<
CPU   Compute   MPI_Init  Recv      Send      Bcast     Allreduce  Stiffness
  0   5.0172    0.000131  0         0         0         3.020069           0
  1   3.011644  0.000141  3.008671  0         2.008501  0.008452           0
  2   3.012449  0.000144  0         0         2.008501  3.016319           0
  3   3.016353  0.000136  3.012414  0         2.008501  0                  0
```

   In the preceding output, notice the new column, `Stiffness`. For information about program stiffness, see the following:

   "Communication Stiffness" on page 39

   The `Bytes sent`, `Calls sending data`, `Bytes received`, and `Calls receiving data` areas are identical to the baseline statistics.

Run 3 — Evaluating Send Late Time

Send late time (SLT) refers to the amount of time a rank waits for data to be received from another rank. For more information about send late time, see "About MPInside Terminology" on page 6.

The following procedure shows a programming run that generates SLT statistics.

Procedure 5-3 To generate send time late statistics

1. Type the following command to load the MPInside module:

   ```
   % module load MPInside
   ```

2. Type the following command to rename the MPInside report to `mpinside_slt_stats`:

   ```
   % setenv MPINSIDE_OUTPUT_PREFIX mpinside_slt
   ```

3. Type the following commands to specify that you want to run MPInside in a way that generates SLT statistics:

   ```
   % setenv MPINSIDE_ECHO_INPUT
   % setenv MPINSIDE_PRINT_DIRTY
   % setenv MPINSIDE_ADD_COLUMN_MEANING
   % setenv MPINSIDE_EVAL_SLT
   % setenv MPINSIDE_EVAL_COLLECTIVE_WAIT
   ```

4. (Conditional) Set the `MPINSIDE_LIB` environment variable to your MPI implementation.

   Perform this step if your MPI implementation is something other than SGI's MPT MPI. The default setting is `MPINSIDE_LIB MPT`, which assumes that SGI MPT is your MPI implementation.

   If you use an implementation that is not SGI's MPT MPI implementation, type the one command from the following list that pertains to your implementation:

Command	**MPI Implementation**
% **setenv MPINSIDE_LIB IMPI**	X86 Intel MPI
% **setenv MPINSIDE_LIB HPMPI**	X86 HP MPI
% **setenv MPINSIDE_LIB OPENMPI**	OpenMPI

5. Type the following command to run the program with MPInside:

   ```
   % mpirun -np 4 MPInside ./example
   ```

6. Compare the `mpinside_slt_stats` report from this run to the `mpinside_stats` report from the previous program run.

   The preamble of the `mpinside_slt_stats` report contains information about the following additional statistics:

```
b_Bcast:          b_MPI_Bcast:       Barrier before MPI_bcast
b_Allreduce:      b_MPI_Allreduce:   Barrier before MPI_Allreduce
w_MPI_Recv:       w_MPI-Recv:        Send Late Time for MPI_Recv
unwind_overhead:  unwind_overhead:   Overhead Unwinding stack
```

```
mpinside_overhead: mpinside_overhead: Various MPInside overheads
Stiffness:         Stiffness:         Bytes sent=Stiffness=(Nb_Requests Att. Send)/(Nb_Request Att. Recv
```

Note: The following output example is very wide. The rightmost two columns are wrapped and shown below the main body of the output for inclusion in this documentation.

In this run, the `Communication time totals` area shows the new timings and is as follows:

```
>>>> Communication time totals (s) 0 1<<<<
CPU  Compute  MPI_Init w_MPI_Recv Recv     Send     b_Bcast  Bcast    b_Allreduce Allreduce unwind_overhead
0000 5.004536 0.000181 0.000000   0.000000 0.000072 0.000002 0.000022 3.024744    0.031092  0.000003
0001 3.000160 0.000181 3.006710   0.000040 0.000000 2.001051 0.000021 0.031649    0.024031  0.000004
0002 3.005915 0.000185 0.000000   0.000000 0.000102 2.018046 0.000018 3.023915    0.020045  0.000004
0003 3.000141 0.000183 2.997876   0.000058 0.000000 2.015068 0.005942 0.021838    0.021981  0.000003

                                             ## The last two columns of output are as follows:
                                                    mpinside_overhead Stiffness
                                                    0.000000          0.000000
                                                    0.000000          0.000000
                                                    0.000000          0.000000
                                                    0.000000          0.000000
```

In the preceding statistics, you can interpret all time in columns `w_MPI_Recv`, `b_Bcast` and `b_Allreduce` to be pure wait time. The pure, or real, physical transfer times are the timings in the `Recv`, `Bcast`, and `Allreduce` columns.

The real transfer times vary little with the example program. This is common in applications that show a load imbalance of this degree.

The output includes the `Stiffness` column. For information about program stiffness, see "Communication Stiffness" on page 39.

The `Bytes sent`, `Calls sending data`, `Bytes received`, and `Calls receiving data` areas are identical to the baseline statistics.

Run 4 — Examining the Call Stack Branches

Call stack branches, or *branches*, show the paths in the program that led to various types of communication. In its output, MPInside organizes the call stack branch information by rank. The MPInside report includes information about the activity at the other end of the commmunication (the call stack partner) and how much of the total run time was consumed by each of these communication-generating program paths.

The following procedure shows how to generate and analyze information about call stack branches in MPInside reports.

Procedure 5-4 To generate and analyze call stack branch reports

1. Type the following command to load the MPInside module:

   ```
   % module load MPInside
   ```

2. Type the following command to rename the MPInside report to `mpinside_branches_stats`:

   ```
   % setenv MPINSIDE_OUTPUT_PREFIX mpinside_branches
   ```

3. Type the following commands to specify that you want to run MPInside in a way that generates call stack branch information:

   ```
   % setenv MPINSIDE_ECHO_INPUT
   % setenv MPINSIDE_PRINT_DIRTY
   % setenv MPINSIDE_ADD_COLUMN_MEANING
   % setenv MPINSIDE_CALLSTACK_DEPTH 6
   % setenv MPINSIDE_CROSS_REFERENCE
   ```

4. (Conditional) Set the `MPINSIDE_LIB` environment variable to your MPI implementation.

   Perform this step if your MPI implementation is something other than SGI's MPT MPI. The default setting is `MPINSIDE_LIB MPT`, which assumes that SGI MPT is your MPI implementation.

   If you use an implementation that is not SGI's MPT MPI implementation, type the one command from the following list that pertains to your implementation:

Command	MPI Implementation
`% setenv MPINSIDE_LIB IMPI`	X86 Intel MPI
`% setenv MPINSIDE_LIB HPMPI`	X86 HP MPI

```
% setenv MPINSIDE_LIB OPENMPI
```

OpenMPI

5. Type the following command to run the program with MPInside:

   ```
   % mpirun -np 4 MPInside ./example
   ```

6. Type the following command to process the additional files that contain information about the MPInside call stack branches:

   ```
   % MPInside_post - s0 - e3 -l mpinside_clstk
   ```

   In addition to the expected MPInside report named `mpinside_branches_stats`, this MPInside run created the following call stack files:

   ```
   mpinside_clstk.0
   mpinside_clstk.1
   mpinside_clstk.2
   mpinside_clstk.3
   ```

   There are four call stack reports because there are four ranks in the example program.

   The `MPInside_post` command processes the call stack files and creates the following reports:

   ```
   mpinside_clstk_post.0
   mpinside_clstk_post.1
   mpinside_clstk_post.2
   mpinside_clstk_post.3
   ```

7. Analyze the call stack branch reports.

   The call stack branch reports show timing and partner information for each MPI function. For information about how to analyze these reports, see "Interpreting the Call Stack Branch Output" on page 33.

Note: The output in Example 1 and Example 2, which follows, is very wide. The rightmost column, or the rightmost two columns, are wrapped and shown below the main body of the output for inclusion in this documentation.

Example 1. The `mpinside_clstk_post.0` report for this run is as follows:

```
MPInside report rank 0
Send Branches Ids : 1 - 255
RECV Branches Ids : 257 - 511
WAIT Branches Ids : 513 - 771

MPI_FUNCTION   Branch ID  Receive Time(s)  Self%  Self totals  #Send reqs  #Recv reqs  Ave MBs sent
MPI_Allreduce  #771       2.011            66.10  66.1         0           1           0
Ancestors: level_2  /home/bryce6/dthomas/MPInside/TESTS/example.c:28
            level_1  /home/bryce6/dthomas/MPInside/TESTS/example.c:38
             main  /home/bryce6/dthomas/MPInside/TESTS/example.c:56
              __libc_start_main  ??:?

                                             ## The last column of output is as follows:
                                             Ave MBs received
                                             4096

MPI_FUNCTION   Branch ID  Receive Time(s)  Self%  Self totals  #Send reqs  #Recv reqs  Ave MBs sent
MPI_Allreduce  #769       1.008            33.14  99.2         0           1           0
Ancestors: level_2  /home/bryce6/dthomas/MPInside/TESTS/example.c:28
            level_1  /home/bryce6/dthomas/MPInside/TESTS/example.c:37
             main  /home/bryce6/dthomas/MPInside/TESTS/example.c:56
              __libc_start_main  ??:?

                                             ## The last column of output is as follows:
                                             Ave MBs received
                                             4096

MPI_FUNCTION   Branch ID  Receive Time(s)  Self%  Self totals  #Send reqs  #Recv reqs  Ave MBs sent
MPI_Allreduce  #1         0.023            0.75   100.0        1           0           2048
Ancestors: level_2  /home/bryce6/dthomas/MPInside/TESTS/example.c:16
            level_1  /home/bryce6/dthomas/MPInside/TESTS/example.c:37
             main  /home/bryce6/dthomas/MPInside/TESTS/example.c:56
              __libc_start_main  ??:?

                                             ## The last column of output is as follows:
                                             Ave MBs received
                                             0
```

```
MPI_FUNCTION   Branch ID  Receive Time(s)  Self%  Self totals  #Send reqs  #Recv reqs  Ave MBs sent
MPI_Allreduce  #2         0.000            0.00   100.0        1           0           2048
Ancestors: level_2  /home/bryce6/dthomas/MPInside/TESTS/example.c:16
            level_1  /home/bryce6/dthomas/MPInside/TESTS/example.c:38
             main  /home/bryce6/dthomas/MPInside/TESTS/example.c:56
              __libc_start_main  ??:?

                                             ## The last column of output is as follows:
                                             Ave MBs received
                                             0

MPI_FUNCTION   Branch ID  Receive Time(s)  Self%  Self totals  #Send reqs  #Recv reqs  Ave MBs sent
MPI_Allreduce  #770       0.000            0.00   100.0        0           1           0
Ancestors: level_2  /home/bryce6/dthomas/MPInside/TESTS/example.c:32
            level_1  /home/bryce6/dthomas/MPInside/TESTS/example.c:37
             main  /home/bryce6/dthomas/MPInside/TESTS/example.c:56
              __libc_start_main  ??:?

                                             ## The last column of output is as follows:
                                             Ave MBs received
                                             1024

MPI_FUNCTION   Branch ID  Receive Time(s)  Self%  Self totals  #Send reqs  #Recv reqs  Ave MBs sent
MPI_Allreduce  #772       0.000            0.00   100.0        0           1           0
Ancestors: level_2  /home/bryce6/dthomas/MPInside/TESTS/example.c:32
            level_1  /home/bryce6/dthomas/MPInside/TESTS/example.c:38
             main  /home/bryce6/dthomas/MPInside/TESTS/example.c:56
              __libc_start_main  ??:?

                                             ## The last column of output is as follows:
                                             Ave MBs received
                                             1024
```

Example 2. In the following output, notice that some branches are followed with information about the branch partners:

```
MPInside report rank 1
Send Branches Ids : 1 - 255
RECV Branches Ids : 257 - 511
WAIT Branches Ids : 513 - 771
```

```
 MPI_FUNCTION  Branch ID  Receive Time(s)  Self%  Self totals  #Send reqs  #Recv reqs  Ave MBs sent
     MPI_Recv  #258       2.003            39.72  39.7         0           1           0
Ancestors: level_2  /home/bryce6/dthomas/MPInside/TESTS/example.c:20
            level_1  /home/bryce6/dthomas/MPInside/TESTS/example.c:38
             main  /home/bryce6/dthomas/MPInside/TESTS/example.c:56
              __libc_start_main  ??:?
Partners_1_0:   0:#2:100.00:100.00

                                             ## The last two columns of output are as follows:
                                             Ave MBs received  Ave partner wait time(s)
                                             0                 2.003513

 MPI_FUNCTION  Branch ID  Receive Time(s)  Self%  Self totals  #Send reqs  #Recv reqs  Ave MBs sent
     MPI_Recv  #257       1.026            20.34  60.1         0           1           0
Ancestors: level_2  /home/bryce6/dthomas/MPInside/TESTS/example.c:20
            level_1  /home/bryce6/dthomas/MPInside/TESTS/example.c:37
             main  /home/bryce6/dthomas/MPInside/TESTS/example.c:56
              __libc_start_main  ??:?
Partners_1_0:   0:#1:100.00:99.88

                                             ## The last two columns of output are as follows:
                                             Ave MBs received  Ave partner wait time(s)
                                             0                 1.027300

 MPI_FUNCTION  Branch ID  Receive Time(s)  Self%  Self totals  #Send reqs  #Recv reqs  Ave MBs sent
    MPI_Bcast  #772       1.004            19.91  80.0         0           1           0
Ancestors: level_2  /home/bryce6/dthomas/MPInside/TESTS/example.c:32
            level_1  /home/bryce6/dthomas/MPInside/TESTS/example.c:38
             main  /home/bryce6/dthomas/MPInside/TESTS/example.c:56
              __libc_start_main  ??:?

                                             ## The last column of output is as follows:
                                             Ave MBs received
                                             1024

 MPI_FUNCTION  Branch ID  Receive Time(s)  Self%  Self totals  #Send reqs  #Recv reqs  Ave MBs sent
    MPI_Bcast  #770       1.004            19.91  99.9         0           1           0
Ancestors: level_2  /home/bryce6/dthomas/MPInside/TESTS/example.c:32
            level_1  /home/bryce6/dthomas/MPInside/TESTS/example.c:37
             main  /home/bryce6/dthomas/MPInside/TESTS/example.c:56
```

```
              __libc_start_main  ??:?

                                             ## The last column of output is as follows:
                                             Ave MBs received
                                             1024

 MPI_FUNCTION  Branch ID  Receive Time(s)  Self%  Self totals  #Send reqs  #Recv reqs  Ave MBs sent
MPI_Allreduce  #771       0.005            0.10   100.0        0           1           0
Ancestors: level_2  /home/bryce6/dthomas/MPInside/TESTS/example.c:28
            level_1  /home/bryce6/dthomas/MPInside/TESTS/example.c:38
             main  /home/bryce6/dthomas/MPInside/TESTS/example.c:56
              __libc_start_main  ??:?

                                             ## The last column of output is as follows:
                                             Ave MBs received
                                             4096

 MPI_FUNCTION  Branch ID  Receive Time(s)  Self%  Self totals  #Send reqs  #Recv reqs  Ave MBs sent
MPI_Allreduce  #769       0.001            0.02   100.0        0           1           0
Ancestors: level_2  /home/bryce6/dthomas/MPInside/TESTS/example.c:28
            level_1  /home/bryce6/dthomas/MPInside/TESTS/example.c:37
             main  /home/bryce6/dthomas/MPInside/TESTS/example.c:56
              __libc_start_main  ??:?

                                             ## The last column of output is as follows:
                                             Ave MBs received
                                             4096
```

For `Branch ID 258`, for example, notice the following:

- The `MPI_Recv` call (with the `level_2()` call on line `38`) was the partner (the matching send for this receive branch) with rank `0` branch ID `2`. Function `main` calls `level_1`, which calls `level_2`, which calls `MPI_Recv`. The exact source files and line numbers that led to the call to `MPI_Recv` are `.../TESTS/example.c.20` (where the `20` means line `20`) and `.../TESTS/example.c.38`.
- The line `Partners_1_0:  0:#2:100.00:100.00` shows that this branch spent 100% of its time partnering with rank `0`, branch `2`. The second `100.00` in this line reports that 100% of the time spent was waiting on a late send.

MPInside Calculations

This appendix section contains the following topics:

- "About MPInside and the Collective Functions" on page 61
- "Interpreting the Statistics for the `MPI_Bcast` Collective Function" on page 61

About MPInside and the Collective Functions

The collective functions perform across the network. In its output, MPInside considers the number of individual point-to-point operations that were needed for each collective function. When MPInside generates a count for these collective functions, the way the count is created depends on where the count appears in the output.

Interpreting the Statistics for the `MPI_Bcast` Collective Function

The MPInside output contains statistics for the `MPI_Bcast` functions used in the program. These statistics appear in the five tables of output that MPInside generates by default every time it runs. The following list includes each table title and explains how to interpret the statistic for the `MPI_Bcast` function in that table.

Table	**MPI_Bcast Statistic's Meaning**
`Bytes sent`	The number of times each rank acted as the root of an `MPI_Bcast` function.
`Calls sending data`	A count of the number of calls to the `MPI_Bcast` function multiplied by the number of ranks that participated in the function.
`Bytes received`	The number of bytes received by each rank as the result of an `MPI_Bcast` function.

`Calls receiving data`	A count of the number of ranks that received data as the result of an `MPI_Bcast` function.

Interpreting the Statistics for the `MPI_Allreduce` Collective Function

The MPInside output contains statistics for the `MPI_Allreduce` functions used in the program. These statistics appear in the five tables of output that MPInside generates by default every time it runs. The following list includes each table title and explains how to interpret the statistic for the `MPI_Allreduce` function in that table.

Table	**MPI_Allreduce Statistic's Meaning**
`Bytes sent`	This field contains `0`. For calls to the `MPI_Allreduce`, the `BYTES SENT` field is meaningless.
`Calls sending data`	The count of the number of calls to the `MPI_Allreduce` function multiplied by the number of ranks that participated in the function.
`Bytes received`	The number of bytes received by all ranks as the result of an `MPI_Allreduce` function.
`Calls receiving data`	The count of the number of times the rank called the `MPI_Allreduce` function.

Index