Unified Parallel C (UPC) User Guide

007–5604–005

New Features in This Manual

This update of the *Unified Parallel C (UPC) User Guide* supports the SGI Performance Suite 1.5 release. The major documentation change for this release is the support for the Many Integrated Core (MIC) technology from Intel Corporation.

Record of Revision

Version	Description
001	April 2010 Original Printing.
002	June 2010 Updated to support the SGI ProPack 7 Service Pack 1 release.
003	November 2011 Updated to support the SGI Performace Suite 1.3 release.
004	May 2012 Updated to support the SGI Performace Suite 1.4 release.
005	January 2013 Updated to support the SGI Performance Suite 1.5 release.

Contents

About This Guide

This guide describes the SGI® implementation of the Unified Parallel C (UPC) parallel extension to the C programming language standard.

Related Publications and Other Sources

Material about UPC is available from a variety of sources. Some of these, particularly webpages, include pointers to other resources. The following is a list of these sources:

- http://upc.gwu.edu/docs/upc_specs_1.2.pdf

  Hosts a PDF copy of *UPC Language Specifications V1.2, A publication of the UPC Consortium*. This document defines Unified Parallel C (UPC) as a parallel extension to the C programming language standard that follows the partitioned global address space programming model.

- *UPC: Distributed Shared Memory Programming*

  Authors: Tarek El-Ghazawi, William Carlson, Thomas Sterling, Katherine Yelick; ISBN-10: 0471220485 ; Published by Wiley – Interscience - May, 2005.

- http://upc.gwu.edu

  Hosts information that is relevant to UPC.

- http://upc.gwu.edu/downloads/Manual-1.2.pdf

  Hosts a PDF copy of the George Washington University *UPC Manual*, which explains UPC language features.

- `sgiupc`(1) man page

  The SGI Unified Parallel C (UPC) compiler man page, `sgiupc`(1), describes the `sgiupc`(1) command. The `sgiupc`(1) command is the front-end to the SGI UPC compiler suite. The command handles all stages of the UPC compilation process: UPC language preprocessing, UPC-to-C translation, back-end C compilation, and linking with UPC runtime libraries.

- *Message Passing Toolkit (MPT) User Guide*

Describes industry-standard message passing protocol optimized for SGI computers.

- *MPInside Reference Guide*

  Describes the SGI MPInside MPI profiling tool.

- *SGI UV GRU Development Kit Programmer Guide*

  Describes the SGI UV global reference unit (GRU) development kit. It describes the application program interface (API) that allows direct access to GRU functionality.

- SGI hardware reference guides

  The following SGI hardware reference guides provide architectural overviews:

  – *SGI UV 2000 System User Guide*

  – *SGI Altix UV 1000 System User's Guide*

  – *SGI Altix UV 100 System User's Guide*

  – *SGI Altix ICE 8200 Series System Hardware User's Guide*

  – *SGI Altix ICE 8400 Series System Hardware User's Guide*

  – *SGI ICE X System Hardware User Guide*

Obtaining Publications

You can obtain SGI documentation in the following ways:

- Accessing the SGI Technical Publications Library at the following URL:

  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.

- Retrieving man pages by typing `man` *title* on a command line.

- Retrieving SGI reference manuals that are provided in various formats in the SGI Performance Suite software package RPMs.

Helpful Online Resources

Supportfolio is the SGI support web site, including the SGI Knowledgebase, has links for software supports and updates at: https://support.sgi.com/login.

For a complete list of SGI online resources, see the *SGI Peformance Suite 1.x Start Here.*

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 any 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.

- Send mail to the following address:

  SGI
  Technical Publications
  46600 Landing Parkway
  Fremont, CA 94538

SGI values your comments and will respond to them promptly.

Introduction

Uniform Parallel C (UPC) is a Partitioned Global Address Space (PGAS) programming model. Shared variables and arrays can reside anywhere on the parallel computer system or be distributed across the nodes of the system. The UPC language allows direct references to the shared variables regardless of there locality. In addition, the UPC language defines synchronization and collective communication primitives. SGI Unified Parallel C (UPC) conforms to the UPC version 1.2 standard. Parallel I/O, which is not yet a part of the language, is not supported.

This manual describes the SGI implementation of the UPC standard. The `sgiupc`(1) man page includes additional SGI UPC information. This chapter covers the following topics:

Compiling and Running an SGI UPC Program

SGI supports SGI UPC on the following platforms:

- SGI ICE series systems
- SGI UV series systems
- SGI Rackable series systems

Each of the preceding SGI systems contains either an AMD processor or an Intel Xeon processor. These SGI systems can also include an InfiniBand network. If the system contains an Intel Xeon processor, it can also contain Intel Xeon Phi™ Many Integrated Core (MIC) devices. As this manual explains, the steps you use to compile and run a program, and the environment variables you use within your program, differ depending on the presence of InfiniBand or MIC technology. For example, if you

have code sections that you want to run on the MIC devices, you compile the entire program on the Xeon processor, and the system runs the appropriate code sections on the MIC devices.

To determine whether your system is equipped with MIC devices, type one of the following commands:

- On SGI UV systems: `topology -cops`
- On SGI ICE or SGI Rackable systems: `ls /sys/class/mic`

Use one of the following procedures to compile and run an SGI UPC program:

- "Compiling and Running an SGI UPC Program Exclusively on Xeon Processors or AMD Processors" on page 2
- "Compiling and Running an SGI UPC Program Natively on MIC Devices (Intel Xeon Processor Platforms Only)" on page 4
- "Compiling and Running a Heterogeneous SGI UPC Program on both Xeon Processors and on MIC Devices" on page 6

Compiling and Running an SGI UPC Program Exclusively on Xeon Processors or AMD Processors

The following procedure explains how to compile and run an SGI UPC program completely on Xeon processors or AMD processors. Do not use this procedure if your program includes sections that need to be run on MIC devices.

Procedure 1-1 To compile and run an SGI UPC program on Xeon processors or AMD processors

1. (Optional) Determine the UPC compiler versions that are available on your system.

   Perform this step if you need to choose a compiler version.

   Type the following command:

   ```
   % ls /opt/intel/*/bin/compilervars.sh
   ```

   Note the versions that this command returns. The next step in this procedure modifies your environment to use a specific version.

2. Type the following commands:

```
% module load mpt
% module load sgi-upc-devel
% source /opt/intel/version/bin/compilervars.sh intel64
```

   For *version*, specify the compiler version you want to use.

   The preceding commands modify your enviroment variables and enable you to run SGI UPC applications. You need to run these commands before you can retrieve the sgiupc(1) man page and before you can run SGI UPC applications.

   You need to load these programming modules into your environment only once.

3. Use a text editor to open a file, add UPC program statements, and save the file.

   For example, the following example UPC program resides in `hello.c`:

```
#include <upc.h>
#include <stdio.h>
int
main ()
{
  printf("Executing on thread %d of %d threads\n", MYTHREAD, THREADS);
}
```

4. Use the sgiupc(1) command to compile the program and generate an executable file.

   For example, the following command generates an executable file called `hello`:

```
# sgiupc hello.c -o hello
```

   The sgiupc(1) command is the front-end to the SGI UPC compiler suite. This command manages all stages of the UPC compilation process, which are UPC language preprocessing, UPC-to-C translation, back-end C compilation, and linking with UPC runtime libraries.

5. Use either the MPT mpirun(1) command or the mpiexec_mpt(1) command to run the executable program.

   For example, to direct the program to run on four threads, type the following command:

```
# mpirun -np 4 hello
```

You can expect output similar to the following:

```
Executing on thread 1 of 4 threads
Executing on thread 3 of 4 threads
Executing on thread 0 of 4 threads
Executing on thread 2 of 4 threads
```

The statements might not appear in the order listed in the preceding output example.

For more information about `sgiupc`(1) and `mpirun`(1), see the corresponding man pages.

Compiling and Running an SGI UPC Program Natively on MIC Devices (Intel Xeon Processor Platforms Only)

The following procedure explains how to compile and run an SGI UPC program only on the MIC devices included on an SGI system that is based on Intel Xeon processors. On systems with MIC devices, the SGI UPC runtime environment layers on the Intel MPI and on the Intel C compiler and runtime libraries. The `sgiupc`(1) command supports only the Intel C compiler on MIC devices.

To determine whether or not your system has MIC devices, type the following command:

```
/usr/sbin/hwinfo | grep 'N: mic'
```

The following procedure explains how to compile and run complete SGI UPC programs on systems with MIC devices. As an alternative, if you want to run only certain portions of a program on MIC devices, use the procedure in the following topic:

"Compiling and Running a Heterogeneous SGI UPC Program on both Xeon Processors and on MIC Devices" on page 6

Procedure 1-2 To compile and run an SGI UPC program on MIC devices

1. (Optional) Determine the UPC compiler versions that are available on your system.

   Perform this step if you need to choose a compiler version.

Type the following command to retrieve the Intel compiler versions for the Xeon processors:

```
% ls /opt/intel/*/bin/compilervars.sh
```

Type the following command to retrieve the available Intel MPI compiler versions:

```
% ls /opt/intel/impi/*/intel64/bin/mpivars.sh
```

Note the versions that these commands return. The next step in this procedure modifies your environment to use specific versions.

2. Type the following commands to create the SGI UPC programming environment:

```
% module load sgi-upc-devel
% source /opt/intel/compiler_version/bin/iccvars.sh intel64
% source /opt/intel/impi/mpi_version/intel64/bin/mpivars.sh
```

The preceding commands modify your enviroment variables and enable you to run SGI UPC applications. You need to run these commands before you can retrieve the `sgiupc`(1) man page and before you can run SGI UPC applications.

You need to run these commands only once.

3. Use a text editor to open a file, add UPC program statements, and save the file.

For example, the following example UPC program resides in `hello.c`:

```
#include <upc.h>
#include <stdio.h>
int
main ()
{
  printf("Executing on thread %d of %d threads\n", MYTHREAD, THREADS);
}
```

4. Use the `sgiupc`(1) command to compile the program and generate an executable file.

For example, the following command generates an executable file called `hello`:

```
# sgiupc -mmic hello.c -o hello
```

The `sgiupc`(1) command is the front-end to the SGI UPC compiler suite. This command manages all stages of the UPC compilation process: UPC language

preprocessing, UPC-to-C translation, back-end C compilation, and linking with UPC runtime libraries.

The `-mmic` parameter bilds the application to run natively on MIC devices.

5. Use the Intel MPI `mpiexec.hydra`(1) command to run the executable program.

   For example, to direct the program to run on four threads on device `mic0`, specify the `-np 4` parameter, and type the following command:

   ```
   # mpiexec.hydra -np 4 -host mic0 ./hello
   ```

   You can expect output similar to the following:

   ```
   Executing on thread 1 of 4 threads
   Executing on thread 3 of 4 threads
   Executing on thread 0 of 4 threads
   Executing on thread 2 of 4 threads
   ```

   The statements might not appear in the order listed in the preceding output example.

   For more information about `sgiupc`(1) and `mpiexec.hydra`(1), see the corresponding man pages.

Compiling and Running a Heterogeneous SGI UPC Program on both Xeon Processors and on MIC Devices

SGI UPC enables you to run only specific, predefined sections of code on a coprocessor based on the Intel MIC architecture. This topic explains how to compile and run an SGI UPC program that contains both code sections to be run on the Xeon processors and code sections to to be run on MIC devices. The beginning of this procedure is identical to the procedure that explains how to compile and run a program on only Xeon processors. This procedure repeats these initial steps for your convenience.

You can use the `#pragma offload` directive to mark code segments that you want to run on MIC devices. When the compiler encounters a `#pragma offload` directive statement, it runs the code on the MIC coprocessor if it is available. Otherwise, the code runs on the central processing unit.

SGI UPC supports shared variable specifications inside a `#pragma offload` directive. However, SGI UPC does not support the `__declspec((target(mic))` or the `__attribute__(target(mic))` specifications on UPC shared variables. For

more information about the #pragma offload statement, see your Intel C compiler documentation.

The following procedure explains how to run SGI UPC programs with some code sections marked for MIC devices.

Procedure 1-3 To compile and run an SGI UPC program with some code sections marked to run on MIC devices

1. (Optional) Determine the UPC compiler versions that are available on your system.

   Perform this step if you need to choose a compiler version.

   Type the following command:

   ```
   % ls /opt/intel/*/bin/compilervars.sh
   ```

   Note the versions that this command returns. The next step in this procedure modifies your environment to use a specific version.

2. Type the following commands:

   ```
   % module load mpt
   % module load sgi-upc-devel
   % source /opt/intel/version/bin/compilervars.sh intel64
   ```

   For *version*, specify the compiler version you want to use.

   The preceding commands modify your enviroment variables and enable you to run SGI UPC applications. You need to run these commands before you can retrieve the sgiupc(1) man page and before you can run SGI UPC applications.

   You need to load these programming modules into your environment only once.

3. Use a text editor to open a file, add UPC program statements, add the #pragma offload directive where needed, and save the file.

For example, the following example UPC program resides in `hello.c`:

```
__declspec(target(mic))
int global = 55;
__declspec(target(mic))
int foo()
{
    return ++global;
}
main()
{
    int i;
    int j;
    #pragma offload target(mic) in(global) out(i, global) nocopy(j)
    {
        i = foo();
        j = i;
    }
    printf("global = %d, i = %d (should be the same)\n", global, i);
}
```

4. Use the `sgiupc`(1) command and the `mpirun`(1) command to compile the program and run the executable file.

   For example, the following command generates an executable file called `hello`:

   ```
   # sgiupc -offload-build hello.c -o hello
   # mpirun -np 4 ./hello
   ```

   In this example, the `-np 4` parameter runs the program on four threads.

Mixing UPC Programs with Programs Written In Other Languages

You can mix UPC programs with programs written in other languages. The rules are similar to those for mixing C programs with those written in other languages. In the main program, make sure to invoke the UPC library function `__upc_init()` at the start of the main function. This library function is required, and its role is to initialize `libupc`.

The SGI UPC compiler uses a `struct` type to represent a shared pointer. This practice accommodates large thread counts and large blocking sizes. This

representation is subject to change in later releases, so to pass a shared pointer to a non-UPC function, SGI recommends that you use functions supported within UPC to access the individual components.

When you use the sgiupc(1) command to compile the main program, the compiler links in the necessary libraries. If the main program is not a UPC program, or if you did not compile the main program with the sgiupc(1) command, you need to link in the appropriate libraries explicitly.

Using the sgi_upc Directive

You can use the sgi_upc directive to improve program performance.

Consider the following loop:

```
upc_forall (i = 0; i < N; i++; i)
  a[i] = b[i] + c[i];
```

If the array references are all remote, there are 2 X N remote loads and N remote stores performed in this loop.

If you know that the loop can be safely vectorized, insert a #pragma sgi_upc vector=on directive, as follows, to direct the compiler to vectorize the loop:

```
#pragma sgi_upc vector=on
upc_forall (i = 0; i < N; i++; i)
 a[i] = b[i] + c[i];
```

Conversely, if you know that there could be aliasing problems for the shared variables within the preceding loop, you do not want to insert a #pragma sgi_upc vector=on directive.

When you use the sgi_upc directive, the number of remote loads is reduced to 2, and the number of stores is reduced to 1. Each of these actions accesses N elements at a time. The directive reduces the communications overhead that you incur when a program accesses remote data. If a, b, and c are shared restricted pointers, then the compiler can perform this optimization without the user having to specify the sgi_upc directive.

Debugging SGI UPC Programs

The Allinea Distributed Debugging Tool (DDT) is an advanced debugging tool for scalar, multithreaded, large-scale parallel applications. DDT 3.1 and later supports `sgiupc` 1.05 and later.

For more information about DDT, type `ddt -h` or browse to the information on following website:

http://www.allinea.com.

Analyzing Application Performance

The Parallel Performance Wizard (PPW) is a performance analysis tool for partitioned global address space (PGAS) programs. PPW 2.8 and later supports `sgiupc` 1.05 and later.

For more information on PPW, see the `ppw` man page, the `ppwhelp` command, or the following site: http://ppw.hcs.ufl.edu/.

OFED Configuration for UPC

You can specify the maximum number of queue pairs (QPs) for SHMEM and UPC applications when run on large clusters over OFED fabric. If the `log_num_qp` parameter is set to a number that is too low, the system generates the following message:

```
MPT Warning: IB failed to create a QP
```

SHMEM and UPC codes use the InfiniBand RC protocol for communication between all pairs of processes in the parallel job, which requires a large number of QPs. The `log_num_qp` parameter defines the $\log_2$ of the number of QPs. The following procedure explains how to specify the `log_num_qp` parameter.

Procedure 1-4 To specify the `log_num_qp` parameter

1. Log into one of the hosts upon which you installed the MPT software as the root user.
2. Use a text editor to open file `/etc/modprobe.d/libmlx4.conf`.

3. Add a line similar to the following to file `/etc/modprobe.d/libmlx4.conf`:

   ```
   options mlx4_core log_num_qp=21
   ```

   By default, the maximum number of queue pairs is 2^{17} (131072).

4. Save and close the file.
5. Repeat the preceding steps on other hosts.

UPC Job Environment

This chapter describes the following topics:

About the UPC Job Environment

The SGI® Unified Parallel C (UPC) run-time environment includes the the following libraries:

- Message Passing Toolkit (MPT) MPI libraries.
- SHMEM libraries.
- SGI UV Global Reference Unit (GRU) libraries. The GRU libraries are linked automatically on SGI UV 2000, SGI UV 1000, SGI UV 200, and SGI UV 100 systems, all of which contain a GRU coprocessor.

The libraries provide job launch, parallel job control, memory mapping, and synchronization functionality. You can use the `mpirun`(1) or `mpiexec_mpt`(1) commands to launch SGI UPC jobs just as you can for MPT MPI or SHMEM jobs. UPC thread numbers correspond to `SHMEM PE` numbers and to MPI rank numbers for `MPI_COMM_WORLD`.

By default, UPC (MPI) jobs have UPC threads (MPI processes) pinned to successive logical CPUs within the system or cpuset in which the program runs. This is often optimal, but if you need to map UPC threads to logical CPUs in a different way, see the following:

- On the `mpi`(1) man page, see the information under the heading `Using a CPU list` and see the information about the `MPI_DSM_CPULIST` envoironment variable.
- On the `omplace`(1) man page, see the information about placement of parallel MPI and UPC jobs.

Referencing Nonlocal Portions of Shared Arrays (SGI UV™ Systems)

On the SGI UV series systems, SGI UPC offers the following options for performing references to non-local portions of shared arrays:

- Processor-driven shared memory

  By default, UPC uses processor-driven references for nearby sockets.

- Global reference unit (GRU)-driven shared memory

  The GRU is a remote direct memory access (RDMA) facility provided by the UV hub application-specific integrated circuit (ASIC).

- InfiniBand fabric-driven shared memory access

The following environment variables are among the most commonly used to control references to non-local portions of shared arrays:

Environment Variable	Purpose
`GRU_RESOURCE_FACTOR=2`	Some SGI UV systems have Intel processors with two hyper-threads per core, while others have a single hyper-thread per core. When dual hyper-threads per core are available, most HPC codes benefit by leaving one hyper-thread per core idle, thereby giving more cache and functional unit resources to the active hyper-thread to be assigned to one of the UPC threads. This is easy to do because the upper half of the logical CPUs (by number) are hyper-threads that are paired with the lower half of the logical CPUs. Set `GRU_RESOURCE_FACTOR=2` when leaving half of the hyper-threads idle. For more information about the `GRU_RESOURCE_FACTOR` environment variable, see the `gru_resource`(3) man page.
`GRU_TLB_PRELOAD=100`	Set `GRU_TLB_PRELOAD=100` to get the best GRU-based bandwidth for large block copies. For more information about the `GRU_TLB_PRELOAD` environment variable, see the `gru_environment`(7) man page.

`MPI_GRU_CBS=0`

This environment variable makes all GRU resources available to UPC.

For more information about the `MPI_GRU_CBS` environment variable, see the `MPI`(1) man page.

`MPI_SHARED_NEIGHBORHOOD`

By default, UPC uses GRU-driven references for distant references. You can use the the `MPI_SHARED_NEIGHBORHOOD` environment variable to tune the threshold between "nearby" and "distant". For more information about the `MPI_SHARED_NEIGHBORHOOD` environment variable, see "Tuning Runtime Behavior" on page 15.

You can experiment with the `MPI_SHARED_NEIGHBORHOOD=HOST` setting. Some shared array access patterns are faster when you use processor-driven references.

For more information about the `MPI_GRU_CBS` environment variable, see the `MPI`(1) man page.

Tuning Runtime Behavior

The UPC runtime library includes many environment variables that can affect or tune run-time behavior. The following list describes these variables.

Environment Variable **Purpose**

`UPC_ALLOC_MAX`

Sets the per-thread maximum amount of memory, in bytes, that can be allocated dynamically by `upc_alloc()` and the other shared array allocation functions.

If you set `UPC_ALLOC_MAX`, make sure to verify that the `SMA_SYMMETRIC_SIZE` variable is set correctly. The `SMA_SYMMETRIC_SIZE` environment variable must be set to the sum of the value of `UPC_ALLOC_MAX` plus the amount of space consumed by statically allocated arrays in the UPC program.

When you run UPC programs on InfiniBand clusters and you set the `UPC_ALLOC_MAX` environment variable to the right size, the system

preallocates physical memory in the shared array heap. If the actual memory space used by dynamically allocated arrays is less than the preallocated amount, excessive physical memory is consumed.

The default is the amount of physical memory per logical CPU on the system.

For more information about the UPC_ALLOC_MAX environment variable, see the sgiupc(1) man page.

For more information about the SMA_SYMMETRIC_SIZE environment variable, see intro_shmem(3) man page.

Availability:

- SGI ICE systems
- SGI UV systems
- SGI Rackable systems with InfiniBand

UPC_CAUTIOUS_STRICT

When the UPC_CAUTIOUS_STRICT environment variable is set to a nonzero value, it is enabled. When enabled, libupc performs a upc_fence call before all strict accesses, regardless of whether the previous access was strict or relaxed.

When the UPC_CAUTIOUS_STRICT=0, it is disabled. When disabled, libupc performs a upc_fence call only if there were one or more relaxed writes since the previous upc_fence access.

The default is disabled.

For more information about the UPC_CAUTIOUS_STRICT environment variable, see the sgiupc(1) man page.

Availability:

- SGI ICE systems
- SGI UV systems with InfiniBand
- SGI Rackable systems with InfiniBand

UPC_HEAP_CHECK

When UPC_HEAP_CHECK=1, libupc checks the integrity of the shared memory heap from which shared arrays are allocated.

The default value is 0.

For more information about the UPC_HEAP_CHECK environment variable, see the sgiupc(1) man page.

Availability:

- SGI ICE systems
- SGI UV systems
- SGI Rackable systems with InfiniBand

UPC_IB_BUFFER_SIZE

The UPC_IB_BUFFER_SIZE environment variable sets the size of the buffer used for InfiniBand fabric copy operations. This per-thread buffer is allocated and used for remote-to-remote copies over the InfiniBand fabric and for transfers of data to and from the InfiniBand network in which the data cannot be transferred directly.

The default size is 16 kilobytes. The minimum size is 1 kilobytes.

For more information about the UPC_IB_BUFFER_SIZE environment variable, see the sgiupc(1) man page.

Availability:

- SGI ICE systems with InfiniBand
- SGI UV systems with InfiniBand
- SGI Rackable systems with InfiniBand

Tuning Execution Performance (SGI UV Series Systems)

SGI supports a number of MPI and SHMEM environment variables that you can use to tune execution performance on SGI UV systems. Some of the most useful variables are as follows:

Environment Variable	Purpose
MPI_SHARED_NEIGHBORHOOD	When you set MPI_SHARED_NEIGHBORHOOD=HOST, UPC shared arrays use processor-driven shared memory transfers instead of GRU transfers. The size of the memory blocks being accessed in a remote part of a shared array, and other factors, determine whether processor-driven or GRU-driven transfers perform better. The default is MPI_SHARED_NEIGHBORHOOD=BLADE, which directs UPC threads to use processor-driven shared memory for references to shared array blocks that have affinity for the threads associated with sockets on the same UV hub. For more information about the MPI_SHARED_NEIGHBORHOOD environment variable, see the MPI(1) man page.
MPI_GRU_CBS	These environment variables reserve SGI UV GRU resources for MPI, which makes them unavailable to UPC. If you set MPI_GRU_CBS=0, MPI does not reserve permanent GRU resources, which leaves all GRU resources available to UPC. For more information about the MPI_GRU_CBS environment variable, see the MPI(1) man page.
GRU_RESOURCE_FACTOR	This environment variable specifies an integer multiplier that increases the amount of per-thread GRU resources that can be used by a UPC program. If UPC programs are placed such that some portion of the logical CPUs (hyper-threads) on each UV hub are left idle, you can specify a corresponding integer multiplier. For example, if half of the logical CPUs are idle, SGI recommends that you set GRU_RESOURCE_FACTOR=2. For more information about the GRU_RESOURCE_FACTOR environment variable, see the gru_resource(3) man page.

Index