QCDSP Communications Specification & the MPI implementation.

A.N. Jackson & S. Booth

Revision: 1.1 Date: 2001/09/11 13:03:35

Contents

1 QCDSP OS Communications

On QCDSP, the parallel communication subroutines were defined within the OS, and this will also be the case for QCDOC. However, for development purposes, we need a parallel implementation of the code that can be run here at EPCC while the QCDOC hardware is being developed. To do this, we intend to implement the communications using MPI (v.1), via the interface defined in `sysfunc.h`. This interface appears to offer a greater level of functionality than is currently required, as only a subset of the available subroutines are used by the C++ QCDSP code. For now, only the essential elements of the interface will be included here, while keeping in mind the possibility of extending the interface as the need arises.

1.1 Communications Flags

There are a number of constants defined in `scu_enum.h` which associate unique numbers with various communications parameters. The first set of definitions identifies the physical directions in which communications are to be carried out.

enum SCUDir	**Name**	**Value**	**Direction**
	SCU_TM	0	$-t$
	SCU_TP	1	$+t$
	SCU_XM	2	$-x$
	SCU_XP	3	$+x$
	SCU_YM	4	$-y$
	SCU_YP	5	$+y$
	SCU_ZM	6	$-z$
	SCU_ZP	7	$+z$

In the case of QCDSP, this actually mapped the physics directions onto physical wires. For QCDOC, this will also map onto a set of wires using the same numbers (0-7), but the actual physical wire being used will be decided by the OS, for the purposes of machine partitioning. The four different axes of the system are labelled in a similar fashion:

enum SCUAxis	**Name**	**Value**	**Direction**
	SCU_T	0	t-axis
	SCU_X	1	x-axis
	SCU_Y	2	y-axis
	SCU_Z	3	z-axis

Finally, the communications are labelled as being either sends or receives using:

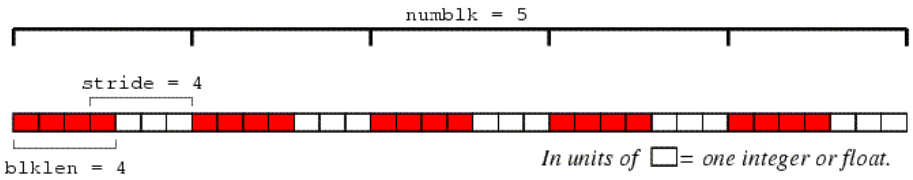
enum SCUXR	**Name**	**Value**	**Meaning**
	SCU_REC	0	Receive from a given direction.
	SCU_SEND	8	Send in a given direction.

1.2 Data structures: The SCUDirArg class

Each instance of the `SCUDirArg` class (defined in `scu_dir_arg.h`) describes the type of data-structure to be transmitted, whether this data should be sent or received, and also the direction in which the data is to communicated. The data types are either contiguous or block-strided, and are defined using the following parameters:

Type	Name	Description
`void*`	`addr`	Base address of the data-block.
`SCUDir`	`dir`	Direction of this communication.
`SCUXR`	`xr`	The send or receive flag.
`int`	`blklen`	Number of elements in each block of data.
`int`	`numblk`	Number of blocks.
`int`	`stride`	Number of elements between the start of the last element of one block and the first element of the successive block.

If this is unclear, consider this simple example of a strided data set:



The most important issue here, from our perspective, concerns the units in which both the block-length and the stride are specified. The unit appears to be the number of `float` or `int` elements, as opposed to the number of bytes. This has serious implications for our aim of allowing both `float` and `double` precision calculations. However, if no `int` data is being transferred between processors, then we can safely assume that the number of bytes associated with each element is that associated with the chosen level of precision, and the problem disappears.

A number of methods are also defined by the `SCUDirArg` class to allow the data-structures to be defined and manipulated.

`SCUDirArg::SCUDirArg (void* addr, SCUDir dir, SCUXR xr, int blklen, int numblk = 1, int stride = 1)`

This constructor allows the parameters of the communication to be set when an instance of the `SCUDirArg` class is created. Note that `numblk` and `stride` will both default to 1 if other values are not supplied.

`void SCUDirArg::Init (void* addr, SCUDir dir, SCUXR xr, int blklen, int numblk = 1, int stride = 1)`

Used to initialise an 'empty' instance of the SCUDirArg class. Like the previous constructor subroutine, `numblk` and `stride` will both default to 1.

`void* SCUDirArg::Addr ()`

Returns the base address associated with this `SCUDirArg`.

`void* SCUDirArg::Addr (void*addr)`

`void SCUDirArg::Reload (void* a, int blklen, int numblk = 1, int stride = 1)`

Used to change the block-length, the number of blocks and value of the stride. On QCDSP, this also reloaded the communication pattern into the DMA.

1.3 The OS Interface

On both QCDSP and QCDOC, the operating system interface is defined in the header file `sysfunc.h`, apart from standard subroutines such as `printf` which are defined elsewhere. In the original implementation, every subroutine in the communications layer is given "friend function" access to the `SCUDirArg` class, so they can look up the values of the instance variables directly.

`int UniqueID ()`

Returns a number which is unique for each node. These numbers have no particular significance, but one part of the library does attempt to used them to seed an RNG, and therefore zerois not advisable.

```
int CoorT ()
int CoorX ()
int CoorY ()
int CoorZ ()
```

Functions to return four-dimensional coordinates of the node.

```
int SizeT ()
int SizeX ()
int SizeY ()
int SizeZ ()
```

Functions to return the size of the 4-D processor grid in each direction.

`int NumNodes ()`

Returns the total number of nodes.

`unsigned int Seed ()`

`unsigned int SeedS ()`
`unsigned int SeedT ()`
`unsigned int SeedST ()`

For initialising the random number seeds, using numbers which on QCDSP were loaded at boot time. For `Seed()`, the seed is different for each node and is changed every time the machine is reset. `SeedS` is the same for each node (spatially fixed, hence the S), but changes in time. `SeedT` is different for each node, but is fixed in time (the T), so it is unchanged by a reset. `SeedST` is the same for each node (spatially fixed, hence the S), and the same after every reset (fixed time, hence T).

`unsigned int sync ()`

Synchronises all the processors.

`int SCURemap (`**SCUDir** *dir*`)`

This subroutine looks up the 'wire number' associated with a given direction. This can be used to create a mapping table, which maps directions #0-7 onto wires #0-7 in whichever order is most appropriate.

`void SCUTrans (`**SCUDirArg**$*$ *arg*`)`

Performs a single data-transfer, as specified by the `SCUDirArg` object pointed to by `arg`.

`void SCUTrans (`**SCUDirArg**$**$ *arg*`, int` *n*`)`

Used to perform `n` simultaneous data-transfers, as specified by the array of `SCUDirArg` objects passed through `arg`.

`void SCUTrans (`**SCUDirArg**$*$ *arg*`, unsigned int`$*$ *offset*`, int` *n*`)`

Perform `n` transfers, all of which have the same block, stride and number of blocks, but have different addresses. Each transfer is started at a specified offset (`offset[i]`) relative to the base-address held in the `SCUDirArg`.

`void SCUSetDMA (`**SCUDirArg**$*$ *arg*`)`
`void SCUSetDMA (`**SCUDirArg**$**$ *arg*`, int` *n*`)`

On QCDSP, this is used to set up the DMA, using the array of comms definitions held in `arg` (no transfers are done).

`void SCUTransAddr (`**SCUDirArg**$*$ *arg*`)`
`void SCUTransAddr (`**SCUDirArg**$**$ *arg*`, int` *n*`)`

This function performs SCU transfers, but does not alter the existing block, stride and number of blocks held in the registers of the SCU. This call must be preceded by a `SCUSetDMA` call. The

base-addresses of the data to be communicated are taken from the array `SCUDirArg` objects pointed to by `arg`.

`void SCUTransComplete ()`

The function does not return until all of the transfers on this PE have been completed.

1.4 Questions concerning the interface.

These last few points can be resolved when we have access to the original `scu_dir_arg.C` from the QCDSP code.

- `Addr()` is assumed to set the base-address of this instance of `SCUDirArg` to `addr`, and then returns the *previous* base-address. Is this the case.
- Has the `Reload()` command been interpreted correctly?

2 MPI Implementation: MPI-SCU

2.1 Introduction

The MPI implementation of the QCDSP SCU layer is a stand-alone system, which may be compiled with or without the QCDSP code. Only MPI version 1 directives are used.

2.2 Contents Of This Distribution

The MPI-SCU distribution contains the following files:

File	Description
`scu_enum.h`	Slightly modified re-implementation of the communications enums.
`scu_dir_arg.h`	Headers for the data-structure definition routines.
`scu_dir_arg.C`	Implementation of the above.
`sysfunc.h`	Headers for the MPI-emulated QOS calls.
`sysfunc.C`	Implementation of the above.
`mpi_requests.h`	Source for the MPI-implementation-specific request handler.
`Makefile`	Makefile for the MPI-SCU library. Does not make the test-programs.
`commsMPI.def`	Example comms-definition file.
`doc/`	MPI-SCU documentation, including this file.
`lib/`	Directory where the MPI-SCU library is placed.
`test/`	Directory for MPI-SCU test programs.

Note that the MPI-SCU Makefile will have to altered in order to work at an institution other that EPCC. The parts that will need to be altered are specified at the top of the Makefile, and concern the MPI compiler name and include/link paths.

2.3 Integrating MPI-SCU Into The QCDSP Code

It should be fairly straightforward to integrate the MPI-SCU library into the QCDSP code.

- Compile the MPI-SCU library on its own (see the above point concerning the Makefile's portability).
- Add the `.../MPI-SCU/` directory to the include path.
- Add the `.../MPI-SCU/lib/` directory to the link path.
- Add the `-lcommsMPI` flag to the compile/link command.

The only problem could be that any given version of the QCDSP code may contain its own versions of some of the files in this distribution (in particular, `scu_enum.h`). For now, we suggest simply moving the non-MPI versions of the conflicting files out of the way. In the future, this should be remedied by adding a suitable MPI flag to the QCDSP code's Makefile.

2.4 Compiler Options

These compiler switches specify some of the basic parameters of the MPI-SCU layer. All of these can be overridden in the Makefile, using the `-D` flag in the compiler flags.

Name	Default	Description
VERBOSE	UNDEFINED	Outputs detailed information concerning the comms calls to a set of log-files (one for each processor).
NDIM	4	Dimensionality of the system. Currently, only NDIM = 4 is meaningful because the interface does not allow access to anything other than 4 dimensions.
COMMS_ENVVAR	"COMMS_DEF"	Name of the environment variable used to specify the run-time parameters of the MPI-SCU layer. If no such environment variable exists, then the COMMS_DEFFILE is used instead.
COMMS_DEFFILE	"commsMPI.def"	Default filename (relative to the location of the executable) of a file that contains the run-time parameters of the MPI-SCU layer.
COMMS_DATASIZE	4	Default size (in bytes) of the fundamental data type to be transferred between processors (i.e. the size of integer or floating-point numbers). This can be overridden at the software (SCUDirArg) level.

2.5 Run-time Control

The parallel environment of the code can be specified at run-time, via the environment variable specified by COMMS_ENVVAR, or the file specified by COMMS_DEFFILE. The environment variable can either directly specify the MPI-SCU parameters, or point to a file containing those definitions. The definitions themselves consist of a series of tag & value pairs, in the format {NAME=value}, specifying the following aspects of the parallel environment:

Item	Level	Description	Default
{GRID=t,x,y,z}	REQUIRED	Defines the number of processors in each direction.	n/a
{LOGFILE=⟨filename⟩}	Optional	This should specify the absolute or relative path of a file to which VERBOSE textual output should be placed. If its value is `stderr` or `stdout` that stream is used instead.	`comlog`
{SEED=⟨integer⟩}	Optional	Specify the RNG seed that SeedS or SeedST will return	`1`
{SEEDFILE=⟨filename⟩}	Optional	Specifies a file containing a number of RNG seeds. This is used by Seed and SeedT to define a different seed for every processor.	`rng.dat`

The environment variable is determined as either pointing to a file or containing the tags based on the whether or not it contains a "{" or not (ignoring whitespace). The contents of the string or the file is turned into a stream of tokens, using the characters [{} ,=\n\t] as delimiting whitespace. The order of the definition items is not important, but the case is.

For example, consider setting the environment variable to a filename (in bash)

```
export COMMS_DEF="commsMPI.def"
```

where the file `commsMPI.def` contains the following information:

```
{ GRID = 64,32,32,32 }
{ LOGFILE = comms.log }
{ SEED = 128366328 }{ SEEDFILE = rngseeds.dat }
```

This will use a $(t, x, y, z) = 64 \times 32 \times 32 \times 32$ processor grid, and all VERBOSE output will be sent to log-files called `comms.log.X` where `X` is the processor number. `SeedS()` & `SeedST()` will return `128366328`, and `Seed()` & `SeedT()` will return a RNG seed taken from the file `rndseeds.dat`.

This could also be achieved using the environment variable alone. For example, in `bash` one can use:

```
export COMMS_DEF="{LOGFILE=comms.log}{SEED=128366328}
{GRID=64,32,32,32}{SEEDFILE=rngseeds.dat}"
```

This will produce exactly the same behaviour as the previous file-based approach.

2.6 The Communications Flags

The flags specified by `scu_enum.h` have only been changed very slightly: An extra flag has been added to the end of each enum to clearly distinguish between undefined and defined values.

Enumeration `SCUDir` now also defines `SCU_NoDir = -1`.
Enumeration `SCUAxis` now also defines `SCU_NoAxis = -1`.
Enumeration `SCUXR` now also defines `SCU_NoXR = -1`.

2.7 The SCUDirArg Class

Each instance of the `SCUDirArg` class defines and commits a MPI_Datatype appropriate for the specified number of blocks, block-length and stride. The basic data type is always defined as being floating-point, but the number of bytes for each float (which defaults to `COMMS_DATASIZE`) an be specified using the following method:

```
SCUDirArg::SetDataSize( int mpi_datasize);
```

where the integer argument specifies the number of bytes required for every element of data. For example, if we wish to use an instance of `SCUDirArg` called `scudat` to transfer `double` data, we simply set:

```
scudat.SetDataSize(8)
```

This class also defines a few extra methods to allow the OS interface to access the MPI datatype and other data-transfer parameters. These are straightforward, and are described in the `scu_dir_arg.h` header file.

2.8 The OS Interface

The subroutines defined in §1.3 have all been implemented in this version of MPI-SCU. As well as these, the MPI version adds a number of new interface routines:

```
void SCUCommsInit( void );
```

Initialisation: Parses the communication parameters, calls `MPI_Init`, etc. This will be called automatically when the code attempts to perform any `SCUDirArg` operations, but is supplied here so that it can be explicitly called at the start of the program.

```
void SCUGlobalSum(Type_tag t, size_t tsize, int n, void *ivec, void *ovec );
```

Perform a global sum directly using `MPI_Allreduce`. This avoids using the standard global sum calculation, which is implemented via a set of `SCUTrans` calls, and so this version will probably be more efficient on most architectures. The results are sent to all processors.
`Type_tag t` Indicates the type of the data: one of `TYPE_float` & `TYPE_int`.
`size_t tsize` Indicates the size of the individual floats or ints (in bytes).
`int n` The number of items to be summed.
`void *ivec` The input vector for the summation.

`void *ovec` The output vector for the summed result.

```
void SCURaiseError( char* errstr );
void SCURaiseError( const char* errstring );
```

Wrapper for the comms-error reporting mechanism. This currently prints the error to the standard output and then exits, but will eventually be made to use the QCDSP code's error reporting mechanism (the `Error` class).

Note that the implementation only identifies data transfers by their direction, but that MPI retains the order of communications. Therefore, multiple transfers in a single direction will work as long as the order of the send commands matches the order of the receives.

2.9 Test Programs

There is currently a single test program for the MPI-SCU layer, which also does not require the QCDSP code in order to work. The next stage will be add a test based on the original SCU-based global sum which will also compare this mechanism to the `SCUGlobalSum` routine. In the longer term, the aim will be to get one of the dynamical Wilson fermion test-codes running via MPI-SCU.

2.9.1 `tests/simple/commstest.C`

This tests the parsing of the communication parameters, and sets up a few strided and contiguous `float`, `int` and `double` transfers between processors. The data thus transmitted is checked for correctness at every stage. See the source files for more information.

2.9.2 `tests/stride/stridetest.C`

To test that the stride had been correctly implemented in MPI, this code (from GF) was compiled and tested on QCDSP, and then re-compiled and re-tested using the MPI-SCU layer here at EPCC. The results were identical. The original QCDSP code did not call SCUTransComplete, and in the MPI version this lead to incorrect answers. On QCDSP, it may be possible to skip the SCUTransComplete call under certain conditions, but this cannot be supported in the MPI implementation.

2.10 Unresolved Issues

- The entire `sysfunc` interface cannot be made `extern "C"`, because there are overloaded subroutines. If external code requires access to the communication-calls with un-munged names, some kind of wrapping or selective `extern "C"` usage may be required.
- In QCDSP, only the root processor could actually perform input and output, whereas under MPI all processors can write to disk. Therefore, as things stand, the code will produce

multiple identical output files. This may cause more serious problems for the *inputting* of data files.

- On a related point, in its current formulation, the MPI-SCU layer works be allowing *all* processors to access the various definitions files. This is simpler than only allowing a single processor to read the files and then distributing the data, but may cause problems on some platforms.
- The request handler defined in `mpi_requests.h` is currently rather poor, and has an arbitrary upper limit of 100 requests per node. This will be changed to handle things properly, via a linked list.