

**SHMOD: Shared Memory Object Data
A Template for Cluster Programming
Sarah Anderson sea@lcse.umn.edu
February, 2003**

**Laboratory for Computational Science & Engineering
University of Minnesota**

----- *d r a f t* -----

This document describes a usage of the SHMOD library for cluster computing using “Shared Memory Object Data”. It uses high bandwidth network access between nodes in a cluster to remotely read or write data. By implementing these objects as local file system files, we provide the application a persistent shared memory. Independent update by participating nodes provides us with a fault-tolerant way to use an arbitrary number of computing nodes to advance the global solution to a domain-decomposed problem.

Required Software

The required database software is currently “mySQL”, available in many versions and documented in <http://www.mysql.com>. The communication and I/O library is written in C. Most array data operations are written in Fortran-77, which typically provides superior performance for these.

SHMOD

The SHMOD library itself in its first versions required Myrinet (www.myri.com) hardware. The “GM” message passing library support is available, but it is also possible to use SHMOD with TCP/IP. In particular, Gigabit ethernet has been found to provide in excess of 90 Mbytes/second network bandwidth (ia32 Linux) and more than 60 Mbytes/second with Windows 32. The sample “Makefile” in the shmod3 release directory is easily modifiable for any Linux version; there will soon be a Microsoft Visual C++ “project” file supplied to simplify construction of the shmod library. See the README file in the shmod release directory for a short description of the routines available in the shmod library.

The example code is found in the “shppm” release directory. It is not simple in the sense of being extremely short - it is, in fact, a fully functional implementation of an efficient 3D explicit mesh hydrodynamics solver. It has the following properties:

- It is based on Shmod, which in turn relies on a functional mySQL database server which must be running on a machine accessible to all participating nodes.
- It assumes sufficient interconnection bandwidth between nodes. Here, sufficient means as much or more bandwidth as one has to local disk. Typical local disks read or write at 50-100 MByte/second, so sufficiently well interconnected Gigabit ethernet will work, as will Myrinet. The application “kernel”, here a simplified version of PPM (Piecewise Parabolic Method) hydrodynamics, must be able to update a sub domain on a node in the time it takes to read and write a context. By overlapping the computation with the pre-fetch and write-back of two other domains, the I/O time can be completely hidden.

The Template Code

1. sweeps.m4

Perhaps the best way to explain the example code is “bottom up”, starting from the computational kernel. The example code uses a directionally split PPM, “sPPM”, to update a 3-D sub domain of the global problem. The update for a single time-step then uses an “X-sweep”, “Y-sweep” and “Z-sweep” in succession. In order to proceed without requiring boundary communication, a diminishing series of redundant computation is performed (as is done in the PPM kernel itself) in these sweeps. To stave off communication even farther, two time steps are computed; this requires six 1-D sweeps on a sub domain. All this is done in the routines defined in “*sweeps.m4*”.

The header file included in most of the template is “*iq.h*”, which defines crucial sub domain definitions (sizes), and various limiting constants. In particular, “*NX*”, “*NY*”, and “*NZ*” are the fixed size of all sub domains. These are constants for simplicity, as are the number of sub domains, “*NTX*”, “*NTY*” and “*NTZ*”. sPPM requires *NBDY* boundary zones on each end of a 1-D strip to be updated, and *NBDY2* boundary zones in dimensions transverse to the 1-D sweep direction. One can see, then that the number of “fake” zones needed for two full time step updates is *NBDYF*. The result of a sub domain update is then a core array of “real” zones, and a copied set of redundant “fake”, boundary zones.

sweeps embodies the most efficient methods we have found to use cache-based microprocessors. It interleaves the physical quantities needed for PPM as *NVAR* consecutive fast-running array indices in the arrays *ddd0* and *ddd1*. It uses an update pattern per CPU of *pencils* of zones in each 1-D sweep direction to block memory references so as to make better use of memory cache. The pencils also are conveniently sized chunks of work to assign to CPUs within a node. This is done with *OpenMP* parallelization directives placed in the sweep logic, as well as in boundary extraction and copying routines. It is perhaps more complicated than one could imagine for a non-directionally split code, but any truly memory efficient 3-D array referencing program should reflect the reality of the microprocessor architecture.

2. Data Structures

The convention throughout is that related data are grouped in common blocks (Fortran), or structs (C), and that these are identically referenced by these include files:

| | | |
|--------------|--------------|---|
| iq.h | | Constants which are used by both the C and Fortran preprocessor |
| dynglobal.h | dynglobal.f | Dynamic data refreshed for each sub domain update |
| stglobals.h | stglobals.f | Static data refreshed when starting a node program |
| | bdrysin.f | The ‘unpacked’ input data for a context update |
| | bdrysout.f | The ‘unpacked’ output data from an update |
| | context.f | The ‘packed’ input and output to an update |
| statistics.h | statistics.f | Descriptive statistics computed as a byproduct to updating a context |
| taskorder.h | taskorder.f | Data describing the order in which to update contexts, used when deciding which task to update. |

3. Context level data structures

The *sweeps* routines are given data in a “packed” array *ddd0* which contains the required fake zones. The sweeps copy back and forth from *ddd0* to *ddd1*, eventually resulting (for the even number of sweeps, 6) back in *ddd0*. The *writeback.f* routines “unpack” the core zones to the array *ccc1* and associated boundary arrays *xl, xr, yb, yt, xlyb, xryb, xlyt, xryt*. Because the global domain is decomposed in two dimensions, (*NTX* and *NTY*) there are only 9 components to the unpacked context. This is a minor simplification of boundary handling code and a reasonable data decomposition. Note that *task* decomposition is still in (*NTX, NTY, NTZ*), that is, 3-D. Once the data is unpacked, it can be written-back with individual Shmod writes. Of course, these are non-blocking writes, and related functions which wait for write-back completion and so on are found in *contextio.c*. Routines which start context pre-fetch, wait for said prefetchs to complete, and pack contexts to *ddd0* are found in *copyer.f*.

Contexts, expressed as unpacked boundary arrays and problem zone arrays, are Shmod objects which are read and written from *storage hosts*. These hosts have sufficient disk and network bandwidth to serve as many concurrent updating nodes as are contemplated. Each such storage server runs a Shmod process (*sriod* for Myrinet, and/or *ipriod* for TCP/IP networks) which responds to computation node requests for data storage or retrieval. In the extreme cases, a storage server process can be present at every computational node, or at none, running instead on a separate set of storage server nodes. In this template code, the *hostname* and *root directory* of each object is specified at the time of creation of the object. (See *create_thing* in Shmod.) This is done by a round-robin assignment of contexts to the pool of storage hosts. Subsequent reads and writes do not specify the location of objects, this information is looked up in a data base query.

Periodic output is done with the LCSE’s “compressed dump” utilities expressed in the *tile_cdump.f* routine. *Tile_cdump* is called as needed as part of a data context write-back. The *cdump* routine uses a simple set of routines from Shmod which accomplish non-local blocking writes, found in the *nio.c* package. It can target any *sriod* or *ipriod* remote I/O server, so that output can be collected on a convenient host or hosts. The location and host of each output file is also entered in a data base table, for use by data tracking and migration utilities.

4. Initialization parameters and the main program

The main program is found in *driver.c*, which handles command-line parameters and the parse of a keyword-value text file, *inputdeck*, used to initially define run parameters. The interpretation of keywords should be obvious given the above discussion of storage hosts, output host and so on, but as always, the final authority is the well commented code itself.

Here is a sample input file, supplied in the template directory:

```
runname slip
# all cdumps go here
dumphost      user02
dumpdir       /scr/sanderso/cdumps
# contexts on each storage host
rootdir       /scr/sanderso/context
```

```

storagehost      clustor01
storagehost      clustor02
storagehost      clustor03
storagehost      clustor04

# mesh expansion over zones at each end
zexpand          10 1.10

#          dt          [safety  dtmin    dtmax] defaults: 0.8 0.0 1000
dtime            0.005
#          steps time
stop             20000    2.0
#          interval    [dtnext]
dtdump           0.02
#
# ----- shear turbulence setup, gamma=5/3
gamma            1.6666666666666666
# c=1, rho=1, gamma=5/3 p=sqrt(c*c*rho/gamma)
#shear  dens  prs                                vx  vy  zpos  noise
shear           1.0   .77459666924148337703  0.5 0.01 0.5  0.01

```

Lines beginning with “#” are comments.

5. Task assignment and control flow

The heart of the template is in *driver.c* and *lookforwork.f*. Overall control flow and the 3-stage prefetch, update and writeback pipeline is in C. Routines are called from with the driver main program to initialize, update, read and write contexts, and those routines are written in Fortran.

The *lookforwork* routine is responsible for querying the database for a “glob” of bytes comprising the dynglobal common block/structure. It contains various data needed for task state tracking, time-out handling, task ordering, and so on. Its basic function is to lock the global *dynglobal* data base, read it into the common block, decide what to do next, write the common block to the database and release the lock. Since this is a time-critical operation no lengthy computation is done here. This allows hundreds of nodes to cooperate in *lookforwork*, as each node performs this transaction relatively infrequently, typically less often than once each 10 seconds.

Future Plans

The techniques described here perform well on large problems computed on a single well connected cluster. Considering very large clusters, which allow us to store the entire problem domain in node memory, or clusters of clusters interconnected less well by WANs, leads us to modifying this scheme. We could provide the adaptation and fault-tolerance features on longer time-scales, by using redundant persistent store less often. Data replication (mirroring) and automatic context migration across cluster boundaries must also be considered. Finally, the performance of the unifying data base probably will have to be increased by coordinating multiple instances of data base servers.