IRS: Implicit Radiation Solver Version 0.9.1 Build Notes

UCRL-CODE-2001-010

Code Description

The implicit radiation solver code, hereafter referred to by the acronym IRS, solves the radiation transport equation by the flux-limited diffusion approximation using an implicit matrix solution. In fact, IRS is a general diffusion equation solver, but its flux limiter imposes the speed of light as the maximum signal speed, hence making it a radiation solver.

IRS uses the preconditioned conjugate gradient method (PCCG) for inverting a matrix equation. Three choices of matrix preconditioners are available: 1) no preconditioner, 2) diagonal scaling, and 3) two-step Jacobi. The PCCG method is

known not to be strictly scalable in a parallel computing sense, but its lack of scalability is well characterized and is accounted for in the benchmark problems as described later. Thus, it serves as a robust test of some aspects of the communication hardware and software on a parallel platform.

IRS is written in the ISO-C standard. The basic PCCG matrix solver is a modified version of that described in Numerical Recipes in C, Second Edition, by Press, et al. The included solver software uses message passing via MPI, OpenMP threads, or a combination thereof. Parallel mesh generation capability is self-contained, as well as the ability to take time-history data of relevant quantities during the course of a diffusion simulation. IRS contains an elaborate set of internal timing functions that allow developers to monitor which facets of a simulation are time sinks. These timers are portable and thread-safe, working with either MPI or OpenMP.

Often, parallel scaling studies are performed by expanding the scope of the problem being studied (e.g., making it physically larger). This is not indicative of real simulations where the physical size is fixed and problems become larger by refinement of the computational mesh on which the simulation is performed. It is this latter approach that is used in the IRS benchmark problems. This choice leads to the non-scalability of the PCCG method in use here: the matrix condition number, and hence the number of iterations, increases with increased resolution. However, we may include this factor in our figure of merit by dividing out the number of iterations.

The benchmark test problem is described as follows: a planar radiation wave diffuses through a regular rectangular mesh from one end to another (and out). The problems execute for longer than it takes to traverse the spatial problem. This forces the radiation iteration count to increase dramatically and is more stressful on certain aspects of parallel communications. The physical problem is a 10 x 10 x 10 cm3 mesh with variable resolution. IRS divides the physical mesh into domains, and typically executes one domain per processor, although it is possible to place more than one on a processor (domain overloading). It will NOT be happy if the number of spatial domains is LESS than the number of processors. For the benchmark it is suggested to use 25 x 25 x 25 computational cells per domain. This is a compromise between very heavy or very light computation-to-communication ratios. Typical processor counts, therefore, might be integers cubed {8,27,64,125,216,\x85}. For testing on LLNL ASCI Blue we restricted ourselves further to powers of even integers so the problems could be decomposed with either 4 MPI tasks per SMP node, or 1 MPI task with 4 threads per node. We have studied the benchmark at processor counts up to 1000 on the LLNL Blue platforms.

IRS will communicate large amounts of data ONLY for the domain surfaces. However, the PCCG method requires two global reductions of data per matrix iteration (sending at most four doubles each time). The global reduction speed and general surface communication speeds are the objectives of the benchmark (as well as the actual compute speed, of course). The figure of merit is to be the

speed per cell-iteration in ONE DOMAIN:

(execution time in microseconds) / (cells/domain * integral of iterations),

and results are to be compared for various problem sizes and communication paradigms if appropriate (MPI alone, or MPI + Threads being typical choices).

Prerequisites to Building the Code

This code depends on the following items

- Perl

  The make process requires several perl scripts. These are included in the tar file. Instructions on how to use them are detailed below.

- Gnu make (possibly)

  The makefiles used by this code work fine with Gnu make on linux, DEC OSF1, and IBM SP2 platforms. However the native 'make' on the IBM SP2 or the DEC OSF1 had troubles with some of the constructs in our makefiles.

  If your native make chokes, it may be necessary to compile Gnu make. This does not imply that the Gnu compiler or loader or any further Gnu tools are needed.

Optional libraries which may be built into the code

In order to create restart files, this code must be configured to use an installed version of the SILO data management I/O library. Without this library, the code will run and produce text files with benchmark numbers and timing information, but it will not be able to read or write data files which contain the problem state.

- Silo library

  The installation and distribution of Silo is documented in other benchmark codes.

Building the Code

- Unzip and untar the tarfile with commands similar to the following:

  gunzip irs.tar.gz

  tar -xvf irs.tar

  Assuming you untarred the file in your home directory, A directory named ~/irs will now exist. This directory will include several sub-directories.

- export PATH=~/irs/scripts:$PATH;

  Further steps require scripts which exist in ~/irs/scripts. The above command will add this path in the bash shell. Use whatever command is appropriate for your shell.

- irs_build build.space irs

  This command should be executed from your home directory, or wherever the irs.tar file was unttarred.

  This creates a "build space" where the code will be compiled. This build space contains links back to the source files in irs. Having a separate build space, rather than building directly in irs provides the following advantages.

  - Multiple build spaces may be made. For example

    irs_build build.compiler1 irs

    irs_build build.compiler2 irs

    Will create two build spaces, one for testing one compiler and the other for testing a different compiler.

  - If necessary the entire build space directory can be removed (ie rm -rf build.space) and the original source files are unaffected.

- cd build.space/build

  Move into the directory where the code will be built.

- irs_config

  This will list the systems for which makefiles are provided. Currently there are makefile for most LLNL OCF machines, including IBM AIX, and various flavors of the Chaos Linux system.

For each of these platforms, there is a configuration option to compile without the presences of the SILO library.

For several of these platforms, there is a configuration option to compile with threads, using the OpenMP 2 specification. That is, the compiler must honor a few simple OpenMP 2 pragmas.

So for example, here are a few configuration options listed by irs_config for the machine 'thunder' at LLNL.

- chaos_3_ia64_elan4 - standard compile on LC chaos_3_ia64_elan4 system WITH SILO
- chaos_3_ia64_elan4_threads - threaded compile on LC chaos_3_ia64_elan4 system WITH SILO
- chaos_3_ia64_elan4_threads_wo_silo - threaded compile on LC chaos_3_ia64_elan4 system WITHOUT SILO
- chaos_3_ia64_elan4_wo_silo - standard compile on LC chaos_3_ia64_elan4 system WITHOUT SILO

If you are testing this on a new platform, you will have to create one of the above makefiles and then modify it for your system. For example

irs_config sp2

Will create the Makefile which you should then modify (see the next step)

- Modify the Makefile

  The Makefile will need to modified by hand to fit your system. Items to find and change include:

  - GMAKE = /usr/local/gnu/bin/make -j 8

    This should be the path to Gnu make. The "-j 8" is used to perform a parallel make using 8 processes. This option may be removed or modified as desired.

  - SILO_LIBPATH = -L/usr/local/silo/lib
    SILO_INCPATH = -I/usr/local/silo/include

    Set these to reflect where the Silo library is installed

    If you comment out all the SILO lines in the makefile, this will create a code which will not compile in or require the existence of the SILO I/O library.

  - MPI_LIBPATH = -L/usr/local/mpi/lib
    MPI_INCPATH = -I/usr/local/mpi/include

Set these to reflect where MPI is installed

- OPENMP_LIBPATH = -L/usr/local/openmp/lib
  OPENMP_INCPATH = -I/usr/local/openmp/include

  Set these to reflect where OpenMP is installed

  If not compiling with OpenMP threads, this section should be commented out of the Makefile.

- OTHER_LIBS = -lother
  OTHER_LIBPATH = -L/usr/local/other/lib
  OTHER_INCPATH = -I/usr/local/other/include

  Any further platform specific libraries and associated include paths and library paths may be placed here.

- CC = mpcc
  LINK = mpcc
  CC_FLAGS_OPT = -c -O2 -qtune=604 -qarch=ppc -qmaxmem=16384
  CC_FLAGS_DEB = -c -g -qmaxmem=16384
  LINK_FLAGS_OPT = -bmaxdata:0x70000000
  LINK_FLAGS_DEB = -bmaxdata:0x70000000

  These are respectively:

  - The C compiler
  - The Linker
  - Compiler flags used to build an optimized code
  - Compiler flags used to build the code for used with a debugger
  - Linker flags used to link an optimized code
  - Linker flags used to link the code for use with a debugger

- ARCHIVE = ar -g -r -v
  ARCHIVE_X = ar x

  Set ARCHIVE to the command to replace or insert new file into an archive.

  Set ARCHIVE to the command to extract a file from an archive.

- Review the rest of the Makefile to see if anything else needs changed.

- make depend

  This command is required and actually performs two functions.

  - It creates a sub-directory Makefile in each of the ~/build.space/sources/xxx directories.

- Runs 'makedepend' on these sub-directory Makefile to create the dependency list used to build the code.
- make

  This will build an optimized code. Use 'make debug' to build a code for use with a debugger. The executable will be place in either ~/build.space/codes_debug or ~/build.space/codes_opt, depending on which was made.

  You may make both an optimized and a debuggable version of the code in the same build space. The resulttant libraries and executables are kept in separate directories.

UCRL-CODE-2001-010