

Conventions for using PIERS

Philip T. Keenan

December, 1991

TR91-40

Conventions for using PIERS

Philip T. Keenan*

December 12, 1991[†]

Contents

1	Introduction	1
2	Directory Organization	2
3	Using PIERS	3
3.1	Dimensioning the source code	3
3.2	Compiling the program	6
3.3	Running numerical experiments	7
3.4	Graphics Commands	9
3.5	Sample Run	10
4	Using a remote Hypercube	11
5	Shell Script Short Cuts	12

1 Introduction

This document is intended to define new conventions for using PIERS, different from those described in the PIERS user manual. The changes reflect the changes we have made to the original PIERS distribution. We have:

*Department of Mathematical Sciences, Rice University

[†]Version 6

- revised the dimensioning program to use command line arguments and to understand equations.
- split the source files into individual subroutines.
- fixed several bugs which prevented using more than 16 processors.
- allowed rectangular mesh orderings in addition to gray code nearest neighbor topologies.
- modified the i/o and graphics for use on Sparc stations rather than on PC's.
- organized dimension files and data sets.
- wrote script files based on this organization, which can re-dimension several versions of PIERS and run hundreds of numerical experiments in a convenient manner.
- wrote a graphics display program for viewing graphical results with X-11 and with PostScript.

Adhering to the conventions described in this document will facilitate keeping track of code modifications, dimension files, dimensioned versions of the program, data sets and output results. It will also allow the use of convenient shell scripts for re-dimensioning PIERS and running test cases.

The rest of this document will describe the new directory organization, followed by sections on how to dimension, compile and run PIERS. We also describe how to use PIERS on remote Hypercubes. Finally, we describe the shell scripts that make it even easier to do these tasks.

2 Directory Organization

Within the `piers` directory we use the sub-directories described below. Certain of these sub-directories, namely `make`, `output`, `graphics`, and `restart`, are additionally partitioned into sub-sub-directories, which are named after dimension files. This allows us to keep separate several differently dimensioned versions of PIERS.

`doc` Documentation files like this manual go here.

src The un-dimensioned source files go here. Permanent modifications should be made to these rather than to the dimensioned versions in the **make** directories. This directory also holds the master **makefile**.

dimen This subdirectory holds the dimensioning files for use with the **dim** program, which replaces the PIERs **dimen** program. It also holds the **buildPIERS** shell script.

make Running the dimensioning program on the files in **src** produces dimensioned versions which are stored in sub-directories of this directory, indexed by the name of the dimension file. These dimensioned source files are then compiled to produce executables, which are kept with them. The **make** directory also holds the **buildPIERShosts** shell script.

data sets This subdirectory holds data sets for experimental runs. It also holds the **runPIERS** shell script.

output Text output files go in sub-directories of this directory, indexed by the dimension file name.

graphics Graphics output files go in sub-directories of this directory, indexed by the dimension file name.

restart Any input or output restart files go in sub-directories of this directory, indexed by the dimension file name.

Note that the above mentioned shell scripts rely explicitly on this directory organization scheme.

3 Using PIERs

3.1 Dimensioning the source code

To facilitate modification of numerical routines in PIERs we have broken the source code up into small files each containing just one subroutine. These are kept in the **src** subdirectory. These used to be dimensioned using the **dimen** program; however, that program uses a somewhat restrictive input style and requires editing the dimension file when only re-dimensioning a subset of the files. Therefore we are superceding FORTRAN based **dimen** with a C++ program called **dim**, which performs the same tasks but is much more flexible. It uses command line arguments to specify which files to operate on.

3.1.1 Dimension Files

We provide a number of sample dimension files, including:

- a 20x20x20 (8,000 elements), 4–16 processors.
- b 20x50x50 (50,000 elements), 4–32 processors.
- c 30x100x100 (300,000 elements), 4–32 processors. This one turns out to be too big for our Hypercube's memory.
- d 30x100x100 (300,000 elements), 16–128 processors.
- e 50x150x150 (1,125,000 elements), 64–128 processors.

Note that PIERS is not currently able to use fewer than 4 processors, and they must form at least a 2 by 2 array. This is in part because it runs out of memory in the single processor case, and in part because it uses node broadcast calls which fail when there are no other nodes to broadcast to.

3.1.2 Dimensioning files by hand

Suppose you wish to re-dimension all the source files using dimension file **a**. Go to the `src` directory and type

```
dim -d a -- *.*
```

This will put the resulting dimensioned files in the `./make/a` directory, which is the correct place for them under our directory conventions.

To re-dimension just one or a couple of files, simply specify their names on the command line. Thus,

```
dim -d a -- divide.b
```

would just re-dimension the subroutine `divide`.

The program takes several arguments. The `-d` argument lets you specify the name of a dimension file; recall that all dimension files are kept in the `dimen` subdirectory. The `-o` option lets you specify an output directory, relative to where the source files are. The syntax is:

```
dim [-d dimen-file-name] [-o output-directory] -- source-file(s)
```

The format of the `dimen` file has been substantially revised. You now only specify symbol definitions, as the directory and file specifications come from the command line. Unlike the old `dimen` program, this version acts on any file regardless of its suffix, and does not change its name.

The character `#` begins a comment, which lasts until the end of the line. White space is ignored except to separate tokens. Symbol definitions use the format:

```
name = value
```

The value portion can be a double quoted string or it can be an arithmetic expression involving both literal numbers and previously defined identifiers. Thus we can now use dimension files in which we set the grid size and the number of processors, and then all the other definitions are *automatically* updated — a major improvement over `dimen`. For example, the four definitions

```
maxXintervals = 20
maxYintervals = 50
maxZintervals = 50
maxProcessors = 32
```

are typically the only ones you need to change, because subsequent definitions like

```
YGB = maxYintervals
YG2 = YGB + 2
```

automatically produce the desired effects, namely here `YGB=50` and `YG2=52`. Expressions can be as complicated as needed and can extend to the end of the line. Expressions can also contain function calls. For example, one could define

```
sample = 4+(maxYintervals/2)+minSquare*minSquare
sample2 = even(max(Y/2,Z/2));
```

Here `sample2` computes the maximum of the two quotients, then rounds it up to the next even integer. Unlike the full K-Language interpreter, however, `dim` does not let the user define functions at run time.

Notes

Symbol names must start with a letter, followed by any number of letters and digits. Right hand side values must be integers or double quoted strings.

Identifiers or double quoted strings on lines by themselves are printed when encountered.

The special symbol `dimFile` is assigned the name of the dimension file.

The option `-D` turns on debugging support. Lines in the source files beginning with `$DBG` in the first column are ordinarily commented out by the automatic definition `$DBG="CDBG"`. With the `-D` option, we instead get `$DBG=" "`, so that these lines are now activated.

Although the C++ program `dim` provides greater functionality than the FORTRAN program `dimen`, it is substantially smaller and faster. The Sparc station executable takes up only 64 Kbytes, as compared to 152 Kbytes for `dimen`. The `dim` program can dimension the entire PIERS source (335 Kbytes) in just 1.2 CPU seconds, while `dimen` requires 11.1 CPU seconds.

Typing `dim -usage` reveals several other advanced options for the `dim` program, including all those associated with my basic C++ library and error handler. In particular, the options `-3` and `-a` together allow the program to mimic the three character names used by `dimen` in situations where they are not otherwise distinguishable as tokens. This allows us to fix the `$VEC` and `$SCL` symbols. We also note that when the `-3` option is not used, any symbol in the source may be followed by a space (to separate it from following characters), which will be suppressed in the output. Thus you retain complete control over the output text.

3.2 Compiling the program

Source files use the following suffix naming conventions:

- h Host FORTRAN routines.
- n Node FORTRAN routines.
- b FORTRAN routines used by both host and node; host copy.

B FORTRAN routines used by both host and node; node copy.

for FORTRAN include files.

c C routines used by the host only.

Any time you (re)dimension a file with the `.b` suffix, you must make a copy of it in the appropriate `make` subdirectory with a `B.B` suffix. This is done with the shell script `bcopy`. Run it in a `make` subdirectory such as `make/a` as

```
bcopy *.b
```

to update all `.b` files, or as for example

```
bcopy divide.b
```

to update just certain files. You should use the latter form when possible, as `make` will re-compile all the resulting `.B` files, since they will look recent to it.

To compile the program we use `make`. This has the nice point that if you are making changes to code, only those source files that have been re-dimensioned or otherwise modified since the last compilation will be re-compiled. Since we have split the source code into small pieces, re-compiling will generally be very quick.

In a subdirectory such as `make/a`, type

```
make node
```

on the Sparc station to re-compile the node program, and type

```
make host
```

on the SRM of the Hypercube to re-compile the host program. Re-compiling from scratch (after re-dimensioning all the files) takes about 5 minutes for the node program and 10 to 15 for the host program.

3.3 Running numerical experiments

After compiling, the appropriate `make` subdirectory contains the host and node programs `resh` and `resn`. Log into the cube, get a subcube, and run `resh`, as for example with

```
getcube -t 16  
resh
```

The host program will then ask for the name of the run, the number of Y and Z processors, and a run code. In this example we assume that you typed `test71` for the run name, 4 and 4 for the processor array dimensions, and `a` for the run code. This will cause it to use the following files, in which we assume for instance that the program was dimensioned with dimension file `b`:

- It will read `datasets/test71.dat` for its input file,
- write to `output/b/test71.4-4a.out` for its output,
- use `restart/b/test71.4-4a.ri` and `restart/b/test71.4-4a.ro` as input and output restart files, and
- any graphics output will be placed in `graphics/b/test71.4-4a.gr`.

Note that each copy of the PIERS executable knows the name of its dimension file, so that it automatically uses the correct subdirectory, such as `b` above.

The use of the run code allows you to make multiple runs on the same program and data set while debugging, and produce uniquely named output files. If the output files already exist the program will stop rather than overwrite them.

To facilitate scaling studies PIERS now also appends a short summary of timing information to the file `datasets/summary` each time it is run. Each block of summary information contains the following information:

- The name of the corresponding output file.
- Timing statistics including the cumulative run-time and the average time per time step.
- A list of those pieces of the step time which contribute more than their fair share of time to it.
- The communication time, as originally defined in PIERS and as adjusted to include all otherwise ignored pieces of the step time.
- The efficiency of the run based on both estimates of communication time, as well as estimates of the sequential run time and the resulting speedup.

3.3.1 Data Sets

To facilitate scaling experiments we no longer specify PROCY and PROCZ in the data sets themselves, but at run time as inputs to PIERS. This allows us to write shell scripts that can run numerous test cases without re-editing.

Among our standard scenarios are:

ptk1 A 4x18x13 mesh (756 elements) with an irregular cross section.

test7 A 6x24x24 regular mesh (3456 elements).

test7m A 12x36x36 regular mesh (15,552 elements).

test7l An 18x48x48 regular mesh (41,472 elements).

t100 A 30x100x100 regular mesh (300,000 elements).

mega A 50x150x150 regular mesh (1,125,000 elements).

PIERS now does some checking to see that the data set will actually fit the bounds specified when it was dimensioned.

3.4 Graphics Commands

To cause graphs to be written to the graphics output file, type one or more of the following commands while the program is running. In these commands *m* and *n* stand for integers.

a Well Oil Rates vs. Time

b Well WOR vs. Time

c Well Oil and Water Rates vs. Time

d n Contour Oil Pressure in Grid Layer $n = 1 \dots 30$.

e n Contour Water Pressure in Grid Layer $n = 1 \dots 30$.

f m n Contour Oil Pressure section from Well *m* to Well *n*; well numbers are 1 through 9 inclusive.

g m n Contour Water Pressure section from Well *m* to Well *n*; well numbers are 1 through 9 inclusive.

h m n Change Flow Rate of Well *m* by *n* percent; well numbers are 1 through 9 inclusive; the percent must be an integer in $[-9, 9]$.

The resulting graphics file contains only printable ASCII characters. Each line begins with an abbreviated command name, such as LI to draw a line, followed by data. This is the same data the original version of PIERS produced, stripped of Escape codes. Anyone can then write a translator for such files into their favorite graphics language, such as PostScript. Such translators are not part of PIERS. I use a translator I wrote in C++ to convert the files into my own graphics format, called HGL, from which I can display them on screen under X11 windows, or on paper via a PostScript printer. People with access to an Intel Hypercube which has the extra cost hardware socket package should be able to add interactive graphics capability without much extra coding.

3.5 Sample Run

The following transcript describes how to make a new dimension file, re-dimension the source code, re-compile it and run it.

```
cd piers/dimen
cp a e
emacs e
```

Here you edit the new dimension file *e* by changing the problem size in the first couple lines. Then proceed with:

```
cd ../make
mkdir e
cd ../src
dim -d e -- *
cd ../make/e
bcopy *.b
make node
cd ../..
mkdir output/e
mkdir graphics/e
mkdir restart/e
```

Now log into the Hypercube and do

```
cd piers/make/e
make host
```

When both are finished compiling, do the following on the Hypercube:

```
getcube -t 16
resh
```

Enter the name of the run, such as `test71`, and the size, such as 4 by 4, and a code such as “a”. When the run is finished, try for example

```
more output/e/test71.4-4a.out
```

to see the output file.

The good news is that the shell scripts described below simplify all this down to just a couple commands, and handle more than one dimension file or data set at a time!

4 Using a remote Hypercube

To run PIERS on a remote Hypercube:

1. Create a PIERS directory on the remote machine and make sub-directories for `make`, `datasets`, `output`, `graphics` and `restart`.
2. To use a particular dimension file like `a`, create the sub-directories `make/a`, `output/a`, `graphics/a` and `restart/a`.
3. Do the dimensioning process on the local machine.
4. You can also do the compiling on the local Sun and local Hypercube.
5. Using `ftp`, bring over the executables `resh` and `resn` from the `make` subdirectory, eg. `make/a`.
6. Also using `ftp`, bring over one or more data sets, and the script file `runPIERS`.
7. Run the `runPIERS` script as described below.
8. When done, you can `ftp` back the output and graphics files produced by the runs.

To use ftp, type `ftp` followed by the full name of the computer you wish to connect to. Type your name and password for that computer, then change to the appropriate directory with `cd`. Use the command `get filename` to get a file, or `mget *.*` to get all the files. When finished, type `quit` to end the ftp session. Note that `put` and `mput` transfer files in the opposite direction to `get` and `mget`. To speed up transferring several files, type `prompt no` before the `mget` or `mput`. To transfer binary files, type `binary` prior to transferring any files.

5 Shell Script Short Cuts

The script `buildPIERS` in the `dimen` directory will dimension, `bcopy` and (node) compile one or more copies of `PIERS`, automatically creating all necessary sub-directories. Execute it, giving as command line arguments the names of one or more dimension files. For example,

```
cd piers/dimen
buildPIERS a b
```

will re-dimension the host and node programs and re-compile the node program for dimension files “a” and “b”.

To re-compile the corresponding host programs, log onto the Hypercube and execute the `buildPIERShosts` script in the `make` directory. Give it the same dimension file names as arguments, and it will re-compile the host programs. For instance,

```
cd piers/make
buildPIERShosts a b
```

will re-compile the host program for dimension files “a” and “b”.

To run a particular version of `PIERS` on one or more data sets, use the `runPIERS` script. Log onto the cube and use `getcube` to allocate some number of nodes. Give as arguments the dimension file name, the processor array sizes, the run code, and one or more data set names. For example,

```
cd piers/datasets
runPIERS c 4 4 a *.dat
```

will run the `c` version of PIERS on a 4x4 processor array using all the data sets and with run code "a".

To erase the resulting output files, use for example

```
cd piers/datasets
cleanPIERS c 4 4 a *.dat
```

To facilitate scaling comparisons, we also provide shell scripts to run all the standard test cases at all the different dimensioning levels against all possible numbers of processors. In particular, we provide:

run4 The 2x2 cases on a 4 processor cube.

run16 The 3x3 and 4x4 cases on a 16 processor cube.

run32 The 5x5 cases on a 32 processor cube.

run64 The 6x6, 7x7, and 8x8 cases on a 64 processor cube.

run128 The 9x9, 10x10, and 11x11 cases on a 128 processor cube.

To use these scripts you must first execute the command

```
setenv code A
```

where **A** may be replaced by any character string you wish to use as the run code. Then go to the `datasets` directory and type for instance `run128 &`. This will call `getcube` and `relcube` for you, as well as run several test cases. All output, including error messages, is automatically redirected to two files, called in this case `log128` and `nodeLog128`. This means you can log out while the run is happening in the background, and the next day examine the log files to see what happened. However, to be a good citizen you should log back in when you think the job is through and do a `relcube -a` yourself, explicitly, just in case some strange error causes the run to hang and never release the cube.