ℙfortran Users Guide

a ℙarallel extension of Fortran

Terry W. Clark
University of Houston

Rick Kufrin
University of Illinois

Edward Robinson
University of Houston

Release 1.4.1

June 15, 1995

Contents

1 Introduction

The IPfortran programming approach eases the task of writing parallel programs for massively parallel computers by providing constructs to coordinate non–local memory accesses. At IPfortran's center is an operator that specifies the location of off–processor variables and an operator for aggregate operations. Portability and ease of programming are achieved by unifying off–processor accesses with the guarded–memory model [2]. Because the programmer writes low–level *node code* with IPfortran, the loss in performance compared to using the lowest–level primitives is typically negligible.

This document discusses the installation, maintenance, invocation, and runtime systems of IPfortran. For details about using IPfortran, please see the IPfortran *Reference Manual* [1].

2 Users guide

2.1 Installing

The IPfortran default target machine can be hard–wired at compiler time with the Makefile.pf variable `TARGET` where valid machine types may be taken from: IPSC, NCUBE, CM5, PVM3, KSR, and SGI. This default machine can be overriden with compiler switches (see below).

2.2 Invoking

The IPfortran translator **pfc** can be invoked with

pfc [*switches*] f_1.pf [f_2.pf [...]]

or alternatively

pfc [*switches*] $\star$.pf

where:

f_i.pf: the name of the IPfortran (or FORTRAN77) source file; this filename *must* have *.pf* as a suffix.

switches: Optional command line switches listed below.

2.3 Include files

Optional include file search directories can be used with the syntax:

$pfc -Id1 -Id2 ... -Idn myprog.pf

for directories d1 ... dn.

Up to 10 directories can be specified with one directory per I flag. The directories are searched in order of occurrence; the current working directory is always searched and it is searched last unless -I. is the first include search directory. The maximum nesting depth of include files is 10.

2.4 Switch summary

The following switches select the indicated target system for code generation:

-ipsc	Intel
-ksr	KSR-1; uses library SHMSG or TCGMSG.
-sgi	SGI; uses SHMSG.
-cm	CM5.
-pvm	PVM.
-ncube	NCUBE.

The remaining switches are

-I	Search this directory for include files.
-scratch <*n*>	Number of bytes for IPfortran buffer (default 100000).
-T string	Use *string* as variable name for THISNODE (default).
-i<*n*>	Default integer size of n bytes.
-r<*n*>	Default real size of n bytes.
-d<*n*>	Default double precision size of n bytes.

2.5 Variable sizes

The default variable sizes, in bytes, used by IPfortran are given below. These sizes can be overridden with the compiler switches described above. (See Section 2.11.2 for KSR-1 details.)

machine	integers	single precision	double precision	complex
KSR	8	8	16	8
Others	4	4	8	4

2.6 IPfortran Library

System-dependent sends and receives are made from the IPfortran library, called *pflib*. There are several libraries supported by this version:

pflibintel	Intel hypercube, simulator, Touchstone Delta, and Paragon.
pflibshm4,8	KSR-1 shared-memory message passing.
pflibksr	KSR-1 TCGMSG.
pflibsgi	SGI multiprocessor shared-memory message passing.
pflibpvm3	PVM based for networks of workstations or Cray multiprocessors

Consult your local *pfc* man page for details on the naming conventions and library locations.

The IPfortran library, or its equivalent, must be linked to (loaded with) a program translated by the IPfortran translator.

2.7 Scratch memory and variables

Temporary memory for arrays uses a block of storage declared with the array *pf_ib*. The default size of this storage can be modified using the **-scratch** switch. This storage is re-typed as needed through the use of equivalence statements.

IPfortran generates scalar variables for buffering or naming quantities. These variables are named from the base string *o10000* with successive variables incrementing the suffix *10000*. Clashes can occur if these variable names are used as *implicitly* typed variables in your program.

Labels are also generated by IPfortran in generating FORTRAN77 loops. These are guaranteed to be unique to the program section.

2.8 Intrinsic names

In addition to the intrinsic functions used by the target processor, IPfortran has a reserved variable-name set.

THISNODE Processor number of the processor running the code.

THISPROC Process number of the process running the code.

NUMNODE Total number of processors.

The current implementation of the compiler also imposes some additional reservations, although these may be cancelled in a later version.

pflib IPfortran library module.

pf_ib Integer buffer.

pf_sb Integer2 buffer.

pf_rb Real buffer.

pf_db Real8 buffer.

pf_cb Character buffer.

pf_xb Complex buffer.

pf_inctag Tag increment subroutine (pflib).

pf_snd Send subroutine (pflib).

pf_rcv Receive subroutine (pflib).

iqdim Cube dimension subroutine interface.

2.9 Using IPfortran

IPfortran translates the input *.pf*, producing output into a *.f* file. Code not involving IPfortran extensions is output "as is," including comments and blank lines. Code generated from translating statements involving {} or @ adheres to the rules of FORTRAN77.

There are three levels of advisory messages with IPfortran: fatal, error and warning. *Fatals* result in immediate termination of the translator with a diagnostic message and `.f` file removal. *Errors* also result in `.f` file removal, but the translator will continue processing the input. *Warnings* result in advisorys alone with output written as usual to the `.f` file.

We emphasize that the intrinsic variable set for IPfortran includes variables such as *thisnode.* These can be used freely in any subroutine or function that has been processed by IPfortran – redeclaration of these intrinsic variables will result in error.

IPfortran can be used on one, or all, of the modules composing a program with the exception of the entry point to the Fortran program: *The main program (entry point to a program) must be processed by the IPfortran processor.*

2.10 Sample Makefile for the iPSC/860

Having added another file-type suffix slightly complicates the compilation process. The *make* program, however, can be used to automatically invoke the correct sequence of compilations. With the following Makefile, the *make* program will automatically check dates for the file sequence *planet.pf*, *planet.f*, *planet.o* and the executable *planet*, and recompile as necessary.

sample makefile:

```
 PFC = pfc
 F77 = if77
 PFLAGS = -scratch 1000000
 FFLAGS = -O2 -node
 LFLAGS = -node
 LIBS = -lpf
 .SUFFIXES: .pf .f .s .o
 .f.o:; $(F77) $(FFLAGS) -c $*.f
 .pf.f:;$(PFC) $(PFFLAGS) $*.pf
 .pf.o:;$(PFC) $(PFFLAGS) $*.pf
$(F77) $(FFLAGS) -c $*.f

 planet:$$@.o
$(F77) $(LFLAGS) -o $@ $@.o $(LIBS)
```

2.11 IPfortran Programs and the KSR

The KSR language and operating system directly support parallelism in threads executing within a shared address space [4]. The separate address space feature of the SPMD model can be achieved with a multiprocess implementation of the application communicating through shared-memory. Currently IPfortran uses two such systems: TCGMSG [3] and SHMSG (see below).

2.11.1 IPfortran and TCGMSG

TCGMSG has experienced problems with O.S. upgrades. We recommend using the SHMSG library (Section 2.11.3).

To run a compiled program on the KSR, the program is passed as an argument to the TCGMSG program *parallel*. Suppose we want to run the program *baton* from the current working directory. In this directory, the file *baton.p* must be present, or created, prior to issuing

$ parallel baton

The file *baton.p* contains:

```
# This template is piped through sed to replace
# a with the name of the program.
# Add other hosts and processes as desired.
clark bubba 2 baton /home/clark/cwd
```

giving the user name, the host name, the number of processors, the executable name and the directory from which to run the program. Be sure to check the local system policies for processor set allocation.

A checklist for making the KSR executable with IPfortran are:

- Use the **-ksr** flag with IPfortran translator,
- Use the *KSR* version of *pflib.f* (or libpf.a),
- Compile with the **-para** and **-i4** flags,
- ld with the libraries *-ltcgmsg* and *-lrpc.*

A sample Makefile, for the IPfortran program *baton*, follows.

```
LIBDIR = /usr/local/lib
TCGMSG = $(LIBDIR)/libtcgmsg.a
LIBS = -lrpc -lpthreads
FFLAGS = -i4
LFLAGS = -para
PFFLAGS = -ksr
F77 = f77
PFC = /usr/local/bin/pfc

.SUFFIXES:      .pf .f .pc .c .s .o
.pf.o:; $(PFC) $(PFFLAGS) $*.pf
        $(F77) $(FFLAGS) -c $*.f
.f.o:; $(F77) $(FFLAGS) -c $*.f
.pf.f:; $(PFC) $(FFLAGS) $*.pf

baton:   baton.o pflib.o
        $(F77) $(LFLAGS) -o baton baton.o pflib.o $(LIBS) $(TCGMSG)
```

Note the **-i4** switch is required; 8–byte integers are not supported under TCGMSG, but are for the KSR SHMSG implementation described in Section 2.11.3.
There are working examples on the TCAMC KSR at /usr/local/src/pf1.x/suite. A common mistake in using the KSR IPfortran is to execute the application without the *parallel* command described above. A IPfortran program will execute apart from parallel, but not correctly.

2.11.2 The KSR-1 and IPfortran variable sizes

IPfortran interprets variable sizes according to the specific architecture. The size of integers, reals, double precision can be controlled with compiler flags, overriding the IPfortran defaults. We recommended that variable sizes are consistent throughout a program, that is, all reals the same size, all double precision the same size, and so on. With experience, however, it's possible to use almost any combination of sizes permitted by the native compiler.

The following table is useful for determining how to size variables in your program. It shows the default IPfortran variable sizes for the KSR-1, and the effect of IPfortran switches. The 3rd and 4th columns respectively. Note that a dash indicates no effect.

IPfortran variable sizes for the KSR-1			
type	default	-i4	-r8
integer	8	4	–
real	8	–	–
double	16	–	8
complex	8	–	–

2.11.3 IPfortran and SHMSG

The SHMSG system is a stand-alone library of message-passing routines for shared-memory architectures. Internally, it is similar to TCGMSG, however, there are differences in the user interface. In addtion, the SHMSG implementation supports 4–byte or 8–byte integers (see Section 2.5 for defaults). The usage described here applies to both KSR-1 and SGI multiprocessors.

Two libraries are required for running a SHMSG–based program on the KSR: the standard IPfortran library and the message–passing library. The standard library comes in 4–byte and 8–byte versions, corresponding to 4–byte or 8–byte integer native Fortran. These libraries are

1. *pflibshm4.f*
2. *pflibshm8.f*

The message passing library is called *shmsg.a.*

2.11.4 Compiling programs for SHMSG

A sample Makefile follows, however you are encouraged to check your local machine's man page for the location of test programs and Makefiles.

```
F77 = f77
CC = cc
INC = .
PFC      =       /home1/tclark/bin/pfc
SHMSG    =       /home1/tclark/x/shmsg.a
```

```
LIBS    =       $(SHMSG) -para

# 4-byte integers
PFFLAGS      = -ksr -I$(INC) -scratch 10000 -i4
FFLAGS       =       -i4 -noopt
PFL     =       pflibshm4.o

# suffix rules
#
.SUFFIXES:      .pf .f .pc .c .s .o

.c.o:; $(CC) $(CFLAGS) -c $*.c
.f.o:; $(F77) -I$(INC) $(FFLAGS) -c $*.f
.pf.f:; $(PFC) $(PFFLAGS) $*.pf
.pc.c:; $(PC) $(CFLAGS) $*.c
.pf.o:; $(PFC) $(PFFLAGS) $*.pf
        $(F77) -I$(INC) $(FFLAGS) -c $*.f

test4:  test4.o $(PFL)
        $(F77) -o test4 test4.o $(PFL) $(LIBS)
```

2.11.5 Specifying processor configurations for SHMSG based programs

A configuration consists of the number of processes (P) and the number of cells (C) on which the processes will execute. Processes will be bound to some cell. If P > C, processes will be cell mapped many-to-one.

A third parameter here is the number of cells in your pset (actual_C), or the "machine" that your application sees. If C > actual_C, SHMSG will adjust and inform you that intra-application pset timesharing will occur.

Intra-application timesharing is not recommended at this time.

Applications can have problems with intra-application pset timesharing, especially when P > 2C. We have found that either it works, or it doesn't. We discourage this usage.

The configuration, then, conists of two integers contained in a file called Config (upper case 'C'). This file must be in the directory where program executes. The values of P and C are integers on the first line of this file, e.g., a first record of "8 8" specifies 8 processes to be executed on 8 processors.

Processor configurations should be powers of two for Pfortran compatiblity. That limitation will disappear with an expanded Pfortran communication library. Non-power of two processor operations will work with SHMSG in any case, and with Pfortran provided global exchanges are not used there.

2.11.6 Running SHMSG based programs

Type the program name in the directory where the Config file resides. Input and output behave as usual.

2.11.7 Stopping SHMSG based programs

So that your application releases system resources, a cleanup routine must be invoked prior to normal termination. The Pfortran compiler will translate a Fortran **stop** to a *pfstop* which in turn invokes the routines to release resources. A Fortran program should terminate with a Fortran **stop**.

For interactive processes, use ctl-c. If the process is not attached to a terminal, send it the SIGINT signal, i.e. kill -2 <pid>. If all else fails, use kill -9, with the disadvantage that the processes cannot release system resources when hit with the SIGKILL signal.

2.11.8 Things to look out for

Hung program: If your program hung, or abnormally terminated, take a look at system ipc resources using the command "ipcs." If this seems like too much trouble, just run bubba:: `~clark/bin/rmem`. This will remove the shared-memory segments that the message-passing system used, and did not release, due to drastic termination, e.g. kill -9.

Intra-application pset timesharing: When an application's processes timeshare cells, a less efficient algorithm is used for coordinating access to shared data in the message-passing system. This stems from a bug in a hardware mechanism used for enforcing mutual exclusion.

2.12 KSR specific limitations

Using IPfortran on the KSR results in efficent, and portable programs. There are, however, some limitations that you should be aware of. If any of these limitations impede your work, then again, submit comments to the system administrator; software solutions are available for all of the limitations listed below.

- Using *IMPLICIT NONE* will result in compiler warnings as the KSR switch does not produce declarations for some intrinsic KSR functions. Workaround: put the declarations in the IPfortran source.
- Global sum algorithms require power of two processors, e.g. 1,2,4, ... Using other than powers of two processors will result in erroneous results for IPfortran global sum operations. Workaround: use powers of two total number of processors, or do not use the IPfortran intrinsic global sum operations.
- The KSR debugger is not operational for IPfortran programs. Workaround: Use print statements or debug on a Intel hypercube.

2.13 IPfortran Programs and PVM

The PVM version of IPfortran is layered on the Oak Ridge version of PVM (version 3 or later) [**?**] and requires the supplied library *pflibpvm3* as well as the PVM libraries [1].

The PVM version of IPfortran has been used on networks of workstations, including HPs and SGIs, and on an eight–processor Cray EL.

Note that vendor-supplied versions of PVM may not be compatible with the version used in IPfortran; refer to vendor documentation for details. If you experience problems, you may wish to obtain and install PVM from the Netlib archive.

2.13.1 Compiling and linking programs for PVM

Translation and compilation of IPfortran programs for PVM is as usual. Supply the switch **-pvm3** to *pfc*. You must include the following additional libraries at link time:

- The IPfortran library for PVM, *libpfpvm3.a* (supplied with the distribution).

[1] PVM software can be obtained electronically via anonymous ftp to `ftp://netlib.att.com/netlib/pvm3` or via the World-Wide Web at URL `http://www.netlib.org/pvm3/index.html`.

- The Fortran PVM3 interface library, *libfpvm3.a*.
- The PVM3 group services library, *libgpvm3.a*.
- The base PVM3 library, *libpvm3.a*.

Supply the libraries to the linker in the order presented above.

2.13.2 Sample Makefile for PVM

Here is a sample Makefile for PVM (locations and names of object files and libraries may differ on your system):

sample makefile:

```
EXE          = PFNODE
PFC          = pfc
F77          = f77
LD           = $(f77)
PFFLAGS      = -pvm3 -scratch 1000000
FFLAGS       = -O2
LIBDIRS      = -L/usr/local/Pfortran/lib -L/usr/local/pvm3/lib
LFLAGS       = $(LIBDIRS)
LIBS         = -lpfpvm3 -lfpvm3 -lgpvm3 -lpvm3
.SUFFIXES: .pf .f .s .o
.f.o:; $(F77) $(FFLAGS) -c $*.f
.pf.f:;$(PFC) $(PFFLAGS) $*.pf
.pf.o:;$(PFC) $(PFFLAGS) $*.pf
$(F77) $(FFLAGS) -c $*.f

$(EXE):$$@.o
       $(LD) $(LFLAGS) -o $@ $@.o $(LIBS)
```

Please note the following:

1. IPfortran and PVM libraries should be specified in the order shown. You can indicate alternate directories for library locations with the **-L** switch.

2. The PVM mechanism for spawning node programs (PVM3 routine **pvm_spawn**) requires a character string which names the executable to run on nodes of the virtual machine. This is "hardwired" as the name **PFNODE** in the IPfortran initialization routine **pf_com** and is therefore used in the link step above (*make* macro **EXE**). You can specify an alternate name by editing *pflibpvm3* and rebuilding the library.

2.13.3 Running PVM IPfortran programs

IPfortran programs for PVM follow the same steps for execution as other PVM programs.

1. Initiate the PVM daemon, either directly or using the PVM console.
2. Define the virtual machine, using PVM console commands or by supplying a host configuration file to the PVM daemon.
3. Execute **PFNODE** (or the executable name specified in **pf_com**) to run your IPfortran program.

2.13.4 Additional reserved names for PVM

The PVM version of *pflib* defines two additional COMMON blocks, containing a number of PVM-related variables. Normally, there will be no conflict with user variables, as the IPfortran translator does not insert these COMMON blocks into user program modules. They are provided below for completeness.

pf_pvm INTEGER data used for PVM task management.

pf_pvmc CHARACTER data used for PVM task management.

pf_mytid PVM task ID for this node process.

pf_info PVM status variable.

pf_numt Number of tasks spawned in this virtual machine.

pf_tids Array of PVM task IDs.

pf_speeds Relative speed of hosts in the virtual machine.

pf_hosts Host identifiers.

pf_archs Architecture identifiers.

These data are used when the virtual machine is created and tasks are spawned. Internally, PVM uses "task IDs" to identify the node processes; IPfortran for PVM maps between PVM's task IDs and IPfortran's node ID's by storing the task IDs in the *pf_pvm* COMMON block.

2.13.5 Restrictions on IPfortran for PVM

1. IPfortran for PVM does not generate data-type-dependent communications; all messages are sent as a simple sequence of bytes. As a result, heterogeneous virtual machines are not supported and will likely produce incorrect results due to variations in number representation. At present, you should execute IPfortran PVM programs on homogeneous virtual machines (e.g. workstation clusters or symmetric multiprocessors).

2. As mentioned in the section on KSR-specific limitations, IPfortran intrinsic global sum operations will not work correctly with non power of two processors.

3. There is an internal limit on the maximum number of PVM node programs which may be spawned. This is specified in *pf.h* and *pflibpvm3* as the constant **PVM_MAXTASKS** (currently 64).

2.14 IPfortran Programs and the CM-5

The CM-5 version of IPfortran is layered on Thinking Machines Corporation's CMMD message-passing library [?, ?] and requires *pflibcm5*, supplied with the IPfortran distribution. CMMD supports both "hostless" and "host/node" programming models; IPfortran uses the hostless model.

2.14.1 Compiling and linking programs for the CM-5

The process of translation and compilation for CM-5 IPfortran programs is similar to other supported systems. Supply the switch **-cm5** to the *pfc* translator and use a supported Fortran77 compiler such as *f77* or *af77*. To produce a CM-5 executable, you should use the linker *cmmd-ld*, standard with CM-5 system software.

2.14.2 Sample Makefile for the CM-5

A sample Makefile was presented for the iPSC/860 previously. Here is the equivalent Makefile for the CM-5 (locations and names of object files and libraries may differ on your system):

sample makefile:

```
PFC          = pfc
F77          = f77
LD           = cmmd-ld
PFFLAGS      = -scratch 1000000
FFLAGS       = -O2
LFLAGS       = -comp $(F77)
LIBS         = -lpfcm5
.SUFFIXES: .pf .f .s .o
.f.o:; $(F77) $(FFLAGS) -c $*.f
.pf.f:;$(PFC) $(PFFLAGS) $*.pf
.pf.o:;$(PFC) $(PFFLAGS) $*.pf
$(F77) $(FFLAGS) -c $*.f

planet:$$@.o
       $(LD) $(LFLAGS) -o $@ $@.o $(LIBS)
```

Note that *cmmd-ld* requires a language specification (**-comp**); failure to supply this switch will produce an error at link time.

2.14.3 Additional reserved names for the CM-5

The CM-5 uses separate networks, called the *data* and *control* networks, for point-to-point message-passing and cooperative communications such as reduction operations, scans, and broadcasts. Accordingly, CMMD provides separate library routines for point-to-point and cooperative communication. IPfortran for the CM-5 generates calls to the appropriate routines; this is implemented by additional routines in the CM-5 version of *pflib*. Other utility routines are provided for convenience; they are summarized below:

pf_bsnd Broadcast routine (send).

pf_brcv Broadcast routine (receive).

pf_sync Synchronization of all nodes.

pf_clock Timing function.

2.14.4 CM-5 I/O considerations

Hostless CMMD programs may perform I/O (through an I/O server running either on the partition manager or I/O control processor). Several different I/O modes are supported: *local independent*, *global independent*, *global synchronous broadcast*, and *global synchronous sequential.* CMMD opens *stderr*, *stdin*, and *stdout* (Fortran units 0, 5, and 6, respectively) in global synchronous broadcast mode. By default, subsequent I/O streams are opened in local independent mode. During initialization, IPfortran resets the I/O mode of *stdout* to global independent so that node programs may perform output at any time without global node synchronization.

IPfortran programs may reset the I/O mode by directly calling the appropriate CMMD routines (**CMMD_set_open_mode** and **CMMD_set_io_mode**). If this is done, be sure to include the CMMD Fortran include file (`/usr/include/cm/cmmd_fort.h`).

Be aware that the pool of available file descriptors may be depleted by opening even a small number of files in local independent mode. Please refer to [**?**] for full details of CMMD I/O.

2.14.5 Use of CMMD node timers

IPfortran uses the standard CMMD timing facility, which provides up to 64 independent node timers measuring busy, idle, and elapsed time. These timers are numbered 0 through 63; IPfortran initializes node timer 63 at startup – calls to the routine *pf_clock* return the busy time (in double precision seconds) measured by timer 63 since the program started. Manipulation of this timer by user programs may result in unexpected return values.

3 Internals

The IPfortran translator consists of a lexical analyzer, parser and code generator. The lexical analyzer and parser were obtained from the *f2c* program developed by AT&T Bell Laboratories and Bellcore.

IPfortran source is processed a single line at a time. If the line contains a IPfortran construct, namely: or {, then a syntax tree is constructed. A traversal of the syntax tree produces the native Fortran77. Code without IPfortran constructs, on the other hand, are passed through the code generator and buffered *as is* in an internal buffer with the generated code.

This section under construction.

References

[1] B. Bagheri, T. W. Clark, and L. R. Scott. IPfortran (a parallel extension of Fortran) Reference Manual. Research Report UH/MD–119, Dept. of Mathematics, University of Houston, 1991. Available via anonymous ftp from `kacha.chem.uh.edu` as `pub/pf/pfmanual`.

[2] Babak Bagheri. *Parallel Programming with Guarded Objects.* PhD thesis, Pennsylvania State University, 1994.

[3] Robert J. Harrison. harrison@tcg.anl.gov, 1992.

[4] Henry Burkhardt III. Overview of the KSR1 computer system. Technical Report KSR-TR-9202001, Kendall Square Research, Boston, February 1992.