A++/P++ Manual
(version 0.7.5)

Daniel Quinlan
Lawrence Livermore National Laboratory
L-560
Livermore, CA 94550
925-423-2668 (office)
925-422-6287 (fax)
dquinlan@llnl.gov
Quinlan's Web Page: http://www.llnl.gov/casc/people/dquinlan
A++/P++ Web Page http://www.llnl.gov/casc/Overture/A++P++
A++/P++ Manual (postscript version)
A++/P++ Quick Reference Manual (postscript version)
LACC Number: LA-CC-96-1
LAUR Number: LA-UR-95-3273
UCRL Number: UCRL-MA-136511

August 16, 2000

August 16, 2000

Chapter 0

Copyright

0.1 In Plain English

This software has been released to the public domain, the copywrite notice below applies.

0.2 NOT In Plain English

Preface

Welcome to the A++/P++ array class library. A++ and P++ are both C++ array class libraries, providing the user with array objects to simplify the development of serial and parallel numerical codes. C++ has a collection of primitive types (e.g., int, float, double), A++ and P++ add to this collection the types intArray, floatArray, doubleArray. The use of these new types are as indistinguishable as possible from the use of the compiler's builtin types. Since A++ and P++ faithfully represent elementwise operations on arrays whether in a serial or parallel environment, numerical codes written using these types are thus easier to develop and are portable from serial machines to parallel machines. This greatly simplifies the development of portable code and allows the use of a single code on even very different architectures (using codes originally developed on PC's or workstations). It is hoped that the A++P++ classes provide the user a sufficiently high level to insulate him/her from machine dependencies and yet low enough a level to provide expressiveness for algorithm design[1].

The purpose of this work is to both simplify the development of large numerical codes and to provide architecture-independence through out their lifetimes. By architecture independence we mean an insulation from the details of the particular characteristics of the computer (parallel or serial, vector or scalar, RISC or CISC, etc.). A degree of serial architecture independence already comes from the use of C++, FORTRAN, and other high level languages, but none of these insulate the programmer from details of parallel computer architectures. Message passing libraries provide a common means to support parallel software across several conceptually similar computer architectures, but this does not simplify the complexities of developing parallel software. The A++/P++ array classes are intended to hide the details of the computer architecture including its parallel design (where one exists).

The use of the array objects provided in the A++/P++ class library is much like scalar variables used in FORTRAN, C, or C++. In many respects the array objects are identical to FORTRAN 90 arrays, the principle difference is that these array classes require no specialized parallel compiler (since we use any C++ compiler, all of which I am aware), thus providing a great deal of portability across machines. Specifically, the same code written for a PC or

[1] You be the judge!

SUN workstation, runs on the Cray or CM-5 [2], or the Intel i860, etc.

[2]The CM-5 is a particularly difficult machine to program due to its use of multiple vector units on each node.

Forward

A++ is a serial C++ array class, P++ is the parallel version of the exact same array class interface. A++/P++ can be characterized as a parallel FORTRAN 90 in C++; fundamentally, A++/P++ is simple. A++/P++ is a library and not a compiler and that is its most attractive feature. It is fundamentally simpler than a compiler and builds on top of existing, and well optimized, serial compiler technology. Since it is a library in C++, it works with other compilers (any C++ compiler, we have found) and reserves to those compilers (and future C++ compilers that will to a better job) local code optimizations that are machine dependent. Since A++/P++ is a library it can be used with new features of C++ as they are available without resource consuming retrofit of features into research compilers. Templates, for example, represent a substantial problem to research compilers, but since A++/P++ is just more C++ code it works with any of the new C++ compilers.

A++ has been tested and used at Los Alamos National Laboratory since late 1993, and has proven quite stable since summer 1994. Work on P++ is more recent, continued work will be required for a while still. Separate research work is attempting to address higher efficiency for the array class work, this an other work represents collaborative work with other people at Los Alamos and different universities.

Acknowledgments

I'd like to thank the people in the Numerical Analysis and Parallel Computing Cell of CIC-19 at LANL for their suggestions for improvements and patience while bugs they found were fixed. And I'd like to thank my family for putting up with the whole process.

In particular I would like to thank Bill Henshaw, who has contributed the largest numer of bug reports over the years and has contributed to the current stability of A++/P++. also, Kristi Brislawn, who has both maintained and contributed significant pieces of A++/P++ over the years. Finally, my thanks to Nehal Desai, who as a summer student contributed much of the chapter that now represents the A++/P++ tutorial.

Many students and staff have contributed and continue to contribute to the development of A++/P++.

Contents

List of Figures

Chapter 1

Introduction

This introduction includes a description of what this manual provides, how to use the manual, and the terminology related to the examples that are provided. Included in this introduction is an overview of the A++/P++ array class library. Error messages are contained in the Appendix. Further information is provided about the **A++/P++ Web Site** where more information is available and where the latest copy of the documentation is available. This Web site is presently still in development.

1.1 About This Manual

This manual is divided into seven principle chapters:

- Requirements, Installation and Testing
- A++/P++ Programming Model
- A++: Serial Array Class
- P++: Parallel Array Class
- Tutorial
- Examples
- Reference

These are intended to simplify your use of this manual.

The **Requirements, Installation and Testing** chapter walks the user through the setup of the A++/P++ library. Installation requirements are also explained. A small set of tests are available which verify the installation.

The **A++/P++ Programming Model** chapter explains how to think about the array objects. It provides a conceptual model to help understand how to write code using the array classes.

The **A++: Serial Array Class** chapter describes A++ in more detail and explains what can and can't be done with the array classes. It is intended that this chapter be specific to details and finer points of A++ usage.

The **P++: Parallel Array Class** chapter distribes P++ in more detail and explains what can and can't be done with the parallel array classes. It is intended that this chapter is specific to details and finer points of P++ usage.

The **Tutorial** chapter walks the user through first a simple example and then a more complex example. A collection of simple to advanced programs are contained in the distribution and demonstrate the more sophisticated use of A++/P++ for numerical software development.

The **Examples** chapter provides A++/P++ code fragments that are useful in displaying features of A++/P++ that would otherwise be difficult to explain.

The **Reference** chapter provides detailed information about the use of the A++/P++ array class library. It describes the individual objects that A++/P++ makes available, and each of their global and member functions. It is through the use of the array and other associated objects and the A++/P++ functions that one writes a numerical application. A++/P++ is designed to be intuitive and the use of the array objects is thus similar to that of all other array languages and extensions (e.g., FORTRAN 90, and HPF, MatLab, etc.).

The appendix contains Booch diagrams classifying both the object-oriented design of both A++ and P++ separately. Some knowledge of Booch notation is helpful. Also in the appendix is a list of error messages in A++/P++. These are provided to simplify understanding of internal checking done in A++/P++ and provide detailed explanation of each type of error message that can be reported. They are numbered for convenience; this part of A++/P++ is still in development.

A later version of the manual will include performance data on different machines so that the use of different features in A++/P++ can be better understood. This work is incomplete at present.

1.2 A++/P++ Web Site

We presently have a World Wide Web home page; it can be accessed via
http://www.c3.lanl.gov/∼dquinlan/A++P++.html.

This site is updated regularly with the newest documentation, as it is developed[1].

1.3 Summary

A++/P++ was developed to simplify the development of numerical software. Specifically it allows the expression of a single application developed in the serial environment to be run on sophisticated (and invariably hard to program) parallel machines. It is intended as at least a partial solution to a growing software

[1]All the A++/P++ documentation is presently under development

crisis in the development of large numerical codes as these codes are required to be run on many different architectures (especially complex parallel architectures). But by using A++/P++ the user is insulated from the large differences between high performance computer architectures. At the same time, A++ at its lowest level is optimized to run on specific architectures in ways that are not practical for the users's application to support. With the single source code able to run on large numbers of serial and parallel machines, A++/P++ supports the natural evolution of scientific codes from a serial development environment to a parallel execution environment without constant reimplementation. This provides for both simplified and cheaper software maintenance.

Since A++/P++ is a class library it works with most any C++ compiler including many research oriented C++ compilers (such as special parallel C++ compilers). Thus functionality added by such supersets of C++ are attractive to explore with the A++/P++ array class and this is readily done.

Chapter 2

Portability

This document details the current tested status of A++/P++ on different platforms (architecture and compiler combinations). If your particular combination of architecture/compilers is not listed here, likely A++/P++ will still work, the defaults within the configuration are to use the GNU g++ and GNU gcc compilers (but all this can be specified on the command line of the configure script. "configure -help" provides a more complete listing of the options by which to configure A++ and P++. A separate document details the installation procedure for A++/P++.

This chapter details the current status of portability of A++/P++. We support a broad number of machines, however we have better access to some more than others and this effects the degree to which we can test A++/P++ with different compilers on these platforms. Some platforms (e.g. HP) we don't have access to, but people have contributed the correct options.

2.1 Supported Platforms

A++/P++ uses autoconf for managing the installation of the A++/P++ distribution. Automake is also internally used, as a result all Makefiles follow the GNU standards for makefile options. The A++/P++ configure script is also built by autoconf. All the input file for automake (Makefile.am files) are also included in the distribution.

The options used for all architectures and compilers combinations are gathered together into a single file (A++/config/config.options). Perl scripts read this file and find the correct variable setting for the construction of Makefiles in each directory. The construction of the Makefiles in each directory is organized by autoconf.

2.2 Working and tested for BOTH dynamic and static libraries

This section list the platforms for which both static and dynamic librarieswork properly. These are the most commonly used environments and so they have the most amout of support.

- A++
  - Solaris with (CC and cc)
  - Linux with (g++ and gcc)
  - SGI with (CC and cc compiler)
  - Dec with (CXX and cc compiler)
  - Blue Pacific (cxx & cc compilers)
- P++
  - Dynamic libraries don't work with the MPICH version of MPI
  - Blue Pacific (vendor'snon-mpich MPI)
  - Dec (vendor's non-mpich MPI)

2.3 Working for ONLY static libraries

This section list platforms and compiler versions for which the dynamic libraries DO NOT work properly.

- A++
  - SGI with KCC
  - Solaris with KCC
  - (anything with KCC, I think)
- P++
  - Solaris with CC
  - Solaris with KCC
  - SGI with CC
  - Dec with CXX
  - Blue Pacific (cxx & cc compilers)

2.4 Tested by others but not tested by us

Some users use A++/P++ on a number of other platforms and have worked with us to make sure that we get the configuration options correct for there architecture and compiler combinations.

- A++
  - HP (vendor's compiler)
- P++
  - HP (vendor's compiler)

2.5 Not tested by anyone (lately)

There are several machines where we would like to be able to run but we have not tested A++/P++ in a long time or where the code has never been tested.

- Blue Mountain machine at Los Alamos
- Red machine at Sandia

2.6 Tests Done On Arbitrary Platforms

Autoconf permits the specification of many tests on each platform where the installation of softare takes place. We use a number of standard tests in autoconf but more importantly we add many which are specific to the installation of C++ applications and additional tests which are specific to the use of applications on parallel machines.

Each test is available as a macro and is distributed with A++/P++. These tests include:

- Tests performed specific to serial platforms:
  - Tests on the target C and C++ compilers (see below)
- Tests performed specific to C++:
  - Test for use of `bool` in target C++ compiler
  - Test for support of **explicit template instanciation** in the target C++ compiler (this is a test borrowed from the SAMRAI project).
  - Test of support for dynamic libraies
- Tests performed specific to parallel platforms:
  - Search for MPI location (a standard CASC autoconf macro)
  - Tests specific to MPI

∗ Search for MPI libraries, include directory, and mpirun

∗ Test for **mpicc** and **mpiCC** to be used in place of normal C++ and C compilers

∗ Test compile and run of example MPI code on target platform using target compiler options using different numbers of processors. Current test test example MPI application over 1-6 processors. Testing over more processors would complicate the installation on arbitrary machines (like single CPU workstations running MPI).

∗ Test for requirement of mpirun with `-machinefile <filename>` option.

Chapter 3

Requirements, Installation and Testing

This chapter contains the software and hardware requirements of the A++/P++ array class library. Additionally, it details the installation of the software. Included are directions for how to modify your environment to use PVM and where to get the PVM software. Since A++/P++ can optionally use a graphics library for visualization of the A++/P++ array data, info is included about where the **Plotmtv** software is available and how to use it with A++/P++.

3.1 Requirements and Options

3.1.1 What Hardware you require

As best I know any computer will do, a PC under MS DOS (with 640K RAM) will likely run out of memory in the use of the array class for any meaningful problem. However a large PC should work fine. A++ has been used on Sun workstations, Cray supercomputers (X-MP, Y-MP, C-90), IBM RS-6000 workstations, SGIs, ...

3.1.2 What Software you require

You will require a C++ compiler, there is no way around it. Additionally you will require either a C or FORTRAN compiler[1].

Use of Optional Hardware and Software:

If you have a parallel computer you can use P++ (otherwise A++ and P++ are equivalent (except for some additional overhead in P++, because it will recognize that you are not using more than a single processor, but it will store

[1] Currently the Machine Dependent Interface for A++/P++ is only available in C, so you require a C compiler.

some additional information that A++ will not)[2]). If you want to use P++ you will require some communication library. Presently P++ works with PVM and MPI. The PVM home page is
http://www.epm.ornl.gov/pvm/pvm_home.html, info on where to find pvm is available there. MPI is available via anonymous ftp from info.mcs.anl.gov and it is located in the directory pub/mpi the MPI Web page is at
http://WWW.ERC.MsState.Edu:80/mpi/, MPI is available for a large and growing number of parallel machines and network environments, and thus allows P++ to run on any of these environment where MPI is supported [3].

Where to get the A++/P++ array class library software:

The software is available from Los Alamos National Laboratory, contact: dquinlan (dquinlan@llnl.gov). A++/P++ is currently made availabel as part of teh Overture Framework. Overture's Home Page is
http://www.llnl.gov/casc/Overture.

3.2 Where to Get A++/P++

The installation of A++ is simple once you have the A++P++-0.X.X.tar.gz file. This is available from the Overture web pages at
http://www.llnl.gov/casc/Overture.

3.3 How to Install A++/P++

A++ and P++ are distributed together; they are meant to be installed together as well. Assuming you have the A++P++-0.X.X.tar.gz file use gunzip and then untar the file to build the A++ directory.

The configure script requires no additional parameters, it will figure out what sort of machine you have and run numerous tests to see what options should be used internally in the installation (this is the standard autoconf mechanism). We have added many additional test (testing for MPI, dynamic library capabilities, etc.).

A relatively new feature of A++/P++ is the use of autoconf which builds a configure script which will setup A++ and P++ plus generate the makefile for the final build. There are is a single configure script in each directory of the directory hierarchy (the A++ and P++ configure scripts can be run separately for example).

To use the configure script (there is no other way) type `configure -help` to see the options. The output should appear something like this for the configure script at the top level directory:

```
configure -help
```

[2]This is a trivial point since both A++ and P++ have the same interface.

[3]MPI was announced at Super Computing 93, PVM has been around much longer. MPI has received considerable vendor support, thus P++ will be restricted to PVM and MPI, but this is sufficient for general use

```
Usage: configure [options] [host]
Options: [defaults in brackets after descriptions]
Configuration:
  --cache-file=FILE       cache test results in FILE
  --help                  print this message
  --no-create             do not create output files
  --quiet, --silent       do not print ‘checking...’ messages
  --version               print the version of autoconf that created configure
Directory and file names:
  --prefix=PREFIX         install architecture-independent files in PREFIX
                          [/usr/local]
  --exec-prefix=EPREFIX   install architecture-dependent files in EPREFIX
                          [same as prefix]
  --bindir=DIR            user executables in DIR [EPREFIX/bin]
  --sbindir=DIR           system admin executables in DIR [EPREFIX/sbin]
  --libexecdir=DIR        program executables in DIR [EPREFIX/libexec]
  --datadir=DIR           read-only architecture-independent data in DIR
                          [PREFIX/share]
  --sysconfdir=DIR        read-only single-machine data in DIR [PREFIX/etc]
  --sharedstatedir=DIR    modifiable architecture-independent data in DIR
                          [PREFIX/com]
  --localstatedir=DIR     modifiable single-machine data in DIR [PREFIX/var]
  --libdir=DIR            object code libraries in DIR [EPREFIX/lib]
  --includedir=DIR        C header files in DIR [PREFIX/include]
  --oldincludedir=DIR     C header files for non-gcc in DIR [/usr/include]
  --infodir=DIR           info documentation in DIR [PREFIX/info]
  --mandir=DIR            man documentation in DIR [PREFIX/man]
  --srcdir=DIR            find the sources in DIR [configure dir or ..]
  --program-prefix=PREFIX prepend PREFIX to installed program names
  --program-suffix=SUFFIX append SUFFIX to installed program names
  --program-transform-name=PROGRAM
                          run sed PROGRAM on installed program names
Host type:
  --build=BUILD           configure for building on BUILD [BUILD=HOST]
  --host=HOST             configure for HOST [guessed]
  --target=TARGET         configure for TARGET [TARGET=HOST]
Features and packages:
  --disable-FEATURE       do not include FEATURE (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
  --with-PACKAGE[=ARG]    use PACKAGE [ARG=yes]
  --without-PACKAGE       do not use PACKAGE (same as --with-PACKAGE=no)
  --x-includes=DIR        X include files are in DIR
  --x-libraries=DIR       X library files are in DIR
--enable and --with options recognized:
  --enable-PXX         also configure P++
```

And for the A++ directory, the output of configure -help is:

```
configure -help
Usage: configure [options] [host]
Options: [defaults in brackets after descriptions]
Configuration:
  --cache-file=FILE       cache test results in FILE
  --help                  print this message
  --no-create             do not create output files
  --quiet, --silent       do not print ‘checking...’ messages
  --version               print the version of autoconf that created configure
Directory and file names:
  --prefix=PREFIX         install architecture-independent files in PREFIX
                          [@APP_DEFAULT_PREFIX@]
  --exec-prefix=EPREFIX   install architecture-dependent files in EPREFIX
                          [same as prefix]
  --bindir=DIR            user executables in DIR [EPREFIX/bin]
  --sbindir=DIR           system admin executables in DIR [EPREFIX/sbin]
  --libexecdir=DIR        program executables in DIR [EPREFIX/libexec]
  --datadir=DIR           read-only architecture-independent data in DIR
                          [PREFIX/share]
  --sysconfdir=DIR        read-only single-machine data in DIR [PREFIX/etc]
  --sharedstatedir=DIR    modifiable architecture-independent data in DIR
                          [PREFIX/com]
  --localstatedir=DIR     modifiable single-machine data in DIR [PREFIX/var]
  --libdir=DIR            object code libraries in DIR [EPREFIX/lib]
  --includedir=DIR        C header files in DIR [PREFIX/include]
  --oldincludedir=DIR     C header files for non-gcc in DIR [/usr/include]
  --infodir=DIR           info documentation in DIR [PREFIX/info]
  --mandir=DIR            man documentation in DIR [PREFIX/man]
  --srcdir=DIR            find the sources in DIR [configure dir or ..]
  --program-prefix=PREFIX prepend PREFIX to installed program names
  --program-suffix=SUFFIX append SUFFIX to installed program names
  --program-transform-name=PROGRAM
                          run sed PROGRAM on installed program names
Host type:
  --build=BUILD           configure for building on BUILD [BUILD=HOST]
  --host=HOST             configure for HOST [guessed]
  --target=TARGET         configure for TARGET [TARGET=HOST]
Features and packages:
  --disable-FEATURE       do not include FEATURE (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
  --with-PACKAGE[=ARG]    use PACKAGE [ARG=yes]
  --without-PACKAGE       do not use PACKAGE (same as --with-PACKAGE=no)
  --x-includes=DIR        X include files are in DIR
  --x-libraries=DIR       X library files are in DIR
```

```
--enable and --with options recognized:
  --with-CC=ARG           manually set C compiler to ARG
  --with-M4=ARG           manually set M4 to ARG
  --with-CXX=ARG          manually set C++ compiler to ARG
  --enable-CXXOPT=ARG   manually set CXXOPT to ARG
  --enable-COPT=ARG   manually set COPT (optimization flags) to ARG
  --enable-CXXDEBUG=ARG   manually set CXXDEBUG to ARG
  --enable-CDEBUG=ARG     manually set CDEBUG to ARG
  --enable-CXXOPTIONS=ARG   manually set CXXOPTIONS to ARG
  --enable-COPTIONS=ARG     manually set CDEBUG to ARG
  --enable-CXX_WARNINGS=ARG   manually set CXX_WARNINGS to ARG
  --enable-C_WARNINGS=ARG   manually set C_WARNINGS to ARG
  --with-CXX_TEMPLATES=ARG
                              manually set CXX_TEMPLATES to ARG
  --with-ARCH_LIBS=ARG    manually set ARCH_LIBS to ARG
  --enable-INTERNALDEBUG   turn on internal debugging for any ARG
  --with-USE_TAU_PERFORMANCE_MONITOR=ARG  manually set USE_TAU_PERFORMANCE_MONITOR to YES or NO A
  --enable-SHARED_LIBS, manually enable building of shared libraries, off by default
  --enable-STATIC_LINKER=ARG manually set linker for linking static libraries to ARG
  --enable-STATIC_LINKER_FLAGS =ARG manually set static linker flags to ARG
  --enable-SHARED_LIB_EXTENSION=ARG manually set file extension for shared libraries to ARG (e.g.
  --enable-C_DYNAMIC_LINKER=ARG manually set linker for linking shared library from C object file
  --enable-CXX_DYNAMIC_LINKER=ARG manually set linker for linking shared library from C++ object
  --enable-C_DL_COMPILE_FLAGS=ARG   manually set C compiler flags to make objects suitable for bu
  --enable-CXX_DL_COMPILE_FLAGS=ARG   manually set C++ compiler flags for creating object files s
  --enable-C_DL_LINK_FLAGS=ARG   manually set flags for linking C object files into a shared libr
  --enable-CXX_DL_LINK_FLAGS=ARG   manually set linker flags for linking C++ object files into a
  --enable-RUNTIME_LOADER_FLAGS=ARG   manually set runtime loader flags to ARG
```

For the P++ directory, the output of configure -help is:

```
configure -help
Usage: configure [options] [host]
Options: [defaults in brackets after descriptions]
Configuration:
  --cache-file=FILE       cache test results in FILE
  --help                  print this message
  --no-create             do not create output files
  --quiet, --silent       do not print ‘checking...’ messages
  --version               print the version of autoconf that created configure
Directory and file names:
  --prefix=PREFIX         install architecture-independent files in PREFIX
                          [@PPP_DEFAULT_PREFIX@]
  --exec-prefix=EPREFIX   install architecture-dependent files in EPREFIX
                          [same as prefix]
  --bindir=DIR            user executables in DIR [EPREFIX/bin]
  --sbindir=DIR           system admin executables in DIR [EPREFIX/sbin]
  --libexecdir=DIR       program executables in DIR [EPREFIX/libexec]
  --datadir=DIR          read-only architecture-independent data in DIR
                         [PREFIX/share]
  --sysconfdir=DIR       read-only single-machine data in DIR [PREFIX/etc]
  --sharedstatedir=DIR   modifiable architecture-independent data in DIR
                         [PREFIX/com]
  --localstatedir=DIR    modifiable single-machine data in DIR [PREFIX/var]
  --libdir=DIR           object code libraries in DIR [EPREFIX/lib]
  --includedir=DIR       C header files in DIR [PREFIX/include]
  --oldincludedir=DIR    C header files for non-gcc in DIR [/usr/include]
  --infodir=DIR          info documentation in DIR [PREFIX/info]
  --mandir=DIR           man documentation in DIR [PREFIX/man]
  --srcdir=DIR           find the sources in DIR [configure dir or ..]
  --program-prefix=PREFIX prepend PREFIX to installed program names
  --program-suffix=SUFFIX append SUFFIX to installed program names
  --program-transform-name=PROGRAM
                         run sed PROGRAM on installed program names
Host type:
  --build=BUILD          configure for building on BUILD [BUILD=HOST]
  --host=HOST            configure for HOST [guessed]
  --target=TARGET        configure for TARGET [TARGET=HOST]
Features and packages:
  --disable-FEATURE      do not include FEATURE (same as --enable-FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
  --with-PACKAGE[=ARG]   use PACKAGE [ARG=yes]
  --without-PACKAGE      do not use PACKAGE (same as --with-PACKAGE=no)
  --x-includes=DIR       X include files are in DIR
  --x-libraries=DIR      X library files are in DIR
--enable and --with options recognized:
  --with-CC=ARG          manually set C compiler to ARG
  --with-M4=ARG          manually set M4 to ARG
  --with-CXX=ARG         manually set C++ compiler to ARG
  --enable-CXXOPT=ARG   manually set CXXOPT to ARG
  --enable-COPT=ARG   manually set COPT (optimization flags) to ARG
  --enable-CXXDEBUG=ARG  manually set CXXDEBUG to ARG
  --enable-CDEBUG=ARG    manually set CDEBUG to ARG
  --enable-CXXOPTIONS=ARG  manually set CXXOPTIONS to ARG
  --enable-COPTIONS=ARG    manually set CDEBUG to ARG
  --enable-CXX_WARNINGS=ARG  manually set CXX_WARNINGS to ARG
  --enable-C_WARNINGS=ARG  manually set C_WARNINGS to ARG
  --with-CXX_TEMPLATES=ARG
                           manually set CXX_TEMPLATES to ARG
  --with-ARCH_LIBS=ARG   manually set ARCH_LIBS to ARG
  --enable-INTERNALDEBUG  turn on internal debugging for any ARG
  --with-USE_TAU_PERFORMANCE_MONITOR=ARG  manually set USE_TAU_PERFORMANCE_MONITOR to
  --enable-SHARED_LIBS, manually enable building of shared libraries, off by default
  --enable-STATIC_LINKER=ARG manually set linker for linking static libraries to ARG
  --enable-STATIC_LINKER_FLAGS =ARG manually set static linker flags to ARG
  --enable-SHARED_LIB_EXTENSION=ARG manually set file extension for shared libraries to ARG (e.g.
  --enable-C_DYNAMIC_LINKER=ARG manually set linker for linking shared library from C object file
  --enable-CXX_DYNAMIC_LINKER=ARG manually set linker for linking shared library from C++ object
  --enable-C_DL_COMPILE_FLAGS=ARG   manually set C compiler flags to make objects suitable for bu
  --enable-CXX_DL_COMPILE_FLAGS=ARG   manually set C++ compiler flags for creating object files s
  --enable-C_DL_LINK_FLAGS=ARG   manually set flags for linking C object files into a shared libr
  --enable-CXX_DL_LINK_FLAGS=ARG   manually set linker flags for linking C++ object files into a
  --enable-RUNTIME_LOADER_FLAGS=ARG   manually set runtime loader flags to ARG
  --disable-MPI           Do not set up MPI flags
  --with-mpi-include=DIR  mpi.h is in DIR
  --with-mpi-libs=LIBS    LIBS is space-separated list of library names
                          needed for MPI, e.g. "nsl socket mpi"
  --with-mpi-lib-dirs=DIRS
                          DIRS is space-separated list of directories
                          containing the libraries specified by
                          `--with-mpi-libs', e.g "/usr/lib /usr/local/mpi/lib"
  --with-mpi-flags=FLAGS  FLAGS is space-separated list of whatever flags other
                          than -l and -L are needed to link with mpi libraries
  --with-MPICC=ARG        ARG is mpicc or similar MPI C compiling tool
  --with-mpi-include=DIR  mpi.h is in DIR
  --with-mpi-libs=LIBS    LIBS is space-separated list of library names
                          needed for MPI, e.g. "nsl socket mpi"
  --with-mpi-lib-dirs=DIRS
                          DIRS is space-separated list of directories
                          containing the libraries specified by
                          `--with-mpi-libs', e.g "/usr/lib /usr/local/mpi/lib"
  --with-mpi-flags=FLAGS  FLAGS is space-separated list of whatever flags other
                          than -l and -L are needed to link with mpi libraries
  --with-MPICC=ARG        ARG is mpicc or similar MPI C compiling tool
  --with-mpirun=ARG   ARG is mpirun or equivalent
  --with-mpi-machinefile=FNAME    FNAME lists machines to run mpi progs on
  --with-STL_INCLUDE=ARG    manually set STL_INCLUDE to ARG
  --without-PADRE           Avoid using PADRE Library within P++
  --with-STL_INCLUDE=ARG    manually set STL_INCLUDE to ARG
  --with-GLOBAL_ARRAYS     Use GLOBAL ARRAYS Library (from PNL) within PADRE
```

Note that numerous option araavailable, though non should be reuired for a default installation of A++/P++.

An example configure line to install A++ might be:

```
configure
```

In another example to configure for a specific C++ and C compiler the command line would be:

```
configure --with-CC=cc --with-CXX=CC
```

In still another example to configure for without optimization to improve the compile speed of the A++/P++ library: the command line would be:

```
configure --enable-CXXOPT= --enable-COPT=
```

In still an other example to compile P++ (also turning on INTERNALDEBUG option) in addition to A++:

```
configure --enable-INTERNALDEBUG --enable-PXX
```

On some machines when compiling P++ the location of MPI must be specified. Example specification of MPI location:

```
configure --enable-PXX --with-CC=cc --with-CXX=cxx --enable-SHARED_LIBS \
          --with-mpi-include=/usr/opt/MPI170/include \
          --with-mpi-lib-dirs=/usr/opt/MPI170/lib \
          --with-mpi-libs=mpi \
          --with-mpirun=/usr/opt/MPI170/bin/dmpirun
```

There are clearly numerous options available to specify numerous details of the compilation.

3.3.1 Most common options to `configure`

The most common options to specify for building only A++ are just (no options)

```
configure
```

and for P++ are just (specify compilation of P++):

```
configure --enable-PXX
```

The configure script will handle the identification of the machine and other details automatically or will output what options need to be specified with additional data.

3.3.2 Importance of compiling A++/P++ with the INTERNALDEBUG option

For A++ this is not an important option though using it will provide internal error checking and will likely help catch mistakes you make before they cause mysterious problems which are difficult to explain. For the most part A++ is sufficiently mature that if you just turn on the bounds checking **??** it will catch most user errors.

For P++ we suggest the use of the INTERNALDEBUG option when it is compiled because this will catch internal errors and be more useful to use if you report a bug. But with the INTERNALDEBUG option P++ will run noticeably slower. So you might choose to have two versions. This reflects the fact that we are still fixing bugs within P++ since it is still being tested at Livermore.

Example fragement of output from `configure --help`:

```
  --enable-INTERNALDEBUG   turn on internal debugging for any ARG
```

Example command line for configure showing the specification of the INTRNALDEBUG option

```
configure --enable-INTERNALDEBUG
```

3.3.3 Parallel Communication Libraries

P++ supports the use of either PVM or MPI, but is currently developed using MPI (it used to be the other way around). So we now suggest the use of MPI with P++ for simplicity and because in the future PVM support will be limited (because the PADRE distribution library uses several publically available libraries internally and few of these can support PVM). The following sub-sections discribe the installation of P++ with PVM and MPI respectively. **Note for users at LLNL: MPI is already installed and you should use it.**

How to Install MPI

Get MPI (the ftp site is listed above) and install MPI following the instructions enclosed with MPI, nothing special is required. Then within the installation of A++/P++, it is only required that the use have the MPI/bin directory in their path (so that mpirun can be found). The tests within autoconf (our specialized version of macros for autoconf) will test for MPI, if it is not found the user may have to specifiy the location of the direcctory explicitly using the command line options.

How to Install PVM

Get PVM from the web and install it using the instructions that come with PVM. Nothing about this step is in any way specific to using P++.

3.4 Testing the A++/P++ Installation

Run `make check` from any directory of the A++/P++ directory tree and all test will be run for that subtree. Running `make check` from the top level direcotry (A++P++) will run all tests (for A++ and P++).

A small set of test programs is available in the A++/EXAMPLES and P++/EXAMPLES directories and these can be used to test A++ and P++. The output from both A++ and P++ should be nearly the same. The EXAMPLES directory contains a makefile which can be used to make the example applications. The A++ test program is called "testcode.C" and is located in the EXAMPLES subdirectory. The test code will work properly on a single processor using P++, but not in parallel since it uses indirect addressing which in not a part of P++ yet. There is a separate test code for P++, called "testppp.C".

It does run in parallel and it is a common test that we have for each new release of the A++/P++ implementation.

Additional test will be placed into the A++/TESTS and P++/TESTS directories over time. These are mostly previously fixed bugs over the years which we would like to avoid reintroducing in the future (hence we provide a simple mechanism to test A++ and P++ against previously reported bugs that have been fixed).

These tests (in A++/EXAMPLES, P++/EXAMPLES, A++/TESTS, and P++/TESTS) are automatically run by running “make check” in from the top level A++P++ directory. This is one or our best mechanisms for testing A++/P++.

Chapter 4

Programming Model

Most all software assumes some programming model that will provide the user with sufficient intuition to use the software in a way reasonably consistent with its design. The A++/P++ programming model provides an underlying framework for the design of software using the library A++/P++ array class library. It is intended to be simple since it is an array language at its core A++ is very similar to FORTRAN 90, and P++ is similar to HPF (though without special comment like directives). This chapter will first describe the programming model for A++, the serial array class, and then proceed to define the programming model for P++ which represents extensions of the serial programming model to provide the specification of array object distributions onto multiple processors.

4.1 A++ Programming Model

A++ is simple, the programming model of A++ is focused on arrays as objects (data), and array operations (functions that operate on array objects). We don't assume that all computations can be expressed using such array objects and many are clearly not suited (Gaussian Elimination for example), however a very large set of scientific computations is well suited to expression via array syntax (local relaxation methods) and this portion is what we address and target with the A++/P++ array classes. In addition to A++ all of C++ is available as well as any other libraries written in C++[1].

By providing a programming model centered around an array we don't exclude the interaction with other programming models in the same application. For example, a matrix class library could represent a matrix model for the solution of linear systems and obtain the problem from a part of the application

[1]Since C++ interfaces to FORTRAN (and nearly every other language as well), all other libraries are available to the user. This is the advantage of working within the C++ language to define libraries like A++/P++ rather than resorting to specialized languages with limited portability.

that used the array objects (the array programming model). Such simple interactions between class libraries are intentional and hopefully more complex interactions will result from more extensive use of A++/P++.

The A++/P++ array class library is intended as a foundation class in the sense that it can be used to build more sophisticated user defined types which are application specific. A++/P++ does not however attempt to address the distribution of other more complex objects like trees, lists, etc.[2]

4.2 P++ Programming Model

A Parallel Computer consists (for our simple model) of:

- Processors (many of them) each with its own local memory
- Interconnection network defining now the processors are connected.

The P++ programming model is identical to that of the one for A++, but extended to define the partitioning of the array objects across the local memories of a multiprocessor computer. This is the principle reason why the P++ library can be substituted at compile time for the A++ class library allowing the reuse of the serial course code in the parallel environment.

In the manipulation of array objects P++ abstracts the parallel machine but provides the user with control over the layout of the array objects into the separate memories of each processor. The layout management has its own programming model (this layout model is similar to HPF is many ways but contains additional functionality which is well suited to the manipulation of large numbers of arrays in a parallel environment (instead of just a single grid (or a small number)).

4.3 The programming model of P++

P++ uses the SPMD programming model, this is important since without the SPMD programming model the simplified representation of the parallel program from the serial program would not be practical. Specifically, P++ is an SPMD implementation of a Data Parallel programming model, though not limited exclusively to the data parallel programming model. The data parallel model is implemented using two communication models internally. These allow for efficient communication between aligned array operations and permit unaligned array operations as well. The user never sees either of these two execution models since they are abstracted. What is seen is that array operations between aligned array objects is more efficient than those between unaligned array objects (this should be no surprise since unaligned array operations require more communication, hense they are avoided within most of parallel programming.

[2]Today the are libraries that formally derive from the A++/P++ array objects to add additional functionality specific to grid geometry etc.

Thus P++ combines the serial programming model with a virtual shared grid model where the operations on array objects are executed regardless of their decomposition in the multiprocessor environment. The combined effect of these serial and parallel programming models being identical is the principle means by which P++ allows serially developed codes to be run on distributed memory machines. The efficiency of the execution of the serial code in the parallel environment is determined by the alignment of the data within the array operations.

4.3.1 Single Program Multiple Data stream (SPMD)

In contrast to the explicit host/node programming model which requires both a host and one or more node programs, the SPMD programming model consists of executing one single program source on all nodes of the parallel system.

The implementation of the SPMD model requires that commonly available functionality in the serial environment be provided in the parallel environment in such a way that the serial source code can be used on the distributed memory machine. One of the most important functionalities that is provided in the parallel programming model to support basic functionality of the serial code is a parallel I/O system. This can then be used in place of the serial I/O system, to support the required functionality of the parallel SPMD programming environment.

Currently, only basic functionality of the SPMD programming model (I/O system: printf, scanf; initialization and termination of processes) is available. Implementation details are abstracted from the user. The SPMD programming model replicates the functionality of the traditional parallel host/node programming model. For example, the standard function scanf for reading from standard input is implemented in such a way that an arbitrarily chosen master node reads the data from standard input and distributes it to all other nodes (slave nodes). This master/slave relationship is only present within the Parallel I/O System and not used elsewhere in P++.

4.3.2 Virtual Shared Grids (VSG)

The concept of Virtual Shared Grids gives the appearance of Virtual Shared Memory restricted to array variables. Computations are based on global indexing. Communication patterns are derived at runtime (though communication schedules are cached for improved performance), and the appropriate messages are automatically generated by P++. In contrast to traditional Virtual Shared Memory, where the operating system does the communication without having information about the algorithm's data and structure, the array syntax of P++ provides the means for the user to express details of the algorithm and data structure to the compiler and runtime system. This guarantees that the number of communications and the amount of communicated data is minimized. Through the restriction to structured grids, the same kind and amount of communication, as with the explicit Message Passing model is sent/received and therefore also approximately the same efficiency is achieved. This is a tremendous advantage over the more general (traditional) Virtual Shared Memory model.

There are two basic communication models that are currently implemented in P++. How they can interact, is described in more detail in the examples:

- **VSG Update:**

  In the implementation of the general Virtual Shared Grids concept, within the VSG Update communication model, we Restrict the classical Owner Computes rule, that might be applied to whole expressions, to binary subexpressions and define the Owner arbitrarily to be the left operand. This simple rule handles the communication required in the parallel environment; specifically, the designated owner of the left operand receives all parts of the distributed array necessary to perform the given binary operation. Thus the temporary result and the left operand are partitioned similarly (see Figure 3.1).

– P++ user level:

```
A = B + C
```

– P++ internal execution:

```
1.  T = B + C
    ---------
    P1: T11 = B11 + C1
        receive C21 from P2
        T12 = B12 + C21
    P2: T2 = B2 + C22
        send C21 to P1
    P3: idle

2.  A = T
    -----
    P1: send T1 to P3
    P2: send T2 to P3
    P3: receive T1 from P1
        receive T2 from P2
        A = T
```

Figure 4.1: An example for VSG Update based on the Owner computes rule: A = B + C on 3 processors

Within Figure 3.1 we have three P++ array objects (**A**,**B**, and **C**), each is distributed differently. The first operation is to form a temporary from **B** and **C**. Thus the temporary, **T**, is the result of the operation **B** + **C**. By our VSG rule, **T** is given the same distribution as the left operand, **B**.

So **T** is build with the same distribution as **B** (same size as **B**, as well) and the messages are generated to get the parts of **C** that are required for the operator+ operation with **B**. After **T** is formed the operator= operation is done to fill in the array, **A**, with the intermediate result from **T**. Each operand has a different distribution (since **A** and **B** had different distributions and **T** matches the distribution of **B**). The message passing in generate to get the data relavant for each processors portion of **T** which is required by the processors owning **A**.

- **Overlap Update:** Within the VSG Update communication is introduced (if required because of the indexing of the operands) within each binary operation. This is not always efficient if the arrays are aligned (even if it is the only way to make unaligned array operations work). A well developed technique for handling such issues is to introduce ghost boundaries of overlap between the edges of the partitioned data. Such ghost boundaries are typically meant to be "read only" copies of the neiboring processors data.

  This model is more efficient since within stencile operaations (if the ghost boundary width permits) the replicated data within the ghost bounaries can be read and message passing to get such data is avoided. Upon the completion of the array statement the ghost boundaries can be recopied to force them to be a consistant representation of the nieboring processors data[3]. The point of this alternate communication model is that for aligned array operations (not counting the indexing which would unalign the references to the data) message passing can be done once within an array statement instead of at each binary operation, this is much more efficient (but only is the array objects are aligned, otherwise this technique can not work). Figure 3.2 shows the distribution of a P++ array with ghost boundaries of non zero width.

Figure 4.2: The standard method of grid partitioning with overlap

[3]More detailed mechanisms can be used to represent valid and invalid ghost boundaries and so the update of the ghost boundaries can be scheduled more loosely.

P++ arrays (Virtual Shared Grids) are constructed in a distributed fashion across the processors of the parallel system. Partitioning information and processor mapping is stored internally. A low level library, MultiBlock Parti[4] is used to hold information about the distributions and to handle the update of ghost boundaries and more irregular data transfers as required by the VSG updates. MultiBlock Parti has been of great help in simplifying the desing of P++ and speeding its development.

All information, required for evaluating expressions using either the VSG or Overlap update models, is expressed through the A++/P++ array syntax. Additional information required is obtained from the distribution of the array objects which is stored internally within each array. This information is used to generate message passing through either of the two communication models depending on if the ghost boundaries are sufficiently large to use the more efficient Overlap update model.

[4]Available from University of Maryland and the result of research by Al Susmman and Joel Saltz.

Chapter 5

A++: Serial Array Class Library

This section is not intended to be a reference section (there is already one of those) but is intended to detail how A++ is meant to be used (and discuss how to abuse it too).

A++ provides an array language implemented in C++ as a class library. It provides array syntax for the expression of numerical algorithms, this syntax includes indexing using **Index** objects (triplets representing base, bound, and stride). Beyond this there are many details to explain and clarify.

5.1 Views of A++ arrays

A++ includes overloaded parenthesis "()" operators[1] which allow for the creation of a view into an existing array object. The value returned from the parenthesis operator is another array object, this array object is a *view*. Any modification of the *view* is reflected in the object of which it is a view.

For example,

```
doubleArray A(5,5,5); // A's range is from 0..4 along each axis
A = 1;
Range I(1,3), J(1,3), K(1,3);
A (I,J,K) = 3;        // Sets view of interior of A to 3
```

5.1.1 Indexing

Vector Indexing

A++ provides for the indexing of a region of an array object, as in the previous example using the **Range** objects **I**, **J**, **K**. Here the Range object is used to

[1] using the ::operator()() member function of the array objects.

represent a base bound pair of values over which the array operation are to take place. In addition to the **Range** objects a slightly more flexible object is provided; the **Index** object. The **Index** object stores the base, length of access, and stride. The **Range** and **Index** objects have many different constructors.

Scalar Indexing

A++ also provides scalar indexing, that is indexing using only integer values and returning a reference to a scalar. This scalar indexing is implemented using the same parenthesis operator, but overloaded (in the C++ sense) to take integer values.

For example,

```
doubleArray A(5,5,5);
A = 1;
A(2,3,4) = 5;  // sets a single element in A to the value 5
```

Bases of Arrays

All A++ array objects have a base, bound and stride for each axis of their multidimensional representation. The (Bound-Base)+1 is the length along each axis (assuming stride is 1). The Base by default is the value of the "global default base" which is, by default, initialized to be ZERO. The base can be changed dynamically though the **setBase** member function. If the array is built using **Range** objects, as in:

```
Range I (-10,20);
Range J (10,20);
Range K (1000,2000);
doubleArray A (I,J,K);
```

then the bases of the array are -10, 10, and 1000; for each axis. The lengths along each axis would then be 30,10, and 1000.

Valid indexing of the array objects requires that knowledge of the index space represented by the array object. Using the previous array as an example,

```
Range I_1 (21,25);
Range J (10,20);
Range K (1000,2000);
A (I_1,J,K) = 0;   // error out of bounds access
Range I_2 (0,10);
A (I_2,J,K) = 0;   // correct usage
```

The base, bound and stride can be obtained from the array objects by using the access member function: **getBase**, **getBound**, and **getStride**. Very general functions that work on array objects should not assume the base of the array object is always zero or that the stride is always 1. Though an assumption of the stride being 1 is generally acceptable since the strided view of an array object is 1. However, if your application calls FORTRAN (or any other language), then the strides issue will be important and you should check the stride to accommodate non-unit strides (unit stride implies stride = 1). The pointer to the internal array data, returned by the **getDataPointer** member function, points to the first valid array element and is not offset for any nonzero base.

Thus base of arrays are defined at construction of the array objects, they can be nonzero, and they can be changed dynamically. Note that if a function taking an array object as input changes the base of the array the array object will have the new base as a side effect of the function call.

Bases of Views

The views returned from the indexing of A++ array objects using **Range** and **Index** objects are ordinary A++ objects, no different from the A++ object of which they are a view. Except, that they are marked internally as being a view. The **isAView** member function returns a boolean value to determine if an array object is a view. Other details are important for views as well:

- The base of a view can be different than that of the array of which it is a view. Specifically it is the Base **Index** or **Range** object used to build the view.
- The stride of a view may be not unit stride (Unit Stride == 1).
- The pointer to the raw data for a view might not be what you expect. The pointer to the data returned by the **getDataPointer** member function is a pointer to the first valid element of the original array. The view is a subset of that; determined by the base, bound and stride of the view (minus the base of the original array object). So this must be understood when handing the pointer to data represented by a view of an array object to a FORTRAN function.

5.2 Reference Counting

Reference counting is the storage of a value that represents the number of external references to an object. The purpose is to allow many external references to an object and also permit the object to be cleared from memory when there are no remaining references. For example, the internal array data within the A++ array objects is reference counted. A view of the data is an A++ array which has an external reference to the data of another object (the original array object from which the view was taken).

5.2.1 Internal A++ Reference Counting

This subsection forms an example to explain what reference counting is since internal reference counting of the data within A++ is completely hidden within

A++. So the use need not worry about the manipulation reference counts for the raw data used by the array objects.

```
floatArray A (10);
Range I (0,4);
A(I) = 5;     // A(I) is a view of A with a reference to A's data
```

In this example A(I) is a view of the array data in A, but it is a valid floatArray object. It has an external reference to the same data as in A (A's data). If A were deleted A's data would not be released until the view, A(I), went out of scope (the compiler controls the calls to the destructors since views are local objects (sometimes called compiler temporaries)). This is the way the reference counting works on the array data used internally within the array objects, the user never sees this level of reference counting.

5.2.2 External A++ Reference Counting

If applications use A++ objects, and specifically pointers to them, so as to generate multiple references then A++ has member functions to help manipulate and handle these multiple references. This is the A++ reference counting that this section is really about. The point is that reference counting is more easily done if it is keep with the array objects directly. A++ member data allow this and A++ member functions allow access and manipulation of that member data. The use of this reference counting only appears in special uses of A++ within applications and more commonly within other libraries.

A++ arrays contain reference counting that may be manipulated by the user to allow many references to a single A++ array object. This is required if you wish to build code that uses A++ array object through pointers to those A++ arrays, and support multiple references to the A++ array objects. Just using pointers to A++ array objects is not sufficient to require the use of the A++ reference counting. In general you have to be building separate objects each of which wants to point (via a pointer) to a single A++ array. This method of building code is typical of C style programming, but is largely unnecessary when using A++ array objects since separate array objects can be build that each refer to the same data. The difference is a subtle one, basically you can manage the reference counting your self, or you can let A++ handle it for you, we suggest the latter, but either will work fine[2].

5.3 Interoperability with different languages

A++ can be mixed with other languages quite easily using the C++ `extern "C"` interface. The details of doing this are a C++ issue and is the standard way that C++ is mixed with C language code. Mixing C++ with FORTRAN is unfortunately somewhat compiler dependent. Beyond the C++ to C, or C++ to

[2]We think it is easy to create errors if users are forced to manage such details explicitly.

FORTRAN[3], the mixing of A++ and FORTRAN or C is provided by low level access to the raw data which contains the data values in the A++ array object. Additional member functions are provided to obtain size and view information that is required for interpretation of the raw data pointer (required in the case of a view).

The A++ array class also provides scalar indexing, but scalar indexing is not efficient in A++ since depending on the degree of optimization within the compiler, the inlined functions are not well optimized.

5.4 Temporary Handling

A++ array objects manage the temporaries that their use in array expressions create. In the expression `A = B + C`, one temporary is built to hold the result of `B + C`, as in `Temp = B + C`. This temporary is then used in the expression `A = Temp` to finish the assignment. In this case the assignment can be optimized and the actual assignment of elementwise data avoided by allowing A to copy the pointer to the temporary's raw data. So the operations would be an elementwise addition of B and C and then a copy of a pointer. Such trivial operations perform the same as lower level C for FORTRAN code. The detail regarding temporary management is that the assignment operator that copies the pointer also deletes the temporary `Temp`. It is not clear from a single line of code, but in longer functions that might contain many A++ array expressions, if we failed to manage the lifetime of the temporaries we would allocate one for each array expression. The temporaries would accumulate and waste significant memory resources. For example:

```
// Here we make up a fictitious array class that does not manage temporaries
// we will call this class, analogous to a doubleArray, No_Temporary_Management_doubleArray
void Waste_Memory ()
   \{
     No_Temporary_Management_doubleArray A(100,100,100),
                                         B(100,100,100),
                                         C(100,100,100),
                                         D(100,100,100);

  // statement repeated to show wasted space from long
  // function with many array statements.
     A = B + C + D;
     A = B + C + D;
     A = B + C + D;
     A = B + C + D;
     A = B + C + D;
     A = B + C + D;
```

[3]interestingly the reverse direction is possible as well, but just requires usage of the mangled names (and knowledge of what the mangled names are)

```
\}
```

In this example function, if there was no temporary management then the `C + D` would generate a temporary and the lifetime of that temporary would be the local scope of the function[4]. Since the temporary has local scope its destructor is called when the function exits. In this case we are assuming no temporary handling so `C + D` would generate a temporary and the `B + Temp` would generate a temporary, and then the assignment would be done. Since we assume no temporary handling the assignment operation would likely do an explicit elementwise copy of `A = Temp2`. Thus each line generates two temporaries and there are 6 lines, so we have accumulated 12 temporaries at the end of the function. **Note that this is not the way that A++ works, but is motivation for the temporary handling that A++ provides.** As the function exits the destructor is called and the 12 temporaries are released. Until that point of function exit we had wasted 12 million double words of memory[5].

A++ implements temporary handling which minimizes the number of temporaries that would otherwise accumulate within the execution of array statements. By building the temporaries with a scope that is controlled by A++, the A++ functions internally control the lifetimes of the A++ temporaries[6]. Then operations using A++ array objects check to see if they have a temporary object and if so provide more efficient handling of the operation. For example, in the previous function, the result of `C + D` would generate a temporary, but then `B + Temp` would reuse the temporary array object as an accumulator[7]. Then the assignment operations would recognize the temporary object and copy the pointer the raw memory and delete the (now empty) temporary array object. At the end of the expression there are no temporaries that have accumulated. This is the superiority of this execution model for A++ array objects. Typically at most one temporary is created during the evaluation of an array expression, and none are allowed to accumulate across expressions. For large scale computations this is an important feature of the temporary handling.

The drawback to temporary handling is that if we pass an expression to a function then the first use of the function's parameter will *handle* the parameter right out of existence. To fix up this special case we provide the **evaluate()** function which converts the temporary to a non-temporary to avoid confusing the A++ aggressive temporary management. Note that if a non temporary is

[4]Technically it must be at least as long as the statement and no longer than the local function scope, it is complier dependent (which is the worst of all solutions since it is not consistent within different C++ compilers), unfortunately AT&T Cfront based compilers such as what are most readily available on many high performance computers (Cray YMP, C-90, etc.) assume the temporary would have local scope and many PC compilers (and GNU g++, not a PC compiler) assumes the opposite.

[5]And that was just a little function, more meaningful functions could easily exhaust available computer memory resources.

[6]Specifically, all A++ binary operations return A++ array objects by reference and mark the objects internally as temporary.

[7]this works especially well in long expressions.

passed in to the **eval()** function, then a locally scoped shallow copy is built[8].

5.5 How to abuse A++

Like most good things, there are some ways to break A++, most of them are along the lines of using the access A++ provides to you to get to its low level data and then deleting or changing that data in some way.

- deleting low level A++ array data

```
doubleArray A (100,100);
double *Raw_Data_For_A = A.getDataPointer();
delete Raw_Data_For_A;    // error: delete the data that was allocated for A
```

- passing expressions by reference (without calling **evaluate()**)

```
void foo ( const doubleArray & X )
   {
     X = X * X;  // if X is a temporary (as in from an expression)
                 // then X well be deleted by the operator*
                 // then the assignment using operator= would be
                 // invalid.
   }
doubleArray B (100,100);
foo ( B * B );             // error
foo ( evaluate(B * B) );   // correct
```

- not checking for a view when using the raw data from an array object

```
   doubleArray C (100,100);
   Range I (10,89,2);
// Now get a pointer to the data containd in a view of C using the Range object, I.
   double *Raw_Data_For_C = C(I,I).getDataPointer();

// The wrong way to access the raw data.
   for (int j=0; j < C.getLength(1); j++)           // error: Access of raw data does not
        for (int i=0; i < C.getLength(0); i++)      //        account for view, specifically
             Raw_Data_For_C [j*getLength(1)+i] = 0; //        the base, bound, and stride.

// The correct way is more complex (mostly becase this is a 2D example)
// This example assumes a very general array C with nonzero base and
// general strided view.
   doubleArray & D = C(I,I);  // avoid recomputing the view C(I,I);
   int Base_0   = D.getBase(0) - C.getBase(0);  // we want the offset from the base of D
   int Base_1   = D.getBase(1) - C.getBase(1);  // we want the offset from the base of D
   int Bound_0  = D.getBound(0) - C.getBase(0);// we want the offset from the bound of D
   int Bound_1  = D.getBound(1) - C.getBase(1);// we want the offset from the bound of D.
```

[8]This has the effect of doing what the user would expect without the **evaluate()** function call.

```
// The following assume that the stride of C might not be 1, but in this case we know it is 1.
// for example, the stride 2 view of a stride two array (another view for example)
// would be a stride 4 access of the raw data.
   int Stride_0 = D.getStride(0) * C.getStride(0); // we want the stride of the raw data
   int Stride_1 = D.getStride(1) * C.getStride(1); // we want the stride of the raw data
// This assumes that the length of C is really the length of the raw data
   int Size_0   = C.getRawDataSize(0);

// Note that many compilers will not lift the loop invariant part
// "j*Size_0" and so for such compilers a more efficient looping
// structure is possible (but not shown here)
   for (int j=Base_1; j <= Bound_1; j += Stride_1)      // correct: Access of raw data does not
        for (int i=Base_0; i <= Bound_0; i += Stride_0) //          account for view, specifically
             Raw_Data_For_C [j*Size_0+i] = 0;           //          the base, bound, and stride.
```

These code fragments show the incorrect usage of the low level access that A++ provides. It is not a goal of A++ to protect the user from himself/herself.

Chapter 6

P++

6.1 Goals of the P++ development

The general goal of the P++ development is to provide a simplified parallel programming environment. In this section some ideal requirements for a user interface and programming model for distributed memory architectures are stated. These are fulfilled with the P++ environment for a large, but restricted, class of problems:

- Algorithm and code development should take place in a serial environment.
- Serial source codes should be able to be compiled and recompilable to run in parallel on distributed architectures without modification.
- Codes should be portable between different serial and parallel architectures (shared and distributed memory machines).
- Vectorization, parallelization and data partitioning should be hidden from the user, except for optimization switches to which the user has full access and that have meaning only on vector or parallel environments.

6.2 Partitioning Objects

P++, being of object-oriented design, introduces an object based control of the partitioning of the array object. Specifically we introduce a partitioning object which can be used to build P++ arrays (via a parameter to the P++ array constructor) or modify existing arrays previously built. It is a principle feature of the P++ partitioning objects that all array objects built with a given partitioning are associated with that partitioning object. Manipulation of the partitioning object effects all array objects with which it is associated. For example, specifying a new range of processors for a partitioning object repartitions the P++ array objects associated with that partitioning object. While not important for more simple single grid applications, the level of control provided by partitioning objects is intended for control (and load balancing) of applications containing many grids (e.g. adaptive mesh refinement and overlapping grid applications).

Partitioning objects are provided to allow both user and programmable control. Load balancers would use the programmable control which represents a separate interface to the partitioning objects. Users would more directly use the more simple interface for the specification of the partitioning of an array object. A++ arrays provide member functions as the means of associated an array with a particular partitioning object. This interface allows for a simplified manipulation of many partitioning objects (and thus even more P++ array objects) within a single application.

For example, an adaptive mesh refinement (AMR) grid could contain many array objects associated with each rectangular grid (there would be many rectangular grids within an AMR application) and a single partitioning object for rectangular grid. Within an adaptive mesh refinement grid many grids and thus many partitioning objects would exist. The control of the adaptive mesh refinement grid and the array objects associated with the definition of the application can thus be abstracted through the control of the partitioning objects associated with the adaptive grid.

```
Partitioning_Type P;
doubleArray A(100,100,P); // A uses the partition object P
doubleArray B(100,100);   // B has the default partition object
B.partition(P);           // repartitioning B consistent with P
Range Middle_Processors (100,199);  // specify a fictitious collection of process
Partitioning_Type Q(Middle_Processors);  // build another partitioning object
B.partition(Q);           // B is repartitioned onto the middle 100 processors
```

The example above builds a partitioning object, **P**, which has the default partitioning (across all processors and partitioned along each axis). The array **A** is built to have a partitioning across the processors specified by **P**. The array, **B**, is build with the default partitioning and then repartitioned to be consistent with the partitioning specified by **P** (since **P**, in this case, represented the default partitioning the distribution of **B** is unchanged). The partitioning **Q** is build over the processor 100 through 199, and **B** is repartitioned onto that smaller collection of processors.

6.3 How P++ Arrays are Partitioned

P++ provides multidimensional partitioning of its distributed array objects. The limit of the dimensionality of the partitioning is that of the dimension of the array. The partitioning is effected by the **Partitioning_Type** objects, any two arrays of the same size using the same partitioning object will be partitioned identically (i.e. they will be aligned on the same processors). See the section on **Partitioning_Type** objects for more information. This section will provide examples of how arrays are partitioned in the near future.

6.4 Ghost Boundaries

The partitioning objects contain many features, detailed in the reference section, but in addition to the layout specifications they control the default widths of shared regions along interior partition edges, so called "ghost boundaries." The default width of ghost boundaries defined for a P++ array object is defined by the partitioning object to which it is associated. P++ array objects may additionally override the ghost boundary width specified by the partitioning objects to which they are associated. This allows many arrays to be associated with a specific partitioning object yet restrict the ghost boundary width to be nonzero on only a subset of the associated array objects.

P++ arrays may be modified to include an arbitrary width internal ghost boundary, the default width at present is ZERO, though a better choice of a default width maybe made after more feedback. The purpose of this feature is to permit specific subsets of the array objects associated with a partitioning object to have different ghost boundary widths.

6.5 Communication Models

There are two communication models in P++, the Overlap Update Model and Virtual Shared Grid Model. These handle the interpretation of message passing at each binary operation, assuming that either the partitioning or the indexing would force message passing, either messages are passed to satisfy the binary operation or the message passing will be deferred until the "equals" operator. In the Overlap Update Model message passing within array expressions is deferred until the "equals" operator, while in the Virtual Shared Grid Model message passing is done in each binary operator. These communication models are discussed more fully in the section about A++/P++ programming models.

6.6 How to abuse P++

There are several interesting ways to abuse the P++ programming model. This section is intended to parallel the similar section "How to abuse A++" in section **??**, the methods listed there apply to P++ as well, but P++ has some additional ways in which the user can generate errors. As in A++, all these methods stem from the access that P++ provides the user to low level data or operations. The following example will cause inconsistant storage within the 5th element of the array **A**. It could eventually lead to a more serious error.

```
// Assume that A is an array with ghost boundaries of width greater than zero.
// And that element 5 of A is at an edge of a local processor space
   intArray A(10);
   A = -100;  // initialize A to a negative value (since processor number are >= 0.
   Optimization_Manager::setOptimizedScalarIndexing(On);
   A(5) = Communication_Manager::localProcessNumber();
```

```
    Optimization_Manager::setOptimizedScalarIndexing(Off);
```

The example uses the **Optimization_Manager::setOptimizedScalarIndexing()** function which turns off communication which would otherwize be done within all scalar indexing. The purpose of this function is to permit more efficient scalar indexing for the case when the user knows that NO off processor access is possible (on each processor). If **A** has ghost boundaries then it has multiple positions for some data (data within the ghost boundary width of the partition boundaries) on any two processors.

The problem within the example is that the value returned by **Communication_Manager::localProcessNumber()** will be different for each processor. This is the problem, it would not be a data parallel operation and would result in different values being stored (one in the processor owning the local space where **A(5)** is located, and one in the neighboring processor storing a copy of **A(5)** within its ghost boundary). The problem could be resolved if the ghost boundaries where updated, but nothing within normal P++ operations requires the user to call the ghost boundary update functions directly, so this is considered an error.

The reason this happens is because P++ makes the local processor number available, but we would lose flexibility if we did not make such info available to the user. So it is the user's responsibility to use P++ wisely.

Chapter 7

Developing A++/P++ Applications

There are several details to the development of A++ and P++ applications, this section is intended to present them to new users. This document is intended to be especially useful to new users at LLNL, but most details are the same everywhere. It is assumed that you have A++ and P++ installed. If only A++ has been installed then the P++ section should be ignored. There are not many details to using A++, only P++ (since it uses MPI (or PVM)) has details for which new users should be aware.

Some sites, such as LLNL, use **ssh** as part of their security, this has special significance if you want to run MPI on such a network. So we cover the setup of **ssh** specific to avoiding the request for a password when logging into other machines on the network (even your own machine).

7.1 If You Use SSH On Your Network

If you use **ssh** instead of **rcp** within your version of MPI (consult the person who installed MPI if this is not clear) then yo have an additional step to allow you to run MPI applications (your P++ applications will be an MPI application). This applies to all people working at LLNL.

7.1.1 Why worry about SSH

SSH is a secure mechanism for logging-on to remote machines. The process of running MPI applications IS a process by which MPI (mpirun, specifically) logs on to remote machines to run your applications in a distributed way. SSH will force each process started to log on to the machine where it will run and this will cause it to prompt you for a password. **Even if your running all processes on your own machine.**

For example, if you run on 35 processors you will have to enter you password 35 times, clearly this is not what anyone wants. This section details how to setup **ssh** so that it will **trust** a number of machines that you select and you can run parallel MPI programs without this hassle. This is not a P++ issue, it is an MPI issue when MPI is installed to use **ssh** instead of **rpc** (which is the default for MPI).

7.1.2 Setting up SSH to run MPI Programs

These are directions provided by Brian Miller for the setup of ssh.

To ssh from $HOME to $REMOTE:

If you don't care about complying with the security request to not have .ssh on the common file system,

1. cd ~/.ssh on $REMOTE (make it if it doesn't exist)
2. edit authorized_keys on $REMOTE:~/.ssh (create it if it doesn't exist)
3. copy all lines identity.pub to authorized_keys (from $HOME:~/.ssh/identity.pub, use ssh-keygen on $HOME if this file doesn't exist)
4. make sure permissions are correct (use chmod 600 for authorized_keys)

If you want to comply with the security request, the steps are similar:

1. ssh $REMOTE
2. cd /var/ssh
3. mkdir $USER; chmod 700 $USER; cd $USER
4. mkdir .ssh; chmod 700 .ssh; cd .ssh
5. edit authorized_keys (create if doesn't exist)
6. copy data from $HOME:~/.ssh/identity.pub to $REMOTE:/var/ssh/$USER/.ssh/authorized_keys (again, use ssh-keygen on $HOME if this doesn't exist)
7. save authorized_keys
8. chmod 600 authorized_keys
9. cd ~
10. ln -s /var/ssh/$USER/.ssh ~/.ssh

done for this $REMOTE, repeat for blue099, blue199, west, tc01, tc02,....

7.2 Developing And Running A++/P++ Applications

This section details what you should have to know to develop and execute A++ and P++ applications. We assume that A++ and P++ are already installed and tested using the automated mechanisms described in the installation process.

In general all P++ applications start out as A++ applications which are then recompiled with P++ instead of A++ and run in parallel. The development of the application can take place in either environment, so parallel P++ applications can be developed on a parallel machine directly (though in general parallel machines are considerably less friendly than serial machines for application development).

7.2.1 A++ Applications

A++ applications are developed as source code that:

1. include `#include <A++.h>` at the top,
2. defines an `int main(int argc,char* argv[])` procedure (somewhere in the system of files representing the application),
3. compiles with options and paths so that the A++ header files can be found (-I<your install directory>/A++P++/A++/include),
4. links with the appropriate A++ libraries (`-App -App_static`)

This is normal program development, nothing is special, P++ is a more more complex.

How to run A++ Applications

A++ applications are just standard applications. Executing an A++ application is the same as for any other program you write.

7.2.2 P++ Applications

P++ applications are developed as source code that:

1. include `#include <A++.h>` at the top,
2. defines an `int main(int argc,char* argv[])` procedure (somewhere in the system of files representing the application),
3. compiles with options and paths so that the P++ header files can be found (-I<your install directory>/A++P++/P++/include),
4. links with the appropriate P++ libraries (`-Ppp -Ppp_static -mpi`)

Clearly the process is nearly identical to that of an A++ application (by design).

7.2.3 How to run P++ with MPI

P++ applications are just standard MPI applications. And running an MPI applications is a bit more complex than running a standard serial application.

An MPI application requires the use of a supporting program named `mpirun`, a P++ application is handed in (together with any command line parameters) as a command line parameter to the `mpirun` program. The correct syntax is:

`mpirun -np <numberOfProcessors> <application> <application command-line args>` .

Additional options to `mpirun` can be seen by typing `mpirun -help`, though we only need at most two (`-np` and `-machinefile`). Many machines only require (`-np`). A list of machines specific to LLNL and which options they require can be found at the end of this section.

Specification of a machine file (`-machinefile` option)

On some (most) machines, `mpirun` requires the specification of a `machine file` using the `-m` option (to `mpirun`). This file specifies the machines on the network that the users distributed application will run. For testing purposes all the machine entries in this file can be the same. An example `machine file` (a simple ASCII file) would be:

```
gibs
```

In this case the MPI processes would run on gibs (all of them!)[1]. As many machines as you like can be specified within the machine file. Allocation of processes to machines is based on a round-robin scheduling of the number of processes specified on the `mpirun` command line (using the `-np` option) and the entries in the `machine file`.

Example using a machine file (as it appears when running `make check` in the P++/EXAMPLES directory):

```
mpirun -np 6 -machinefile /home/dquinlan/A++P++/A++P++Source/A++P++/machine.file test_Ppp_execut
```

Notice in this case that only 6 processors are used, this is for test purposes only on a small network of workstations.

Running on a specific machine (your machine)

In general running on any machine is a matter of looking at the command line used in the testing of P++ on that machine where P++ is installed. More details information will later be documented about running on specific machines; Blue Mountain, Tera Cluster, Blue Pacific, Red, etc.

[1]In this case, since **gibs** is Bill's machine, you would likely get email from Bill :-).

A note about using P++ with PVM

We presently are using MPI for the development of P++, we test the implementation with PVM from time to time, but since it is not part of the development environment on a regular basis, its support can lag that of the MPI implementation. If you notice a problem, let us know. We are always looking for any differences in the PVM and MPI support (because there should be none).

7.2.4 How to run P++ with MPI

P++ applications are just standard PVM applications. Much of the development of P++ was initially done using PVM. So we include the setup specific to PVM. LLNL users should ignore this section.

The frustrating part is getting your environment setup to allow you to run PVM. To do this you must:

```
1. Add /usr/local/pvm/lib to path.

2. Add /usr/local/pvm/man to MANPATH.  This isn't necessary to
   make pvm run but is helpful to provide documentation for pvm.

3. Add the following before lines that exit .cshrc if not an
   interactive shell.

   setenv PVM_ROOT /usr/local/pvm

   if (! $?PVM_ROOT) then
      if (-d ~/PVM/pvm3) then
setenv PVM_ROOT ~/PVM/pvm3
      else
echo PVM_ROOT not defined
echo To use PVM, define PVM_ROOT and rerun your .cshrc
      endif
   endif

   if ($?PVM_ROOT) then
     setenv PVM_ARCH `$PVM_ROOT/lib/pvmgetarch`
     set path=($path $PVM_ROOT/bin/$PVM_ARCH)
   endif

Also delete any files named /tmp/pvmd.{\it pid} where {\it pid} is an old process
id number before starting pvm.
```

You can test the pvm installation using the test codes bundled with the pvm distribution. You can add new machines to the pvm environment and get help from the pvm console. For example:

```
572 object> pvm
pvm>
pvm> add fenris
1 successful
                   HOST     DTID
                 fenris    80000
pvm>
pvm> conf
2 hosts, 1 data format
                   HOST     DTID     ARCH   SPEED
                 object    40000     SUN4    1000
                 fenris    80000     SUN4    1000
pvm>
pvm> help
HELP   - Print helpful information about a command
Syntax:  help [ command ]
Commands are:
  ADD    - Add hosts to virtual machine
  ALIAS  - Define/list command aliases
  CONF   - List virtual machine configuration
  DELETE - Delete hosts from virtual machine
  ECHO   - Echo arguments
  HALT   - Stop pvmds
  HELP   - Print helpful information about a command
  ID     - Print console task id
  JOBS   - Display list of running jobs
  KILL   - Terminate tasks
  MSTAT  - Show status of hosts
  PS     - List tasks
  PSTAT  - Show status of tasks
  QUIT   - Exit console
  RESET  - Kill all tasks
  SETENV - Display or set environment variables
  SIG    - Send signal to task
  SPAWN  - Spawn task
  TRACE  - Set/display trace event mask
  UNALIAS - Undefine command alias
  VERSION - Show libpvm version
pvm>
```

Alternatively, you can have a large collection of machines added when you first run PVM by putting a list of machines into a file (one machine name per line) and adding the filename as a parameter when you start PVM. For example, my pvm_hosts file is:

```
fenris
```

```
#
# Comment these out to restrict usage to a single machine (guarneri)
guarneri
oogle
ralphie
sanctus
tasha
uppsala
```

For example, "pvm pvm_hosts", adds the machines listed in the file "pvm_hosts" to the pvm environment. When you exit pvm, pvm remains running in the background. The kill pvm you should use the "halt" command from the pvm console.

Chapter 8

Tutorial

8.1 Introducton

The A++/P++ Library represents array classes written in C++, which seek to simplify scientific programming by providing a general object-oriented framework in which to develop **both** serial (A++) and parallel codes (P++). It is intended to be simple, abstracting away much of the architecture dependence and "bookkeeping" associated with scientific (especially parallel) programming, allowing the researcher/programmer to concentrate on the rapid development of algorithms and/or production of stable software. For more information see the A++/P++ Manual or the A++/P++ Home Page (listed on the front cover).

The A++/P++ is focused on arrays as objects, which encapsulate both data and the operations which can be performed on that data (methods). This approach allows, the programmer to use the A++/P++ data types (intArray, floatArray and doubleArray) much like they currently use the primitive types (int,float and double) available in standard C++. P++ uses a SPMD (single program multiple data) implementation of a data parallel programming model. The data parallel model is implemented using two communication models, which allow 1)for efficient communication between aligned and unaligned array operations and 2) the necessary congruence between serial and parallel libraries.

This tutorial steps through a number of example A++/P++ programs. The examples illustrate some the main concepts in the A++/P++ including: abstraction of the user from machine dependencies, reuse of serial code in a parallel environment, dimension independence in scientific computations, access to FORTRAN 77 (mixed language programming), etc. We present whole (yet simple) A++/P++ applications, the example applications are kept small so as to be presentable in this tutorial style. Each example generally contains 1) A brief introduction 2) The A++/P++ source code, which includes numerous comments discussing the various ways used to the A++/P++ data structures and associated methods 3) Output from Code.

8.2 Examples

8.2.1 Example 1a. ”Hello, World”

This is the simplest A++/P++ example. It illustrates some of the basic features of A++/P++.

```
#include <A++.h>   // this is included in every A++/P++ application
int main(int argc,char** argv)
{
// We are instancing the doubleArray object.  Though it looks like a
// standard Fortran array, it's not
doubleArray A(10);
doubleArray B(10);

// Initialize A and B
A=2;
B=3;

Illustration of the methods associated with doubleArray Objects
``display'' is used to show the values of the Object

A.display(``This is the doubleArray Object A'');
B.display(``This is the doubleArray Object B'')

// We can add to array objects with the ``+'' operator
A=A+B;
A.display(``Addition of A and B'');
}
```

The output from the “Hello,World” program.

```
doubleArray::display() (CONST) (Array_ID = 1) -- This is the doubleArray Object A
Array_Data is a VALID pointer = 3c000 (245760)!
AXIS 0 --->: (   0) (   1) (   2) (   3) (   4) (   5) (   6) (   7) (   8) (   9)
AXIS 1 (  0) 2.0000 2.0000 2.0000 2.0000 2.0000 2.0000 2.0000 2.0000 2.0000 2.0000

doubleArray::display() (CONST) (Array_ID = 2) -- This is the doubleArray Object B
Array_Data is a VALID pointer = 3e000 (253952)!
AXIS 0 --->: (   0) (   1) (   2) (   3) (   4) (   5) (   6) (   7) (   8) (   9)
AXIS 1 (  0) 3.0000 3.0000 3.0000 3.0000 3.0000 3.0000 3.0000 3.0000 3.0000 3.0000

doubleArray::display() (CONST) (Array_ID = 1) -- Addition of A and B
Array_Data is a VALID pointer = 44000 (278528)!
AXIS 0 --->: (   0) (   1) (   2) (   3) (   4) (   5) (   6) (   7) (   8) (   9)
AXIS 1 (  0) 5.0000 5.0000 5.0000 5.0000 5.0000 5.0000 5.0000 5.0000 5.0000 5.0000
```

8.2.2 Example 1b. “Parallel Hello World”

The program below is a parallel version of the Example 1a., and illustrates one of the guiding ideas behind A++/P++, serial code reuse. With the addition of 3 lines, the serial code above becomes an SPMD parallel code .

```
#include <A++.h>   // this is included in every A++/P++ application

int main(int argc,char** argv)
{
// The next two lines are needed to "parallelize" the serial code.
Number_of_Processors=2;
Optimization_Manager::Initialize_Virtual_Machine(" ",Number_of_Processors,argc,argv);

// Instantiation of the doubleArray Object, (notice the similarity to a Fortran array)
doubleArray A(10);
doubleArray B(10);

// Initialize A and B
A=2;
B=3;

// Illustration of the methods associated with doubleArray Objects

// ‘‘display’’ is used to show the values of the Object
A.display(‘‘This is the doubleArray Object A’’);
B.display(‘‘This is the doubleArray Object B’’)

// We can add array objects with the ‘‘+’’ operator
A=A+B;
A.display(‘‘Addition of A and B’’);

// 3rd (and final) line necessary to parallelize code.
 Optimization_Manager::Exit_Virtual_Machine();


}
```

The calls to the OptimizationManager are required because we must specify some information to the message passing libraries (PVM or MPI). For PVM we require 1). The number of Processes to be started 2). The name of the executable that each process should start. MPI requires the argc and argv arguments. The final P++ specific call

```
Optimization_Manager Exit_Virtual_Machine()
```

shuts down the virtual machine. The specification of the number of processors is a specification of the virtual process space, and independent of the number of processors physically available. At present we use MultiBlock Parti within P++, this corresponds to the initialization of the virtual processor space within MultiBlock Parti[1]. The programs above use only one of the 3 type of array objects available in A++/P++. The other object types being intArray and floatArray.

8.2.3 Example 2. 1-D Laplace Equation Solver

This example program solves the 1-D Laplace equation, $U_{xx} = 0$ subject to U(0)=1 and U(1)=1 with Jacobi relaxation.

```
//This example illustrates the "proper" use of the A++/P++ libs.
// The idea is to avoid scalar indexing (eg. the kind of indexing
// you normally do in fortran or C)  through the use of
// the Index and Range Objects.   Scalar indexing is
// very slow, especially for P++, inwhich the arrays are distributed
// over the processors, and considerable amount of communication is necesary to
// retrieved the indexed values.


#include <A++.h>
#include <time.h>

main(int argc,char** argv)
{
int num_of_process=4;
Optimization_Manager::Initialize_Virtual_Machine(" ",num_of_process,argc,argv);

// Instance the doubleArray objects //

int grid_size=10;
doubleArray Solution(grid_size);
doubleArray Solution2(grid_size);
doubleArray temp(grid_size);

//Other variables
double time1,time2,time_total,time2_total;
double Jacobi=5;  // number of steps in the Jacobi relaxation
int i,j;

//Instance the Range(or Index) objects
Range I(1,grid_size-2,1);

//Initialize the  doubleArray objects//
```

[1]This is a library available from the University of Maryland

```
Solution=0.0;
Solution2=0.0;
Solution(I)=1.0;
Solution2(I)=1.0;


// Solving 1-d equation using Index object.
time1=clock();
for (i=1;i<=Jacobi;i++){
Solution(I)=(Solution(I-1)+Solution(I+1))/2; }
time2=clock();
Solution.display("index");
time_total=time2-time1;
printf("index done");

// equivalent expression with scalar (array) indexing //
time1=clock();
for (i=1;i<=Jacobi;i++){
for (j=1;j<=8;j++){
temp(j)=(Solution2(j-1)+Solution2(j+1))/2;}
for (j=0;j<=9;j++){
Solution2(j)=temp(j);}}
time2=clock();
time2_total=time2-time1;
Solution2.display("scalar");

// times taken by
cout <<time_total<<" "<<time2_total<<"\n";


printf("program terminated properly");

Optimization_Manager::Exit_Virtual_Machine();
}
```

Output from Example 2:

```
==================================================
Application_Program_Name set to something (Application_Program_Name =
/n/c3servew/nehal/testcode/pring)
My Task ID = 262149
My Process Number = 0

*****************************************************
P++ Virtual Machine Initialized:
         Process Number            = 0
         Number_Of_Processors      = 2
         Application_Program_Name  = /n/c3servew/nehal/testcode/pring
*****************************************************
```

```
doubleArray::display() (CONST) (Array_ID = 1) -- index
SerialArray is a VALID pointer = 6c000!
doubleSerialArray::display() (CONST) (Array_ID = 8) -- index
Array_Data is a VALID pointer = 82000 (532480)!
AXIS 0 --->: (   0) (   1) (   2) (   3) (   4) (   5) (   6) (   7) (   8) (   9)
AXIS 1 (  0) 0.0000 0.3125 0.6250 0.7812 0.9062 0.9062 0.7812 0.6250 0.3125 0.0000

index donedoubleArray::display() (CONST) (Array_ID = 3) -- scalar
SerialArray is a VALID pointer = 6c024!
doubleSerialArray::display() (CONST) (Array_ID = 8) -- scalar
Array_Data is a VALID pointer = 82000 (532480)!
AXIS 0 --->: (   0) (   1) (   2) (   3) (   4) (   5) (   6) (   7) (   8) (   9)
AXIS 1 (  0) 0.0000 0.3125 0.6250 0.7812 0.9062 0.9062 0.7812 0.6250 0.3125 0.0000
program terminated properlyExiting P++ Virtual Machine!

110000 210000 // "times" for Index and scalar indexing
```

8.2.4 Example 3. Distribution of Arrays in P++

This example illustrates the partitioning of arrays by P++.

```
//    This example shows the "partitioning" of arrays
//     with the use of the Paritioning_Type object
//    It will also illustrate the manipulation of a "local array", within P++.
//
#include <A++.h>
#include <P++.h>
main(int argc,char** argv)
{
int num_of_process=10;
Optimization_Manager::Initialize_Virtual_Machine("",num_of_process,argc,argv);

//Build partition object which uses 5 processors (0-4)
Partitioning_Type PartitionA(3);
//Now divide intArray A among the Processors
intArray A(10,10,PartitionA); // A is partitioned among the  first 3 processors
      // if no  partitiong object is specified then
      // the Array is paritioned among the total
      // number of processors (in this case 10)

// Assign "A" an initial value with Index Operators
A=10;
// We can use a mix of Index object(s) and scalar indexing to assign
// values to A
Index I(0,7);    // I=[0..7];
A(I,1)=1;         // Notice that we can mix the Index operator and a scalar index
A(I,2)=2;
A(I,3)=3;
// Display "A".  A++ uses a FORTRAN style array A(cols,rows). See
// the output.  Each processor prints out it's local piece of
the distribted array
A.display();

// As stated above, P++ is single program multiple data (data parallel), so a single
// P++ program is running on all the processors.  However, each processor has
// only a small portion of the global data. This data is paritioned automatically
// P++, and communication is done implicitly after each each statement
// In the case of <type>Array, each processor
// keeps a small amount of the global Array, which is infact an A++ Array
// object.  Thus we can if we wish extract and manipulate "local" data

// Extract "Local_Array" from the global Array A
intSerialArray Local_Array=A.getLocalArray();


// Let's use some of the "size"  methods in A++
int Num_of_Cols=Local_Array.elementCount(); // total size of Local_Array
int Base_0_axis=Local_Array.getBase(0);   // base value for 0 axis
int Bound_0_axis=Local_Array.getBound(0);  // bound for 0 axis


// Display "Local_Array".  If you are using PVM look in your
// pvml file to see results (usually in the /tmp directory).
Local_Array.display();
}
```

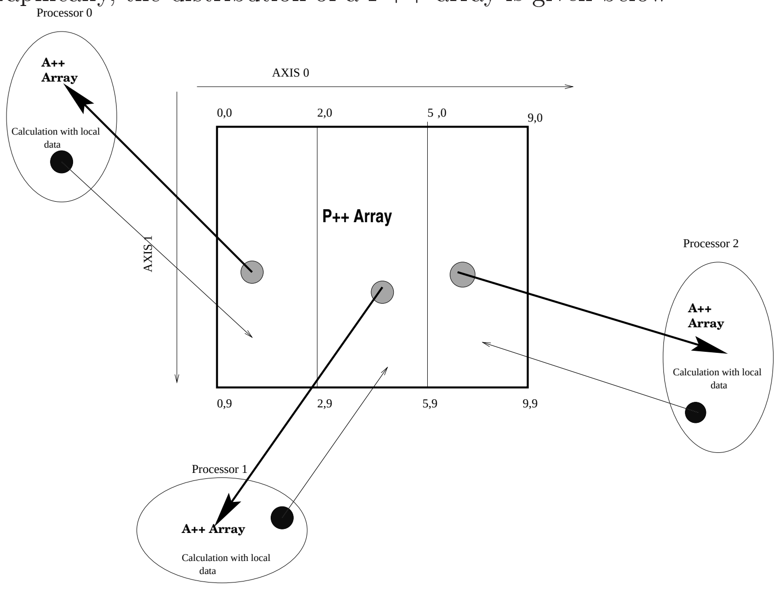
Output from example 3

```
Application_Program_Name set to something (Application_Program_Name =
/n/c3servew/nehal/testcode/distrib)
My Task ID = 262199
####  My Process Number = 0

****************************************************
P++ Virtual Machine Initialized:
         Process Number             = 0
         Number_Of_Processors       = 10
         Application_Program_Name   = /n/c3servew/nehal/testcode/pringle2
****************************************************

intArray::display() (CONST) (Array_ID = 1) --
SerialArray is a VALID pointer = 6e000!
intSerialArray::display() (CONST) (Array_ID = 4) --
Array_Data is a VALID pointer = 82000 (532480)!
AXIS 0 --->: (   0) (   1) (   2) (   3) (   4) (   5) (   6) (   7) (   8) (
9)
```

```
AXIS 1 (  0)   10   10   10   10   10   10   10   10   10   10
AXIS 1 (  1)    1    1    1    1    1    1    1   10   10   10
AXIS 1 (  2)    2    2    2    2    2    2    2   10   10   10
AXIS 1 (  3)    3    3    3    3    3    3    3   10   10   10
AXIS 1 (  4)   10   10   10   10   10   10   10   10   10   10
AXIS 1 (  5)   10   10   10   10   10   10   10   10   10   10
AXIS 1 (  6)   10   10   10   10   10   10   10   10   10   10
AXIS 1 (  7)   10   10   10   10   10   10   10   10   10   10
AXIS 1 (  8)   10   10   10   10   10   10   10   10   10   10
AXIS 1 (  9)   10   10   10   10   10   10   10   10   10   10
intSerialArray::display() (CONST) (Array_ID = 2) --
Array_Data is a VALID pointer = 7e000 (516096)!
AXIS 0 --->: (   0) (   1) (   2)
AXIS 1 (  0)   10   10   10
AXIS 1 (  1)    1    1    1
AXIS 1 (  2)    2    2    2
AXIS 1 (  3)    3    3    3
AXIS 1 (  4)   10   10   10
AXIS 1 (  5)   10   10   10
AXIS 1 (  6)   10   10   10
AXIS 1 (  7)   10   10   10
AXIS 1 (  8)   10   10   10
AXIS 1 (  9)   10   10   10


Output in the pvml files
==================
[t80040000] [t40042] My Task ID = 262210
[t80040000] [t40042] My Process Number = 1
[t80040000] [t40042]
[t80040000] [t40042] *****************************************************
[t80040000] [t40042] P++ Virtual Machine Initialized:
[t80040000] [t40042]  Process Number             = 1
[t80040000] [t40042]  Number_Of_Processors       = 10
[t80040000] [t40042]  Application_Program_Name =/n/c3servew/nehal/testcode/distrib
[t80040000] [t40042] *****************************************************
[t80040000] [t40042]
[t80040000] [t40042] AXIS 0 --->: (   3) (   4) (   5)
[t80040000] [t40042] AXIS 1 (  0)   10   10   10
[t80040000] [t40042] AXIS 1 (  1)    1    1    1
[t80040000] [t40042] AXIS 1 (  2)    2    2    2
[t80040000] [t40042] AXIS 1 (  3)    3    3    3
[t80040000] [t40042] AXIS 1 (  4)   10   10   10
[t80040000] [t40042] AXIS 1 (  5)   10   10   10
[t80040000] [t40042] AXIS 1 (  6)   10   10   10
[t80040000] [t40042] AXIS 1 (  7)   10   10   10
[t80040000] [t40042] AXIS 1 (  8)   10   10   10
[t80040000] [t40042] AXIS 1 (  9)   10   10   10


[t80040000] [t40043] My Task ID = 262211
[t80040000] [t40043] My Process Number = 2
[t80040000] [t40043]
[t80040000] [t40043] *****************************************************
[t80040000] [t40043] P++ Virtual Machine Initialized:
[t80040000] [t40043]   Process Number             = 2
[t80040000] [t40043]   Number_Of_Processors       = 10
```

```
[t80040000] [t40043]   Application_Program_Name   =/n/c3servew/nehal/testcode/pringle2
[t80040000] [t40043] **************************************************
[t80040000] [t40043]
[t80040000] [t40043] AXIS 0 --->: (   6) (   7) (   8) (   9)
[t80040000] [t40043] AXIS 1 (  0)   10   10   10   10
[t80040000] [t40043] AXIS 1 (  1)    1   10   10   10
[t80040000] [t40043] AXIS 1 (  2)    2   10   10   10
[t80040000] [t40043] AXIS 1 (  3)    3   10   10   10
[t80040000] [t40043] AXIS 1 (  4)   10   10   10   10
[t80040000] [t40043] AXIS 1 (  5)   10   10   10   10
[t80040000] [t40043] AXIS 1 (  6)   10   10   10   10
[t80040000] [t40043] AXIS 1 (  7)   10   10   10   10
[t80040000] [t40043] AXIS 1 (  8)   10   10   10   10
[t80040000] [t40043] AXIS 1 (  9)   10   10   10   10
```

Graphically, the distribution of a P++ array is given below



8.2.5 Example 4. The Heat Equation

In this example we solve the non-dimensional heat equation $T_t = T_{xx}$ subject to two boundary conditions. T=2*x for $0 \leq x \leq .5$ and T=2(1-x) $.5 < x \leq 1$, where x is the spatial variable. The equation is solved with an explicit finite difference scheme. [G.D. Smith, Numerical Solution of Partial Differential Equation: Finite Difference Methods, Clarendon Press, 3rd Edition. pg 12].

```
// In this example we solve the heat equation.
// We will solve this problem with an explicit finite difference
// scheme.  See G.D. Smith pg 12.

#include <A++.h>
#include <time.h>
```

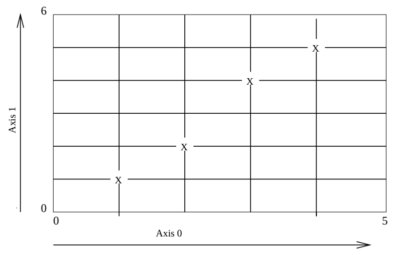
```
main(int argc,char** argv)
{
int num_of_process=5;
Optimization_Manager::Initialize_Virtual_Machine("",num_of_process,argc,argv);

// Length of physical dimensions and Length in time dimension //
double Length_x;
double Length_t;

// number of spaces in x and t
int spaces_in_x;
int spaces_in_t;

// spatial discretization
double dx;
// temporal discretization
double dt;


int i,j;
int time_step;

double time1;
double time2;
double total_time;
// r= dt/(dx^2)
double r;

// initialize variables
Length_x=1;
Length_t=1;
// change this line to increase spatial resolution
spaces_in_x=12;
spaces_in_t=1000;

dx=(Length_x/spaces_in_x);
dt=(Length_t/spaces_in_t);

r=dt/(dx*dx);
//------------------------------------------

Index I(1,spaces_in_x-1);

doubleArray Solution(spaces_in_x+1,spaces_in_x+1);
doubleArray temp(spaces_in_x+1,spaces_in_x+1);
// Initialize the
Solution=0.0;


// Setup boundary conditions //
// In this case we HAVE to use scalar index to setup the
//  boundary conditions
for  (i=1;i<=(int)(spaces_in_x/2);i++)
Solution(i,0)=2*i*dx;

for (i=(int)((spaces_in_x/2)+1);i<=spaces_in_x-1;i++)
Solution(i,0)=2*(1-i*dx);

Solution.display("initial and boundary conditions");

time1=clock();

// Notice that we are "mixing" the Index object I and normal scalar indexing
// in this finite difference "stencil"
for (int timestep=0;timestep<=8;timestep++){
       Solution(I,timestep+1)=r*(Solution(I+1,timestep)-2*Solution(I,timestep)+
        Solution(I-1,timestep))+Solution(I,timestep);
}

time2=clock();
total_time=time2-time1;

Solution.display("The Solution ");
printf("%f\n",total_time);
printf("program terminated properly");

Optimization_Manager::Exit_Virtual_Machine();
}
```

Output from Example 4

```
Initial Conditions
Array_Data is a VALID pointer = 84000 (540672)!
AXIS 0 --->: (   0) (   1) (   2) (   3) (   4) (   5) (   6) (   7) (   8) (   9) (  10)
AXIS 1 (  0) 0.0000 0.2000 0.4000 0.6000 0.8000 1.0000 0.8000 0.6000 0.4000 0.2000 0.0000
AXIS 1 (  1) 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
AXIS 1 (  2) 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
AXIS 1 (  3) 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
AXIS 1 (  4) 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
AXIS 1 (  5) 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
AXIS 1 (  6) 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
AXIS 1 (  7) 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
AXIS 1 (  8) 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
AXIS 1 (  9) 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
AXIS 1 ( 10) 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
doubleArray::display() (CONST) (Array_ID = 1) -- The Solution
SerialArray is a VALID pointer = 70000!

doubleSerialArray::display() (CONST) (Array_ID = 11) -- The Solution
Array_Data is a VALID pointer = a6000 (679936)!
AXIS 0 --->: (   0) (   1) (   2) (   3) (   4) (   5) (   6) (   7) (   8) (   9) (  10)
AXIS 1 (  0) 0.0000 0.2000 0.4000 0.6000 0.8000 1.0000 0.8000 0.6000 0.4000 0.2000 0.0000
AXIS 1 (  1) 0.0000 0.2000 0.4000 0.6000 0.8000 0.9600 0.8000 0.6000 0.4000 0.2000 0.0000
AXIS 1 (  2) 0.0000 0.2000 0.4000 0.6000 0.7960 0.9280 0.7960 0.6000 0.4000 0.2000 0.0000
AXIS 1 (  3) 0.0000 0.2000 0.4000 0.5996 0.7896 0.9016 0.7896 0.5996 0.4000 0.2000 0.0000
AXIS 1 (  4) 0.0000 0.2000 0.4000 0.5986 0.7818 0.8792 0.7818 0.5986 0.4000 0.2000 0.0000
AXIS 1 (  5) 0.0000 0.2000 0.3998 0.5971 0.7732 0.8597 0.7732 0.5971 0.3998 0.2000 0.0000
AXIS 1 (  6) 0.0000 0.2000 0.3996 0.5950 0.7643 0.8424 0.7643 0.5950 0.3996 0.2000 0.0000
AXIS 1 (  7) 0.0000 0.1999 0.3992 0.5924 0.7551 0.8268 0.7551 0.5924 0.3992 0.1999 0.0000
AXIS 1 (  8) 0.0000 0.1999 0.3986 0.5893 0.7460 0.8125 0.7460 0.5893 0.3986 0.1999 0.0000
AXIS 1 (  9) 0.0000 0.1998 0.3978 0.5859 0.7370 0.7992 0.7370 0.5859 0.3978 0.1998 0.0000
AXIS 1 ( 10) 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000

So after 8 timesteps  (.009 secs) the "1-d rod" has the following
temperature distribution

(   0) (   1) (   2) (   3) (   4) (   5) (   6) (   7) (   8) (   9) (  10)

0.0000 0.1998 0.3978 0.5859 0.7370 0.7992 0.7370 0.5859 0.3978 0.1998 0.0000
```

8.2.6 Example 5. Indirect Addressing

Indirect addressing allows the indexing of non-consecutive points in an array. For example suppose we wish to index the points in the figure below:



```
// This example illustrates the indirect addressing in A++/P++.
// Whereas the Index and Range object contain consecutive value
// (eg.Index I(0,N) == 0,1,..N-1).  Indirect addressing allows
// indexing of non-consective values.
//
//
```

```
#include <A++.h>

main(int argc,char** argv)
{
int num_of_process=3;
Optimization_Manager::Initialize_Virtual_Machine(" ",num_of_process,argc,argv);

  cout << "====== Test of A++ =====" << endl;

//  Index::setBoundsCheck(on);  //  Turn on A++ array bounds checking

  int n=6;
  int m;
  floatArray a(n,n), b(n,n), c(n,n);
  a=999.;
  b=0.;
  c=999.;

// number of points to index
  m=4;

// create two 1-d intArrays
  intArray i1(m), i2(m);

// Assign values to the intArrays
// We could also read in values from a file
  for( int i=0; i<=1; i++ )
  {
    i1(i)= (i+1)   % n;
    i2(i)= (i+1) % n;
  }

  for ( i=2;i<=3;i++)
  {
     i1(i)=(i+1);
     i2(i)=(i+2);
  }
  i1.display("Here is i1");
  i2.display("Here is i2");

// now we can either assign values to these points
// or read their values
        a(i1,i2)=6;
        a.display("here is a*");
        b(i1,i2)=c(i1,i2);
        b.display("here is b");

}
```

Output from Example 5

```
floatArray::display() (CONST) (Array_ID = 1) -- here is a*
Array_Data is a VALID pointer = 3c000 (245760)!
AXIS 0 --->: (   0) (   1) (   2) (   3) (   4) (   5)
AXIS 1 (  0) 999.0000 999.0000 999.0000 999.0000 999.0000 999.0000
AXIS 1 (  1) 999.0000 6.0000 999.0000 999.0000 999.0000 999.0000
AXIS 1 (  2) 999.0000 999.0000 6.0000 999.0000 999.0000 999.0000
AXIS 1 (  3) 999.0000 999.0000 999.0000 999.0000 999.0000 999.0000
AXIS 1 (  4) 999.0000 999.0000 999.0000 6.0000 999.0000 999.0000
AXIS 1 (  5) 999.0000 999.0000 999.0000 999.0000 6.0000 999.0000
floatArray::display() (CONST) (Array_ID = 2) -- here is b
Array_Data is a VALID pointer = 3e000 (253952)!
AXIS 0 --->: (   0) (   1) (   2) (   3) (   4) (   5)
AXIS 1 (  0) 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
AXIS 1 (  1) 0.0000 999.0000 0.0000 0.0000 0.0000 0.0000
AXIS 1 (  2) 0.0000 0.0000 999.0000 0.0000 0.0000 0.0000
AXIS 1 (  3) 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
AXIS 1 (  4) 0.0000 0.0000 0.0000 999.0000 0.0000 0.0000
AXIS 1 (  5) 0.0000 0.0000 0.0000 0.0000 999.0000 0.0000
```

8.2.7 Example 6. Application of Indirect Addressing

This example calculates the jacobian of a finite element (an important step, which maps the local finite element to the super element). The program uses indirect addressing to get the x and y coordinates of element, but actually calculate the jacobian in a series of FORTRAN subroutines.

```
#include <A++.h>
#include <math.h>
#include <fstream.h>
#include <stdlib.h>
#include <stdio.h>
//
// Application of indirect addressing to FEM
// jacobian of element.

// This allows us to call the FORTRAN subroutine test on a Sun Ultra
// The C++/FORTRAN interface is compiler and hardware specific.
//
extern "C" void test_(double*,double*,double*);
main()
{
intArray MeshPts(10,4);
doubleArray Global(12,3);
```

```
char* filename_mesh="meshdata";
char* globalpts="globalfile";
int pt[4];
char buf[80];
int i,j,x;
int element;
// sample data file
// element number       global nodal pts
//   1   2 3 4 5
//   2   4 5 7 9
//   3   9 5 6 3

ifstream fin(filename_mesh);

while(fin.getline(buf,80) !=0){
      (void) sscanf(buf,"%i  %i %i %i %i\n", &element, &pt[0],&pt[1],&pt[2],&pt[3]);
        for (i=0;i<=3;i++)
           MeshPts(element,i)=pt[i];
}
fin.close();
MeshPts.display();

// Global(nodal pts,[0:1]) == Cartesian Global Coordinates
// eg.  For nodal pts 1, the
// Global(1,0)=0.0   x coordinates of nodal pt 1
// Global(1,1)=1.0   y coordinates of nodal pt 1

// Read data file into MeshPts array
// global nodal pt x-coor y-coor
// 1   0.0  0.0
// 2   1.0  2.0
// 3   3.0  4.0
// 4   1.0  3.13
// 5   0.0  2.35
// 6   3.34  3.56
// 7   29.38  393.0
// 8   2.3  23.3
// 9   10.23  1.29
int nodal_pt;
float value[2];
ifstream fin2(globalpts);
// read in the datafile
while(fin2.getline(buf,80) !=0){
       (void) sscanf(buf,"%i  %f %f\n",&nodal_pt, &value[0], &value[1]);
        Global(nodal_pt,0)=(double)value[0];
        Global(nodal_pt,1)=(double)value[1];
```

```
   }

//Global.display();
fin2.close();

// The use indirect addressing to find the x and y coordinates of each element
// ptsx(2)=Global(MeshPts(1,2),0) =x coordinate of nodal pt 3
//===============================================================
Range I(0,3);
       intArray tempArray(1,6);
       doubleArray ptsx(6);
       doubleArray ptsy(6);
//  initialize variables
     tempArray=0;
     ptsx=0.0;
     ptsy=0.0;

//  The element we want to find the X and Y coordinates
     int element_number=1;

//   read the global pts into an intArray
//
MeshPts(1,I).display("meshpts");
tempArray(I)=MeshPts(element_number,I);


//    use indirect addressing to
//    get the x and y coordinates of the  element
ptsx=Global(tempArray,0);
ptsy=Global(tempArray,1);

ptsx.display();
ptsy.display();

// now lets calculate the jacobian for the points (ptsx and ptsy)
// Since there is FORTRAN code to do this
// We just call the it subroutine  from A++.
//
doubleArray jacob(2,2);
// change the base to work more easily  with FORTRAN
jacob.setBase(1);

//  The Fortran Subroutine
test_(ptsx.getDataPointer(),ptsy.getDataPointer(),jacob.getDataPointer());

jacob.display();
```

```
}
```

The Fortran Subroutines

```
subroutine test(a,b,jacob)

C  Use Real*8 passing <type> double
real*8  a(5),b(5)
real*8  gpt(3), gwt(3)
real*8  r,s
real*8 nvect(10)
real*8  dnrvect(10),dnsvect(10)
real*8   jacob(2,2)
gpt(1)=-.5777
gpt(2)=.5777
gwt(1)=1.0
gwt(2)=1.3


do 10 j=1,2
   do 20 i=1,2
      r=gpt(i)
      s=gpt(j)

       call nvec(r,s,nvect)
       call dnrvec(r,s,dnrvect)
       call dnsvec(r,s,dnsvect)


c
        jacob(1,1)=vectmult(dnrvect,a)
jacob(1,2)=vectmult(dnrvect,b)
jacob(2,1)=vectmult(dnsvect,a)
jacob(2,2)=vectmult(dnsvect,b)


20 continue
10 continue

end



function vectmult(a,b)
real*8  a(10)
real*8  b(10)
real*8  temp,vectmult
real*8  temp2

temp=0
do 4 i=1,4
temp2=a(i)*b(i)
temp=temp2+temp
4 continue
vectmult=temp
```

```
return
end



         subroutine nvec(r,s,nvect)
                 real*8   r,s,nvect(10)
                 integer i,j,k
                        do 10 i=1,10
                             nvect(i)=0.
10                      continue
                 nvect(1)=.25*(1-r)*(1-s)
                 nvect(2)=.25*(1+r)*(1-s)
                 nvect(3)=.25*(1+r)*(1+s)
                 nvect(4)=.25*(1-r)*(1+s)
                 return
                         end


subroutine dnsvec(r,s,dnsvect)
real*8  r,s,dnsvect(10)
integer i,j,k
do 10 i=1,10
dnsvect(i)=0.0
10 continue
dnsvect(1)=-.25*(1-r)
dnsvect(2)=-.25*(1+r)
dnsvect(3)=.25*(1+r)
dnsvect(4)=.25*(1-r)
return
end


subroutine dnrvec(r,s,dnrvect)
real*8 s,r,dnrvect(10)
integer i,j,k
do 10 i=1,10
dnrvect(i)=0.0
10 continue

dnrvect(1)=-.25*(1-s)
                 dnrvect(2)=.25*(1-s)
                 dnrvect(3)=.25*(1+s)
                 dnrvect(4)=-.25*(1+s)
return
end
```

Output from Example

This interfacing with FORTRAN is important, because it opens the possibility of using A++/P++ with a number of scientific library (eg. LAPACK, SLATEC,etc).

8.3 Example Makefile

This example makefile shows the use of a single A++/P++ source code which is compiled with A++ to build the A++ application and uses P++ to build the P++ application. The source code is unchanged and used to build both A++ and P++ application codes. While the makefile itself is somewhat complicated, this demonstrates how a single code written for A++ can be reused to build the equivalent P++ (parallel) application.

```
# The following may be changed by the user
# This works for programs in the APPLICATIONS directory
# change ARCH to match the architecture chosen during configuration (installation)
of A++/P++
ARCH = SUN4

# NOTE: APP_HOME must be a absolute path to work with some compilers
APP_HOME    = ../A++
APP_INCLUDE = $(APP_HOME)/include
APPLIB_DIR  = $(APP_HOME)/$(ARCH)

# NOTE: PPP_HOME must be a absolute path to work with some compilers
PPP_HOME    = ../P++
PPP_INCLUDE = $(PPP_HOME)/include
PPPLIB_DIR  = $(PPP_HOME)/$(ARCH)

# This is where PVM lives at Los Alamos
PVMLIB = /usr/lanl/pvm/lib/SUN4/libgpvm3.a /usr/lanl/pvm/lib/SUN4/libpvm3.a

CC_Compiler = CC

# *****************************************************
# You should not have to change anything below this line
# *****************************************************

all: riemann p++_riemann mg p++_mg array_test p++_array_test adaptive p++_adaptive

.SUFFIXES: .c .C .cc .o .cxx .a .o .cpp


# ***************************************************************************
# Example rule for building A++ versions of codes below
# ***************************************************************************
.C.o :
$(CC_Compiler) -I$(APP_INCLUDE) $(CC_FLAGS) -c $*.C

# ***************************************************************************
# Test program to test random features of A++
# ***************************************************************************
array_test : array_test.o
$(CC_Compiler) $(CC_FLAGS) -o array_test array_test.o -L$(APPLIB_DIR)
-lA++ -lm

# This should show how lines which use A++ source build either a serial
#(A++) or parallel (P++) application
p++_array_test : array_test.C
```

```
$(CC_Compiler) $(CC_FLAGS) -c -I$(PPP_INCLUDE) -o p++_array_test.o
array_test.C
$(CC_Compiler) $(CC_FLAGS) -o p++_array_test p++_array_test.o
-L$(PPPLIB_DIR) -lP++ $(PVMLIB) -lm

# ***************************************************************************
# Riemann solver
# ***************************************************************************
riemann : riemann.o
$(CC_Compiler) $(CC_FLAGS) -o riemann riemann.o -L$(APPLIB_DIR) -lA++ -lm

# This should show how lines which use A++ source build either a serial
#(A++) or parallel (P++) application
p++_riemann : riemann.C
$(CC_Compiler) $(CC_FLAGS) -c -I$(PPP_INCLUDE) -o p++_riemann.o riemann.C
$(CC_Compiler) $(CC_FLAGS) -o p++_riemann p++_riemann.o -L$(PPPLIB_DIR)
-lP++ $(PVMLIB) -lm

# ***************************************************************************
# Simulation of an adaptive solver using deferred evaluation and task recognition
# ***************************************************************************
adaptive : adaptive.o
$(CC_Compiler) $(CC_FLAGS) -o adaptive adaptive.o -L$(APPLIB_DIR) -lA++
-lm

# This should show how lines which use A++ source build either a serial
#(A++) or parallel (P++) application
p++_adaptive : adaptive.C
$(CC_Compiler) $(CC_FLAGS) -c -I$(PPP_INCLUDE) -o p++_adaptive.o
adaptive.C
$(CC_Compiler) $(CC_FLAGS) -o p++_adaptive p++_adaptive.o -L$(PPPLIB_DIR)
-lP++ $(PVMLIB) -lm

# ***************************************************************************
# Multigrid example for 1-3D problems!
# ***************************************************************************
mg: mg.o mg1level.o pde.o mg_main.o
$(CC_Compiler) $(CC_FLAGS) -o mg mg.o mg1level.o pde.o mg_main.o
-L$(APPLIB_DIR) -lA++ -lm

# This should show how lines which use A++ source build either a serial
#(A++) or parallel (P++) application
p++_mg : pde.C mg_main.C mg.C mg1level.C
$(CC_Compiler) $(CC_FLAGS) -c -I$(PPP_INCLUDE) -o p++_pde.o pde.C
$(CC_Compiler) $(CC_FLAGS) -c -I$(PPP_INCLUDE) -o p++_mg_main.o mg_main.C
$(CC_Compiler) $(CC_FLAGS) -c -I$(PPP_INCLUDE) -o p++_mg.o mg.C
$(CC_Compiler) $(CC_FLAGS) -c -I$(PPP_INCLUDE) -o p++_mg1level.o
mg1level.C
$(CC_Compiler) $(CC_FLAGS) -o p++_mg p++_pde.o p++_mg_main.o p++_mg.o
p++_mg1level.o -L$(PPPLIB_DIR) -lP++ $(PVMLIB) -lm

# Similar Multigrid code in C
mg_c: mg_c.c
$(C_Compiler) mg_c.c -o mg_c -lm

clean:
rm -f array_test riemann adaptive mg mg_c *.o core
rm -f p++_array_test p++_riemann p++_adaptive p++_mg mg_c *.o core
```

8.4 More example on the A++/P++ Home Page

A++/P++ has a WWW Home Page which contains more, longer, and more meaningful examples of A++/P++ programs. The URL for the A++/P++ Home Page is: **http://www.c3.lanl.gov/˜dquinlan/A++P++.html**. This site is updated regularly with the newest documentation.

Chapter 9

Examples: Code Fragments

This is a collection of example A++/P++ code fragments. It is intended to show some of the many ways that A++/P++ can be used. There are two sections, one on A++/P++ examples and the scond on P++ specific examples that demonstrate parallel features of P++.

9.1 A++/P++ Examples

These examples are common to both A++ and P++ array classes. They show a complex mix of operations taken from many A++/P++ codes.

```
#define BOUNDS_CHECK
#include "A++.h"

void main ()
   {
     int Array_Size = 100;

  // Index Constructor examples
     Index I ( 1 , Array_Size-2, 1 );  // position=1, count=Array_Size-2, stride=1
     Index J = I;                      // make an Index object J just like  I
     Index K = I-1;                    // make an Index object K just like I-1
     Index L = -I;                     // make L like I but with negative stride
     Index M = 5;                      // make Index object from integer index
     Index N;                          // build an uninitialized Index object
     N = I+1;                          // Index assignment to build N like offset of I

  // Array Constructor examples
     doubleArray A1 (Array_Size);
     floatArray B1 (Array_Size,Array_Size);
     doubleArray C1 (Array_Size,Array_Size,Array_Size);
     intArray D1 (Array_Size,Array_Size,Array_Size,Array_Size);
     floatArray E1 = B1;
     doubleArray F1 = B1(I-1,J);

     double *Fortran_Array_Pointer = new double [Array_Size+1][Array_Size];
     doubleArray G (Fortran_Array_Pointer,Array_Size,Array_Size+1);
```

```
 // Arrays for use in examples below
    doubleArray A (Array_Size,Array_Size);
    doubleArray B (Array_Size,Array_Size);
    doubleArray C (Array_Size,Array_Size);
    doubleArray D (Array_Size,Array_Size);
    double x = 42;

 // example of array-scalar assignment
    A = x;
    A (I) = x;
    A (I-1) = x * x;

// examples of array-array assignment operations and use of Index objects
   B = A;
   B = C = D = A;
   A (I,J) = B (J,J);
   A (I-1,J) = B (I+1,J);

// Scalar indexing
   A (0,12) = x;
   A (5,12) = A (0,12);
   x = A (1,12) + B(0,12);

// examples of array-array arithmitic operations
   A = B + (C * B - D) / A;
   A (I,J) += B (I,J) / C (I,J);
   A (I-1,J) *= B (I+1,J);

// examples of Jacobi relaxation (9-point stencil)
   A (I,J) = ( A (I+1,J+1) + A (I,J+1  ) + A (I-1,J+1) + A (I+1,J  ) +
               A (I-1,J  ) + A (I+1,J-1) + A (I,J-1  ) + A (I-1,J-1) ) / 8.0;

// examples of Jacobi relaxation (5-point stencil)
   A (I,J) = ( A (I,J+1) + A (I,J-1) + A (I+1,J) + A (I-1,J) ) / 4.0;

// more complex operations
   B (I,J) = ( A (I-1,J-1) * B (I+1,J+1) + C (I-1,J) * D (I,J+1) -
               D (I,J) * B (I,J) * ( A (I,J) - B (I,J) ) ) / ( C (I,J) + D (I,J) );

// examples of relational operator
   intArray Mask = B >= C;
   Mask = !B;
   Mask = !(B && C) != (!B | !C); // DeMorgan's Law

// example of replace operator
   A (I,J).replace ( B (I,J) <= 0.001 , 0.001 );
   A (I,J).replace ( A (I,J) <= C(I,J) , C(I,J) );

// simple example of "where" statement
   where ( B >= C)
        A = 0.001;

// more complex example of "where" used for multiple statement block
   where ( B(I,J) >= C(I,J) )
      {
        A(I,J) = ( A (I,J+1) + A (I,J-1) + A (I+1,J) + A (I-1,J) ) / 4.0;
        B(I,J) = 0.001;
        C(I,J) = 0.001;
      }

 // examples of max function use
    x = max (B);
    A = max (B , C * B);
    A = max (B , C , A);

 // examples of miscellaneous function use
    x = sum (B);
    A = cos (B) * sqrt (C);
    B(I,J) = (cos (B) * 2.0 )(I,J);

 // examples of changing bases of array objects
    A.setBase (1);     // Force A to have indexing similar to Fortran array
    setGlobalBase (1); // Set all future arrays to have Fortran like base of 1
    A.setBase (x);
    A.setBase(x) = B;  // Shows value returned from setBase
    A.setBase (x,0);
    A.setBase (x*x,1);

 // examples of bases and bound access
    Array_Size = A.dimension(0);
    printf ("Number of elements in A = %d \n",A.elementCount();
    for (int j = A.getBase(1); j <= A.getBound(1); j++)
         for (int i = A.getBase(0); i <= A.getBound(0); i++)
              A(i,j) = foo (i,j);

 // examples of display functions
    A (I,J).display("This is A (I,J)");
    A = B + (C * D).display("This is C * D in expression A = B + (C * D)");
    (A = B * D).view("This is A = B * D");
    A.view("This is A (same view as above)");

 // 2 ways to pass array objects by reference
    void foo ( const doubleArray & X );
    foo ( evaluate (A + B) );

    C = A + B;
    foo ( C );

 // passing array objects by value requires no special handling
    void foobar ( const doubleArray X );
    foobar ( A + B );

 // examples of fill functions
    A(I,J).fill(x);

    printf ("PROGRAM TERMINATED NORMALLY \n");
   }
```

9.2 P++ Specific Examples

This section presents some examples that are specific to parallel P++ operations. These example deal directly with the distributions of array objects onto the multiple processors available within the parallel environment.

```
#define BOUNDS_CHECK
#include <A++.h>

int main(int argc, char** argv )
   {
     Index::setBoundsCheck (On);
     int numberOfProcessors = 128;
  // P++ looks for the application name if "" is specified
     Optimization_Manager::Initialize_Virtual_Machine("",
             numberOfProcessors,argc,argv);

  // Example of using a partition object (assume number of procesors is >= 64)
     Partition_Type Partition_A (64);   // Build partition object which uses processors 0-63
     floatArray A(100,100,Partition_A); // Build array using default "block-block" distribution
                                        // across the processors represented by Partition_A.

  // Example of distribution onto subrange of processors
     Range ProcessorSubrange_B (27,37);
     Partition_Type Partition_B (ProcessorSubrange_B); // Build partition object which uses processors 27-3
     floatArray B(100,100,Partition_B);                // Build an array distributed "block-block"over
                                                       // processors 27-37

  // Simple example of alignment specification
     Range all;              // Default range object implies "all" of wherever it is used
     floatArray C (100,100);  // Build array "block-block" over all processors
     floatArray D = C (0,all); // Align D with boundary of C

  // Simple example of array redistribution
     Range all;                                        // Default range object implies "all" of wherever it
     Range ProcessorSubrange_E (45,83);
     Partition_Type Partition_E (ProcessorSubrange_E); // Build partition object which uses processors 45-8
     floatArray E (100,100,Partition_E);               // Build array "block-block" over processors 45-83
     floatArray F = E (0,all);                         // Align F with boundary of E
     Partition_E.SpecifyProcessorRange(Range(2-12));   // Redistribute E and F on to processors 2-12
                                                       // note that F is STILL aligned with the boundary of

  // More complex redistribution example.  This example builds a collection
  // of different sized arrays each associated with the same partitioning object.
  // then the arrays are all repartitioned through simple manipulation of the
  // partition object.  the arrays are initially distributed onto processor 0,
  // then on an increassing number of processors until all processor are used,
  // then repartitioned onto a decreassing number of processors untill finally
  // distributed only on processor zero.
     int Size = 10;
     Partitioning_Type Partition (Range(0,0));
     doubleArray Temp_A(Size,Partition);
     doubleArray Temp_B(Size*2,Partition);
     doubleArray Temp_C(Size/2,Partition);
     doubleArray Temp_D(Size*2,Partition);
     doubleArray Temp_E(Size/2,Partition);
     doubleArray Temp_F(Size,Partition);
     int i;
     for (i=0; i < Communication_Manager::Number_Of_Processors; i++)
          Partition.SpecifyProcessorRange (Range(0,i));  // redistribute all arrays associated with "Partition"
     for (i=0; i < Communication_Manager::Number_Of_Processors; i++)
          Partition.SpecifyProcessorRange (Range(i,numberOfProcessors-1));

  // Example using scalar indexing on local part of distributed array
     intArray v(100);
     int ibas = v.getLocalBase(0);
     int ibnd = v.getLocalBound(0);
     Optimization_Manager::setOptimizedScalarIndexing (On);
     for (int i=ibas; i<=ibnd; i++)
          v(i) = i;
     Optimization_Manager::setOptimizedScalarIndexing (Off);

  // Example of getting local A++ array within P++ distributed array
     floatArray X (100,100);  // distributed array
     floatSerialArray X_local = X.getLocalArray();  // Deep copy of local data
     floatSerialArray X_local (X.getLocalArray(),SHALLOW_COPY);  // Shallow copy of local data
     floatSerialArray X_local (X.getLocalArray(),DEEP_COPY);  // Deep copy of local data
     floatSerialArray *X_pointer_to_local = X.getSerialArrayPointer(); // pointer to local data
```

Chapter 10

Reference

10.1 Legend

type	**double**, **float**, or **int**

Variables used in examples below

i,j,k,l	integers used as scalar index variables
Span_I,Span_J,Span_K,Span_L	objects of type **Range**
I,J,K,L	objects of type **Index**
List_I,List_J,List_K,List_L	objects of type **intArray**
A,B,C	*type***Array** variables
Mask	an **intArray** variable
n,m,o,p	any positive integer
Fortran_Array_Pointer	pointer to a Fortran array
x	variable of *type*
axis	dimension 0-3 of the 4D *type***Array**

10.2 Debugging A++P++ Code

10.2.1 Turning On Bounds Checking

Bounds Checking in A++P++ must be turned on and is OFF by default.

Turning On Bounds Checking For All But Scalar Indexing

Bounds checking in A++P++ must be turned on and is OFF by default.

Index::setBoundsCheck (On);	Turns ON array bounds checking!
Index::setBoundsCheck (Off);	Turns OFF array bounds checking!

Turning On Bounds Checking For Scalar Indexing

Scalar bounds checking in A++P++ must be set at compile time. Bounds checking is OFF by default. It may be set on the compile command line or at the top of each program file (**before** #include<A++.h>).

CC -DBOUNDS_CHECK *other options*	Turns on scalar index bounds checking.
#define BOUNDS_CHECK	Turns on scalar index bounds checking in file.

10.2.2 Using dbx with A++

dbx supports calling functions and with the correct version of dbx that understands C++ name mangling, member functions of the A++ array objects may be called with the following example syntax:

call A.display()	dbx calls the display member function for an A++P++ array A

10.2.3 Mixing C++ streams and C printf

Mixing of C++ "cout <<" like I/O syntax with C stype "printf" I/O syntax will generate strange behavior in the ordering of the user's I/O messages. To fix this insert the following call to the I/O Streams library of C++ at the start of your main program.

ios::sync_with_stdio();	Synchronize C++ and C I/O subsystems!

10.3 Range Objects

10.3.1 Constructors

Note: The base must be less than or equal to the bound to define a valid span of an array, if base > bound then the range is considered null.

Range Span_K (±base,±bound),±stride);	Range object Span_K from base, to bound, by stride
Range Span_I;	Range object which is null
Range Span_J = Span_I;	Span_J is a copy of Span_I (not an alias)

10.3.2 Operators

Span_J = Span_I;	assignment operator
Span_I+n;	builds new Range object with position of Span_I + n
n+Span_I;	builds new Range object with position of Span_I + n
Span_I-n;	builds new Range object with position of Span_I - n
n-Span_I;	builds new Range object with position of Span_I - n

10.3.3 Access Functions

Span_I.getBase();	returns base of Span_I
Span_I.getBound();	returns bound of Span_I
Span_I.getStride();	returns bound of Span_I
Span_I.length();	returns (bound-base)+1 for Span_I

10.4 Index Objects

10.4.1 Constructors

The stride in the examples below default to 1 (unit stride) if not specified. That we provide an Index constructor which takes a Range object allows Range objects to be used where ever Index objects are used (e.g. indexing operators).

Index K (±position,count);	Index object K references from position, for count elements, with default stride = 1
Index K (±position,count,stride);	Index object K references from position, for count elements, with stride
Index I;	Index which references all of any array object
Index I(±i);	Index with position=±i, count=1, stride=1
Index J = I;	J is a copy of I (not an alias)
Index K = Span_I;	Index K is built from a Range object, Span_K

10.4.2 Operators

I+n;	new Index with position of Index I + n
n+I;	new Index with position of Index I + n
I-n;	new Index with position of Index I - n
n-I;	new Index with position of Index I - n
J = I;	assignment operator

10.4.3 Access Functions

I.getBase();	returns base of I
I.getBound();	returns bound of I
I.getStride();	returns stride of I
I.length();	returns length of I (accounting for stride)

10.4.4 Display Functions

I.display("label");	Prints Index values and all other internal data for I along with character string "label" to sdtout

10.5 Array Objects

10.5.1 Constructors

A++ arrays are replicated on each processor in P++, while P++ arrays are distributable across processors using user defined distributions (not covered here). Note that the Range objects can be used to build an A++ array, if used, they define the size *and* the base of the array from the Range object provided for each dimension.

*type***Array** A;	array object A (zero length array)
*type***Array** B = A;	array B as a copy of A
*type***Array** C (n);	1D array C of length n
*type***Array** C (n,m);	2D array C of length n × m

*type***Array** C (n,m,o);	3D array C of length n $\times$ m $\times$ o
*type***Array** C (n,m,o,p);	4D array C of length n $\times$ m $\times$ o $\times$ p
*type***Array** C (Span_I);	1D array C of length of Span_I
*type***Array** C (Span_I,Span_J);	2D array C of length of Span_I $\times$ Span_J
*type***Array** C (Span_I,Span_J,Span_K);	3D array C of length of Span_I $\times$ Span_J $\times$ Span_K
*type***Array** C (Span_I,Span_J,Span_K,Span_L);	4D array C of length of Span_I $\times$ Span_J $\times$ Span_K $\times$ Span_L

A++ only

*type***Array** C (Fortran_Array_Pointer, n);	1D array C of length n using existing array
*type***Array** C (Fortran_Array_Pointer, n,m);	2D array C of length n $\times$ m using existing array
*type***Array** C (Fortran_Array_Pointer, n,m,o);	3D array C of length n $\times$ m $\times$ o using existing array
*type***Array** C (Fortran_Array_Pointer, n,m,o,p);	4D array C of length n $\times$ m $\times$ o $\times$ p using existing array

*type***Array** C (Fortran_Array_Pointer, Span_I);	1D array C using existing data
*type***Array** C (Fortran_Array_Pointer, Span_I,Span_J);	2D array C using existing data
*type***Array** C (Fortran_Array_Pointer, Span_I,Span_J,Span_K);	3D array C using existing data
*type***Array** C (Fortran_Array_Pointer, Span_I,Span_J,Span_K,Span_L);	4D array C using existing data

P++ only

*type***Array** C (Fortran_Array_Pointer, n, Local_Size_n);	1D array C of length n using existing array
*type***Array** C (Fortran_Array_Pointer, m, Local_Size_m, n, Local_Size_n);	2D array C of length n $\times$ m using existing array
*type***Array** C (Fortran_Array_Pointer, m, Local_Size_m, n, Local_Size_n, o, Local_Size_o);	3D array C of length n $\times$ m $\times$ o using existing array
*type***Array** C (Fortran_Array_Pointer, m, Local_Size_m, n, Local_Size_n, o, Local_Size_o, p, Local_Size_p);	4D array C of length n $\times$ m $\times$ o $\times$ p using existing array

P++ only

*type***Array** C (n, Partition);	Use existing Partitioning_Type
*type***Array** C (m, n, Partition);	Use existing Partitioning_Type
*type***Array** C (m, n, o, Partition);	Use existing Partitioning_Type
*type***Array** C (m, n, o, p, Partition);	Use existing Partitioning_Type

10.5.2 Assignment Operators

A(I,J) = B(I-1,J+1);	Set elements of A equal to elements of B
A = x;	Set elements of A equal to x

10.5.3 Indexing Operators

Note that indexing support for Range objects is available because Index objects are constructed from the Range objects and the resulting Index object is used.

Indexing operators for scalar indexing: denotes a scalar

A(i)	Scalar indexing of a 1D array object

A(i,j)	Scalar indexing of a 2D array object
A(i,j,k)	Scalar indexing of a 3D array object
A(i,j,k,l)	Scalar indexing of a 4D array object

Indexing operators for use with Index objects: denotes a *type***Array**

A(I)	Index object indexing of a 1D array object
A(I,J)	Index object indexing of a 2D array object
A(I,J,K)	Index object indexing of a 3D array object
A(I,J,K,L)	Index object indexing of a 4D array object

Indexing operators for use with Range objects: denotes a *type***Array**

A(Span_I)	Range object indexing of a 1D array object
A(Span_I,Span_J)	Range object indexing of a 2D array object
A(Span_I,Span_J,Span_K)	Range object indexing of a 3D array object
A(Span_I,Span_J,Span_K,Span_L)	Range object indexing of a 4D array object

Indexing operators for use with intArray objects: denotes a *type***Array**

A(List_I)	intArray object indexing of a 1D array object
A(List_I,List_J)	intArray object indexing of a 2D array object
A(List_I,List_J,List_K)	intArray object indexing of a 3D array object
A(List_I,List_J,List_K,List_L)	intArray object indexing of a 4D array object

10.5.4 Indirect Addressing

The subsection **Indexing Operators** (above) presents the use of intArrays to index A++ arrays (even other intArray objects). The value of the elements of the intArray are used to define the relevant elements of the indexed object (view). It is often required to convert between a mask returned by an relational operator and an intArray whose values represent the non-zero index positions in the mask, however this conversion of a mask to an intArray is currently supported only for 1D.

intArray Indirect_Address = Mask.indexMap()	builds intArray object with values of non-zero index position in Mask
intArray I = (A == 5).indexMap()	builds intArray I as a mapping (into A) of elements in A equal to 5

10.5.5 Arithmetic Operators

All arithmetic operators return a *type***Array** consistent with their input, no mixed type operations are allowed presently. Casting operators will be added soon to permit mixed operations. All operations are performed elementwise and the result returned in a separate *type***Array** (unless one of the operands is a result from a previous expression in which case the temporary operand is reused internally).

B + C;	Add B and C
B + x;	Add B and x
x + C;	Add x and C
B += C;	Add C to B store result in B
B += x;	Add x to B store result in B
B - C;	Subtract C from B
B - x;	Subtract x from B
x - C;	Subtract C from x

B -= C;	Subtract C from B store result in B
B -= x;	Subtract x from B store result in B
B * C;	Multiply B and C
B * x;	Multiply B and x
x * C;	Multiply x and C
B *= C;	Multiply C and B store result in B
B *= x;	Multiply x and B store result in B
B / C;	Divide B by C
B / x;	Divide B by x
x / C;	Divide x by C
B /= C;	Divide B by C store result in B
B /= x;	Divide B by x store result in B
B % C;	B Modulo C
B % x;	B Modulo x
x % C;	x Modulo C
B %= C;	B Modulo C to store result in B
B %= x;	B Modulo x store result in B

10.5.6 Relational Operators

All relational operators return an **intArray**, no mixed type operations are allowed presently. All operations are performed elementwise and return conformable mask (**intArray** object). Mask values are zero if the conditional test was false, and non-zero if operation was true. See **Indirect Addressing** for conversion of zero/non-zero masks into intArrays for use with indirect address indexing.

!B;	mask based on test for zero elements of B
B < C;	mask specifying elements of B < C
B < x;	mask specifying elements of B < x
x < C;	mask specifying elements of C where x < C
B <= C;	mask specifying elements of B <= C
B <= x;	mask specifying elements of B <= x
x <= C;	mask specifying elements of C where x <= C
B > C;	mask specifying elements of B > C
B > x;	mask specifying elements of B > x
x > C;	mask specifying elements of C where x > C
B >= C;	mask specifying elements of B >= C
B >= x;	mask specifying elements of B >= x
x >= C;	mask specifying elements of C where x >= C
B == C;	mask specifying elements of B == C
B == x;	mask specifying elements of B == x
x == C;	mask specifying elements of C where x == C
B ! = C;	mask specifying elements of B ! = C
B ! = x;	mask specifying elements of B ! = x
x ! = C;	mask specifying elements of C where x ! = C
B && C;	mask specifying elements of B && C
B && x;	mask specifying elements of B && x

x && C;	mask specifying elements of C where x && C
B || C;	mask specifying elements of B || C
B || x;	mask specifying elements of B || x
x || C;	mask specifying elements of C where x || C

10.5.7 Min Max functions

These functions (except in the case of the single input reduction operations) return array objects with an elementwise interpretation. Both "min" and "max" represent reduction operations in the case of a single array input. These functions thus return a scalar value from the array input. In A++ the operation is straightforward. In P++ the reduction operators return a scalar, but internally do the required message passing to force the same scalar return value on all processors (assuming a data parallel model of execution).

min (A);	return scalar minimum of all array elements
min (B,C);	min elements of B and C
min (B,x);	min elements of B and x
min (x,C);	min elements of x and C
min (A,B,C);	min elements of A,B and C
min (x,B,C);	min elements of x,B and C
min (A,x,C);	min elements of A,x and C
min (A,B,x);	min elements of A,B and x
max (A);	return scalar maximum of all array elements
max (B,C);	max elements of B and C
max (B,x);	max elements of B and x
max (x,C);	max elements of x and C
max (A,B,C);	max elements of A,B and C
max (x,B,C);	max elements of x,B and C
max (A,x,C);	max elements of A,x and C
max (A,B,x);	max elements of A,B and x

10.5.8 Miscellaneous Functions

All functions return a *type***Array** consistent with their input, no mixed type operations are allowed presently. Functions fmod and mod apply to double or float arrays and integer arrays, respectively. Functions log, log10, exp, sqrt, fabs, ceil, floor, cos, sin, tan, acos, asin, atan, atan2, cosh, sinh, tanh, acosh, asinh, atanh; only apply to **doubleArray** and **floatArray** objects. Function abs applies to only **intArray** objects.

For P++ operation of reduction functions ("sum," for example) see note on reduction operators in P++ in previous subsection (Min Max functions).

fmod (B,C);	B modulo C equivalent to operator B % C
fmod (B,x);	B modulo x equivalent to operator B % x
fmod (x,C);	x modulo C equivalent to operator x % C
mod (B,C);	B modulo C equivalent to operator B % C
mod (B,x);	B modulo C equivalent to operator B % x
mod (x,C);	B modulo C equivalent to operator x % C
pow (B,C);	$B(i)^{C(i)}$ for elements of B and C
pow (B,x);	$B(i)^{x}$ for elements of B and x

pow (x,C);	$x^{C(i)}$ for elements of x and C
sign (B,C);	C with sign of B
sign (B,x);	array with values of x but with sign of B
sign (x,C);	C with sign of x
sum (B);	sum of elements of B
log (B);	log of elements of B
log10 (B);	log10 of elements of B
exp (B);	exp of elements of B
sqrt (B);	sqrt of elements of B
fabs (B);	fabs of elements of B
ceil (B);	ceil of elements of B
floor (B);	floor of elements of B
abs (B);	abs of elements of B
cos (B);	cosine of elements of B
sin (B);	sine of elements of B
tan (B);	tangent of elements of B
acos (B);	arccosine of elements of B
asin (B);	arcsine of elements of B
atan (B);	arctangent of elements of B
atan2 (B,C);	arctangent of elements of B/C
cosh (B);	hyperbolic cosine of elements of B
sinh (B);	hyperbolic sine of elements of B
tanh (B);	hyperbolic tangent of elements of B
acosh (B);	arc hyperbolic cosine of elements of B
asinh (B);	arc hyperbolic sine of elements of B
atanh (B);	arc hyperbolic tangent of elements of B

10.5.9 Replace functions

Replacement of elements is done for non-zero mask elements. Mask and input arrays must be conformable. Since this feature of A++/P++ is redundent with the **where** statement functionality, the replace member function may be devalued at a later date and then removed from A++/P++ sometime after that.

A.replace (Mask , B);	replace elements in A with elements in B depending on value of Mask
A.replace (Mask , x);	replace elements in A with scalar x depending on value of Mask
A.replace (x , B);	replace elements in A with elements in B depending on value of x (equivalent to `if (x) A = B;`)

10.5.10 Array Type Conversion Functions

The conversion between array types is commonly represented by casting operators. However, such casting operators could be called as part of automate conversion which can be especially problematic to debug. To facilitate the conversion between types of arrays we provide member functions that cast an array of one type to an array of another type explicitly. These member functions can, for example, convert an array of type intArray to an array of type floatArray. Or we can convert a floatArray to an intArray. As and example, this mechanism simplifies the visualization of intArray objects using graphics functionality only written for floatArray or doubleArray types. Future work implement casting operators that make the conversion implicit.

A.convertTo_intArray();	return an intArray (convert *type***Array** A to an intArray

A.convertTo_floatArray();	return a floatArray (convert *type***Array** A to a floatArray
A.convertTo_doubleArray();	return a doubleArray (convert *type***Array** A to a doubleArray

10.5.11 User defined Bases

A++/P++ array object may have user defined bases in each array dimension. This allows for array objects to have a base of 1 (as in FORTRAN), or any other positive or negative value.

A.setBase(±n);	Set base to ±n along all axes of A
A.setBase(±n,axis);	Set base to ±n along axis of A
setGlobalBase(±n);	Set base to ±n along all axes for all future array objects
setGlobalBase(±n,axis);	Set base to ±n along axis for all future array objects

10.5.12 Indexing of Views

The base and bound of a view of an array object are dependent on the base and bound of the **Index** or **Range** object used to build the view. Thus a view, A(I), of an array, A, is another array object which carries with it the index space information about it's view of the subset of data in the original array, A.

10.5.13 Array Size functions

Array axis numbering starts at zero and ends with the max number of dimensions (a constant MAX_ARRAY_DIMENSION stores this value) for the A++/P++ array objects minus one. These provide access into the A++ objects and assume an A++ object is being used. An alternative method is defined to permit access to the same data if a raw pointer is being used, this later method is required if a pointer to the array data is being passed to FORTRAN. The access functions for this data have the names *getRawBase()*, *getRawBound()*, *getRawStride()*, *getRawDataSize()*.

A.getBase();	Get base along all axes of A (bases must be equal)
A.getBase(axis);	Get base along axis of A
A.getRawBase(axis);	Get base along axis of A
getGlobalBase();	Get base along all axes for all future array objects
getGlobalBase(axis);	Get base along axis for all future array objects
A.getStride(axis);	Get stride along axis of A
A.getRawStride(axis);	Get stride along axis of A
A.getBound(axis);	Get bound along axis of A
A.getRawBound(axis);	Get bound along axis of A
A.getLength(axis);	Get dimension (array size) of A along axis
A.getFullRange(axis);	return a Range object (base,bound,stride of the array)
A.dimension(axis);	Get dimension (array size) of A along axis (returns a Range object)
A.elementCount();	Get total array size of A
A.numberOfDimensions();	Get total number of dimensions of A
A.isAView();	returns TRUE if A is a subArray (view) of another array object
A.isNullArray();	returns TRUE if A is an array of size zero
A.isTemporary();	returns TRUE if A is a result of an expression
A.rows();	Get number of rows of A (for 2D array objects)
A.cols();	Get number of cols of A (for 2D array objects)

10.5.14 Array Object Similarity test functions

Array axis numbering starts at zero and ends with the max number of dimensions (a constant MAX_ARRAY_DIMENSION stores this value) for the A++/P++ array objects minus one. These member functions allow for the testing of Bases, Bounds, Strides, etc along each axis for two array objects. For example, the return value is TRUE if the Bases match along all axes, and FALSE if they differ along any axis.

A conformability test is included to allow the user to optionally test the conformability of two array objects before the array operation.

A.isSameBase(B);	Check bases of both arrays along all axes (all bases equal return TRUE)
A.isSameBound(B);	Check bounds of both arrays along all axes (all bounds equal return TRUE)
A.isSameStride(B);	Check strides of both arrays along all axes (all strides equal return TRUE)
A.isSimilar(B);	Check bases, bounds, and strides of both arrays along all axes
A.isConformable(B);	Checks conformability of both arrays

10.5.15 Array Object Internal Consistancy Test

This function tests the internal values for consistancy it is mostly included for completeness. It is most usefull within P++ where there is significant testing that can be done between local and global data to verify consistant internal behavior. It is used within A++ and P++ when internal debugging is turned on (not the default in distribution versions of A++ and P++.

A.isConsistant();	Checks internal consistancy of array object

10.5.16 Shape functions

These shape functions redimension an existing array object. The reshape function allows the conversion of an nxm array to an mxn array (2D example), the total number of elements in the array must remain the same and the data values are preserved. The redim function redimensions an array to a different total size (larger of smaller), but does not preserve the data (data is left uninitialized). The resize function is similar to the redim function except that it preserves the data (truncating the data if the new dimensions are smaller and leaving new values uninitialized if the new dimensions are larger. Each function can be used with either scalar or Range object input parameters, additionally each function may be provided an example array object from which the equivalent Range objects are extracted (internally). All these member functions preserve (save and reset) the original base of the array object.

A.reshape(i,j,k,l);	Change dimensions of array using the same array data (same size)
A.reshape(Span_I,Span_J,Span_K,Span_L);	Change dimensions of array using the same array data (same size)
A.reshape(*type***Array**);	Change size of array object using another array object
A.resize(i,j,k,l);	Change size of array object (old data is copied and truncated)
A.resize(Span_I,Span_J,Span_K,Span_L);	Change size of array object using Range objects
A.resize(*type***Array**);	Change size of array object using another array object
A.redim(i,j,k,l);	Change size of array object (old data is lost)
A.redim(Span_I,Span_J,Span_K,Span_L);	Change size of array object using Range objects
A.redim(*type***Array**);	Change size of array object using another array object
transpose (A);	transpose of elements of A

10.5.17 Display Functions

A(I,J).display("label");	Prints array data for the view A(I,J) along with character string "label" to sdtout
A.view("label");	Prints array data and all other internal data for A along with character string "label" to sdtout

Details of the display of the values within an array by the display function are controled by the values assigned to the *type***Array**::DISPLAY_FORMAT variable. This variable has a default value of *type***Array**::SMART_DISPLAY_FORMAT which allows for the auto selection of either DECIMAL or EXPONENTIAL format depending upon the values within the array. Display Format Control Values:

*type***Array**::DISPLAY_FORMAT = *type***Array**::DECIMAL_DISPLAY_FORMAT;	Uses xxx.yyyy format
*type***Array**::DISPLAY_FORMAT = *type***Array**::EXPONENTIAL_DISPLAY_FORMAT;	Uses x.yyyye±zz format
*type***Array**::DISPLAY_FORMAT = *type***Array**::SMART_DISPLAY_FORMAT;	Auto-selects either of above formats

10.5.18 Array Expressions Used For Function Input

Functions passing array objects by reference can't be passed an expression since expressions return temporaries that are managed differently internally. Functions passing expressions by value require no special handling.

foo (evaluate (A+B));	Force (A+B) temporary to be persistent for function *foo* , which passes an array object by reference

10.5.19 Array Aliasing

A++ and P++ arrays can be aliased however all caveats apply as in the use of FORTRAN equivalence. This permits array object to be views of other array objects or indexed parts of other array objects. Note that P++'s adopt function must build the distributed array from the collection of pointers to local memory in each processor and so requires both global and local domain size information (P++ organizes any communication that is required to build the distributed array (currently there is no communication required)).

B.reference (A(I,J));	Force B to reference A(I,J)
B.breakReference ();	Break reference to A(I,J) (builds a copy of previous reference)

A++ only

C.adopt (Fortran_Array_Pointer, n);	1D array C of length n using existing array
C.adopt (Fortran_Array_Pointer, n,m);	2D array C of length n × m using existing array
C.adopt (Fortran_Array_Pointer, n,m,o);	3D array C of length n × m × o using existing array
C.adopt (Fortran_Array_Pointer, n,m,o,p);	4D array C of length n × m × o × p using existing array

P++ only

C.adopt (Fortran_Array_Pointer, n, Local_Size_n);	1D array C of length n using existing array
C.adopt (Fortran_Array_Pointer, m, Local_Size_m, n, Local_Size_n);	2D array C of length n × m using existing array
C.adopt (Fortran_Array_Pointer, m, Local_Size_m, n, Local_Size_n, o, Local_Size_o);	3D array C of length n × m × o using existing array
C.adopt C (Fortran_Array_Pointer, m, Local_Size_m, n, Local_Size_n, o, Local_Size_o, p, Local_Size_p);	4D array C of length n × m × o × p using existing array

10.5.20 Fill Function

More fill functions will be added to later releases of A++/P++. Its purpose is to initialize an array object to value or set of values.

B(I,J).fill(x);	Set elements of B(I,J) equal to x
B(I,J).seqAdd(Base,Stride);	Set elements of B(I,J) equal to Base, Base+Stride, ... , Base+n*Stride default value for Base and Stride are 0 and 1

10.5.21 Access To FORTRAN Ordered Array

A++/P++ provides access to the internal data of the array object using the following access functions. Arrays are stored internally in FORTRAN order and a pointer to the start of the array can be obtained using the getDataPointer member function. In the case of a view the pointer is to the start of the view. It is up to the user to correctly manipulate the data (good luck). Similar access is provide to the array descriptor (though info for it's use is not contained in this Quick_Reference_Manual).

Fortran_Array_Pointer = A.getDataPointer(); Array_Descriptor_Type = A.getDescriptorPointer();

10.6 "where" Statement

Example of **where** statement support in A++/P++. Note that elsewhere statements may be cascaded and that an optional parameter (Mask) can be specified. Note that **elsewhere** must have a set of parenthesis even if no parameter is specified. The mask must be conformable with the array operations in the code block. *On the Cray, and with the GNU g++ compiler, the statement* **elsewhere(mask)** *taking a mask as a parameter is called* **elsewhere_mask(mask)**. *This is due to a problem with parameter checking of macros. The syntax for* **elsewhere()**, *not taking a mask, does not change.* This aspect of A++ syntax may be changed slightly to accommodate these non-portable aspects of the C preprocessor.

```
where  (A == 0)
     {
     B = 0;                   elements of B set to zero at positions where A = 0
     A = B + C;               B added to C and assigned to A at positions where A = 0
     }
elsewhere (B > 0)             Use elsewhere_mask on the Cray and with GNU g++
     {
     B = A;                   elements of B set to A at positions where A ≠ 0 and B > 0
     }
elsewhere ()
     {
     B = A;                   elements of B set to A at positions where A ≠ 0 and B ≤ 0
     }
```

10.7 P++ Specific Information

There are access functions to the lower level objects in P++ which can be manipulated by the user's program. Specifically we provide access to the **Partitioning_Type** that each array uses internally (if it is not using the default distribution). The purpose of providing manual ghost boundary updates is to permit override of the message passing interpretation provide in P++. The resulting reduced overhead provides a simple means to optimize performance of operations the user recognizes as not requiring more than an update of the internal ghost boundaries. The "displayPartitioning" member function prints out ASCII text which describes the distribution of the P++ array on the multiprocessor system. The same functions exist in A++ but don't do anything, this supports backward compatibility between P++ and A++.

10.7.1 Control Over Array Partitioning (Distributions)

The distribution of P++ array objects is controled though partitioning objects that are associated with the array objects. The association of a partitioning object with and array is done either at construction of the array objects or later in the probram. An unlimited number of array objects may be associated with a given partitioning object. The manipulation of the partitioning object translates directly to manipulation of each of the array objects associated with the partitioning. This feature makes it easier to manipulate large number of arrays with a simple interface. Partitioning objects are valid object in A++, but have no meaningful effect, so they are only functional in P++. This is to permit bidirectional portability between A++ and P++ (the serial and parallel environments). An unlimited number of **Partitioning_Type** objects may be used within an application. One of the main purposes of the partitioning objects is to define the distribution of P++ arrays and permit the dynamic redistribution. The expected usage is to have many P++ arrays associated with a relatively small number of **Partitioning_Type** objects.

Constructors

At present the constructor taking a intArray as a parameter is not implemented, it's purpose is to provide a simple means to control load balancing; it is the interface for a load balancer. But load balancing is not a part of A++/P++, load balancers used with parallel P++ applications are presently separate from P++. The most common usage of the partitioning object is to either call the constructor which specifies a subrange of the virtual processor space (this will be truncated to the exitisting virtual processor space if too large a range is specified), or call the default constructor (the whole virtual processors space) and then call member functions to modify the partitioning object.

Partitioning_Type P ();	Default constructor
Partitioning_Type P (Load_Map);	Load_Map is a intArray specifying the work distribution
Partitioning_Type P (Number_Of_Processors);	integer input specifies number of processors to use (start=0)
Partitioning_Type P (Span_P);	**Range** input specifies range of processors to use
Partitioning_Type P1 = P;	Deep copy constructor

Member functions

The operations on a **Partitioning_Type** object are done to all P++ arrays that are associated through that **Partitioning_Type** object. This provides a powerful mechanism for the dynamic control of array distributions; load balancers are expected to take advantage of this feature. The "applyPartition" member function is provided so that multiple modifications to the partitioning object may be done and a single restructuring of the P++ arrays associated with the partitioning object completed subsequently. P++ operation is undefined if the partitioning is never applied to it's associated objects. At present, only the partitionAlongAxis member function does not call the applyPartition function automatically. This detail of the interface may change in the near future to allow a more simple usage.

The partitionAlongAxis member function takes three parameters: int Axis, bool Partitioned, int GhostBoundaryWidth. This simplifies the setting and modification of the partitioning. Afterward this only takes effect once the applyPartition member function is called. Then all distributed arrays associated with the partitioning object are redistributed with the ghost boundaries that were specified.

SpecifyDecompositionAxes (Input_Number_Of_Dimensions_To_Partition);	Integer input
SpecifyInternalGhostBoundaryWidths (int,int,int,int);	Default input is zero

display (Label);	printout partition data
displayDefaultValues (Label);	printout default partition data
displayPartitioning (Label);	graphics display of partition data
displayDefaultPartitioning (Label);	graphics display of default partition data
updateGhostBoundaries (X);	X is a P++ array
partitionAlongAxis (int Axis, bool PartitionAxis, int GhostBoundaryWidth);	input specifies axis
applyPartition ();	force partitioning of previously associated P+

10.7.2 Array Object Member Functions

Array objects have some specific member functions that are meaningful only within P++, as A++ array objects the member functions are defined, but have do nothing. This is done for backward compatability.

Partitioning_Type *X = A.getPartition();	get the internal partition
A.partition(**Partition**);	repartition dynamically
A.partition(*type***Array**);	repartition same as existing array object
A.getLocalBase(axis);	return base of local processor data
A.getLocalBound(axis);	return bound of local processor data
A.getLocalStride(axis);	return stride of local processor data
A.getLocalLength(axis);	return length of local processor data
A.getLocalFullRange(axis);	return a Range object (base,bound,stride of the local array)
A.getSerialArrayPointer();	return a pointer to the local array (and A++ array)
A.getLocalArray();	return a shallow copy of the local array (and A++ array)
A.getLocalArrayWithGhostBoundaries();	return a shallow copy (with ghost boundaries)
A.updateGhostBoundaries();	updates all ghost bondaries
A.displayPartitioning();	prints info on distribution of array data
A.getGhostCellWidth(Axis);	access to ghost boundary width
A.getInternalGhostCellWidth(Axis);	access to ghost boundary width (devalued, will be removed in future r
A.setInternalGhostCellWidth(int,int,int,int);	dynamicly adjusts ghost boundary width
A.setInternalGhostCellWidthSaveData(int,int,int,int);	as above but preserves the data and updates ghost boundaries

10.7.3 Distributed vs Replicated Array Data

Within P++ arrays are distributed, distributions have the following properties:

- 1 An array is distributed in some or all of the dimensions of the array (the user selects such details).
- 2 An array is distributed over a subset of processors.
- 3 An array is distributed over only a single processor (a trivial case of #2 above).
- 4 An array is built onto only one processor and only that processor knows about it (i.e. an A++ array object is built locally on a processor).
- 5 An array is replicated onto all processors (this is really a trival case of #4 above where each array is built locally on each processor). In this case the user is responcible for maintaining a consistant representation of the data which is replicated. This later case is useful for when a small array is required and is analogous to the case of replication of scalars onto every processor since no overhead of parallel support.

P++ also contains **SerialArrays**, (e.g. **doubleSerialArray**). These arrays are simply A++ array objects on each processor. In a data parallel way, if all processors build a serial array object, then each processor builds an array and the array is replicated across all processors. It is up to the user to maintain the consistancy of the array data across all processors in this case. Many arrays that are small are simply replicated, this costs little in additional space and avoid any communication when data is accessed.

10.7.4 Virtual Processors

P++ uses a number of processors independent upon the number of actual processors in hardware. On machines that support it the excess processors are evenly distributed among the hardware processors. This allows for greater control of granularity in the distribution of work. Where it is important to take advantage of this is application dependent. For most of the development this has allowed us to test problems on a number of processors indepentend of the actual number of machines that we have in our workstation cluster.

10.7.5 Synchronization Primative

Note that the Communication_Manager::Sync() is helpful in verifying the all processors reach a specific point in the parallel execution. This is helpful most often for debugging parallel codes.

Communication_Manager::Sync();	Call barrier function to sync all processors

10.7.6 Access to specific Parallel Environment Information

Although access to the underlying parallel information such as processor number, etc. can be used to break the data parallel model of execution such information is made available within P++ because it can be useful if used correctly. As an example of correct useage moving an application using graphics from A++ to P++ often is simplified if a specific processor is used for all the graphics work while others are idle. Access to the process number allows the code on each processor to branch dependent upon the processor number and thus simplifies (at initially) the movement of large scale A++ applications onto parallel machines using P++. Some of the data is only valid for either PVM or MPI, and some data is interpreted different by the two communication libraries.

Communication_Manager::numberOfProcessors();	get number of virtual processors
Communication_Manager::localProcessNumber();	get processor id number
Communication_Manager::Sync();	barrier primative
Communication_Manager::My_Task_ID;	get process id
Communication_Manager::MainProcessorGroupName;	Name of MPI Group

10.7.7 Escaping from the Data Parallel Execution Model

Since the data parallel style is only assumed for the execution of P++ array operations, but not enforced, it is possible to break out of the Data Parallel model and execute any parallel code desired. Users however are expected to handle their own communication. Since some degree of syncronization is helpful in moving into and out of the data parallel modes, the Communication_Manager::Sync() function is expected to be used (though not required).

10.7.8 Access to the local array

Each P++ distributed array on each processor contains a local array (a SerialArray object (same as an A++ array object)). The local array is availabel with and without ghost boundaries.

Access to the local array without ghost boundaries

The local array stores the local part of the distributed array data. Access to the localArray is obtained from:

A.getLocalArray();	return a shallow copy of the local array (an A++ array)

Access to the local array with ghost boundaries

Ghost boundaries are not visible within the local array since the local array is a view of the partition of the distributed space on the current processor. The ghost boundaries (if the ghost boundary width is nonzero) **are** present, but access to them from the view would result in an out of range error. Another mechanism for accessing the local array is required to get the local array containing the ghost boundaries.

A.getLocalArrayWithGhostBoundaries();	return a shallow copy (with include ghost boundaries)

The access to the ghost boundaries is possible from this view, but the user must know how to interpret the ghost boundaries within the returned local array object. (Hint: they are at the boundaries and the widths along each access are given by the ghost boundary widths obtained from the partitions.)

10.7.9 Examples of P++ specific operations

We provide some simple examples within the A++/P++ manual, please consult that chapter on Examples to see illustrations of the useage of the P++ specific functions.

10.8 Optimization Manager

Optimization manager is an object whose member functions control properties of the execution of the A++ and P++ array class (see reference manual). More member functions later will allow for improved optimization potential. The setup of the "Virtual Machine" may be separated outside of the P++ interface since not all machine environments require it (both MPI and PVM do, so it is present in P++ currently).

The "Program_Name" should be initialized with the complete name of the executable (including path), however in environments where it is supported P++ will automatically search for the string if only `""` is specified. This is a feature that can not be supported on all architectures (or PVM would handle it internally).

Initialize_Virtual_Machine (char* Program_Name = "" , int Num_Processors = 1, int argc, char argv);**	First P++ statement
Exit_Virtual_Machine ();	Last P++ statement
setOptimizedScalarIndexing (On_Off_Type On_Off = On);	Optimize performance

10.9 Diagnostic Manager

There are times when you want to know details about what is happening internally within A++/P++. We provide a limited number of ways of seeing what is going on internally and getting some data to understand the behavior of the users application. More will be added in future versions of A++/P++.

10.9.1 Report Generation

There are a number of Diagnostic manager function which generate reports of the internal useage. Some reports are quite long, other are brief and summarize the execution history for the whole application.

getSizeOfClasses();	Reports the sizes of all internal classes in A++/P++
getMemoryOverhead();	returns memory overhead for all arrays
getTotalArrayMemoryInUse();	returns memory use for array elements
getTotalMemoryInUse();	reports total memory use for A++/P++
getnumberOfArraysConstantDimensionInUse(dimension, inputTypeCode);	reports by array dimension
getMessagePassingInterpretationReport();	Communication Report
getReferenceCountingReport();	Reference Counting Report
displayCommunication (const char* Label = "");	communication report by processor
displayPurify (const char* Label = "");	Displays memory leaks by processor (uses purify)
report();	Generates general report of A++/P++ behavior
setTrackArrayData(Boolean trueFalse = TRUE);	Track and report on A++/P++ diagnostics
getTrackArrayData();	get Boolean value for diagnostic mechanism
buildCommunicationMap ();	Builds map of communications by processor
buildPurifyMap ();	Builds map of purify errors by processor
getPurifyUnsupressedMemoryLeaks();	Total Memory leaked

Features and counted quantities include:

- The use of **int Diagnostic_Manager::getSizeOfClasses()** displays a text report of the sizes of different internal structures in A++P++.
- The use of **int Diagnostic_Manager::getMemoryOverhead()** returns an integer that represents the number fo byte of overhead used to store intenal array descriptors, partitioning information (P++ only), etc.; for the whole application at the time that the function is called.
- The use of **int Diagnostic_Manager::getTotalArrayMemoryInUse()** returns an integer representing the total number of array elements in use in all array objects at the time that function is called.
- The use of **int Diagnostic_Manager::getTotalMemoryInUse()** returns the total number of bytes in use within A++/P++ for all overhead and array elements at the time the function is called.

- The use of **int Diagnostic_Manager::getnumberOfArraysConstantDimensionInUse()** returns the number of arrays of a particular dimension and of a particular type. this function is an example of the sort of diagnostic questions that can be written which interogate the runtime system to find out both global and local properties of its operation.
- The use of **int Diagnostic_Manager::getMessagePassingInterpretationReport()** generates a report (organized from each processor, but reported on processor 0). The report details the number of MPI sends, MPI receives, the number of ghost boundary updates (one update implies the update of all ghost boundaries on an array, even if this generates fewer MPI messages than ghost boundaries), and the number VSG updates *regular section transfers* (the more general communication model which permits operations between array objects independent of the distribution across multiple processors).
- The use of **int Diagnostic_Manager::getReferenceCountingReport()** generates a report of the internal reference counts used in the execution of array expressions. This function is mostly for internal debugging of reference counting problems.
- The use of **int Diagnostic_Manager::report()** generates a summary report of the execution of the A++/P++ application at the point when it is called.
- The use of **int Diagnostic_Manager::setTrackArrayData()** turns on the internal tracking of array objects as part of the internal diagnostics and permits the summary report to report more detail. It is off by default so that there is no performance penalty associated with internal diagnostics. This must be set at the top of an application before the first array object is built.

10.9.2 Counting Functions

Optional mechanisms in A++/P++ permit many details to be counted internally as part of the report generation mechanisms. All functions return an integer.

resetCommunicationCounters ();	reset the internal message passing counting mechanism
getNumberOfArraysInUse();	returns the number of arrays inuse
getMaxNumberOfArrays();	returns the max arrays in used at any point in time
getNumberOfMessagesSent();	returns the number of messages sent
getNumberOfMessagesReceived();	returns the number of messages received
getNumberOfGhostBoundaryUpdates();	returns number of updates to ghostboundaries
getNumberOfRegularSectionTransfers();	# of uses of general communication mechanism
getNumberOfScalarIndexingOperations();	scalar indexing
getNumberOfScalarIndexingOperationsRequiringGlobalBroadcast();	scalar indexing with communication

Features and counted quantities include:

- The use of **int Diagnostic_Manager::resetCommunicationCounters()** permits the internal counters to be reset to ZERO.
- Number of arrays in use **int Diagnostic_Manager::getNumberOfArraysInUse()** The number of arrays in use at any point in the execution is useful for gauging the relative use od A++/P++ and spotting potential memory leaks.
- Max arrays in use **int Diagnostic_Manager::getMaxNumberOfArrays()** This function tallies the most number of arrays in use at any one time durring the execution history (note: records use in increments of 300).
- Reset message counting **int Diagnostic_Manager::resetCommunicationCounters()** Resest the message counters to ZERO to permit localized counting of messages generated from code fragements.
- Number of messages (sent) **int Diagnostic_Manager::getNumberOfMessagesSent()** returns the total messages since the beginning of execution or from the last call to **Diagnostic_Manager::resetCommunicationCounters()**.
- Number of messages (received) **int Diagnostic_Manager::getNumberOfMessagesReceived()** returns the total messages since the beginning of execution or from the last call to **Diagnostic_Manager::resetCommunicationCounters()**.
- Number of messages (received) **int Diagnostic_Manager::getNumberOfGhostBoundaryUpdates()** Returns the total number of calls to update the ghost boundaries of arrays. Note that some calls will not translate into message passing (e.g. if only run on one processor or if the ghost boundary width is ZERO). Reports on number of messages since the beginning of execution or from the last call to **Diagnostic_Manager::resetCommunicationCounters()**.

10.9.3 Debugging Mechanisms

These functions provide mechanisms to simplify the error checking and debugging of A++/P++ applications.

getPurifyUnsupressedMemoryLeaks();	Total Memory leaked
setSmartReleaseOfInternalMemory(On/Off);	Smart Memory cleanup
getSmartReleaseOfInternalMemory();	get Boolean value for smart memory cleanup
setExitFromGlobalMemoryRelease(Boolean);	setup exit mechanism
getExitFromGlobalMemoryRelease();	get Boolean value for exit mechanism
test (*type***Array**);	Destructive test of array object
displayPurify (const char* Label = "");	Displays memory leaks by processor (uses puri
buildPurifyMap ();	Builds map of purify errors by processor

- The use of **void Diagnostic_Manager::setSmartReleaseOfInternalMemory()** (called from anywhere in an A++/P++ application) will trigger the mechanism to cleanup all internally used memory within A++/P++ after the last array object has been deleted. Specifically it counts the number of arrays in use (and the number of arrays used internally (e.g. where statement history, etc.) and when the two values are equal it calls the **void globalMemoryRelease()** function which then deletes existing arrays in use and other data used internally (reference count arrays, etc.). The user is warned in the output of the **void globalMemoryRelease()** function to not call any functions that would use A++/P++ since the results would be *undefined*.

- The use of the **void Diagnostic_Manager::setExitFromGlobalMemoryRelease()** will force the application to exit after the global memory release (and from within the **void globalMemoryRelease()** function itself. The user may then specify that the normal exit from the base of the **main** function is an error and thus detect the proper cleanup of memory in test programs using the exit status (stored in the **$status** *enviroment variable* on all POSIX operating systems (most flavors of UNIX). If purify is in use (both A++/P++ configured to use **purify** and running with purify) then **purify_exit(int)** is called. This function or's the memory leaks, memory in use, and purify errors into the exist status so that the **$status** *enviroment variable* can be used to detect purify details within test codes. A++/P++ test codes are tested this way when A++/P++ is configured to use PURIFY. P++ applications can not always communicate detected purify problems on other processes AND output the correct exit status, this is only a limitation of how **mpirun** returns it's exit status.

- The use of **void Diagnostic_Manager::test(***type***Array** A**)** allows for exhaustive (destructive) tests of an arrya object. This is useful in testing an array object for internal correctness (more robust testing than the nondestructive testing done in the Test_Consistancy() array member function).

- The use of **void Diagnostic_Manager::displayPurify()** generates a report of purify problems found (currently this mechanism does not work well, since many purify errors can only be found at exit).

10.9.4 Misc Functions

All other functions not yet documented in detail.

getMessagePassingInterpretationReport();	Communication Report
getReferenceCountingReport();	Reference Counting Report
getSizeOfClasses();	Reports the sizes of all internal classes in A
getMemoryOverhead();	returns memory overhead for all arrays

getTotalArrayMemoryInUse();	returns memory use for array elements
getTotalMemoryInUse();	reports total memory use for A++/P++
getnumberOfArraysConstantDimensionInUse(dimension,inputTypeCode);	reports by array dimension
displayPurify (const char* Label = "");	Displays memory leaks by processor (uses purify)
getPurifyUnsupressedMemoryLeaks();	Total Memory leaked
report();	Generates general report of A++/P++ behavior
setSmartReleaseOfInternalMemory(On/Off);	Smart Memory cleanup
getSmartReleaseOfInternalMemory();	get Boolean value for smart memory cleanup
setExitFromGlobalMemoryRelease(Boolean);	setup exit mechanism
getExitFromGlobalMemoryRelease();	get Boolean value for exit mechanism
setTrackArrayData(Boolean trueFalse = TRUE);	Track and report on A++/P++ diagnostics
getTrackArrayData();	get Boolean value for diagnostic mechanism
test (*type***Array**);	Destructive test of array object
buildCommunicationMap ();	Builds map of communications by processor
buildPurifyMap ();	Builds map of purify errors by processor
displayCommunication (const char* Label = "");	communication report by processor
resetCommunicationCounters ();	reset the internal message passing counting mechanism

10.10 Deferred Evaluation

Example of user control of Deferred Evaluation in A++/P++. Deferred Evaluation is a part of A++ and P++, though it is not well tested in P++ at present.

Set_Of_Tasks Task_Set;	build an empty set of tasks
Deferred_Evaluation (Task_Set)	start deferred evaluation
{	
B = 0;	array operation to set B to zero – DEFERRED
A = B + C;	array operation to set A equal to B plus C – DEFERRED
}	
Task_Set.Execute();	now execute the deferred operations

10.11 Known Problems in A++/P++

- Copy constructors are aggressively optimized away by some compilers and this results in the equivalent of shallow copies being built in the case where an A++/P++ array is constructed from a view. Note that as a result shallow copies of A++ arrays can be made unexpectedly. A fix for this is being considered, but it is not implemented.
- Performance of A++ is at present half that of optimized FORTRAN 77 code. This is because of the binary processing of operands and the associated redundent loads and stored that this execution model introduces. A version of A++/P++ using expression templates will resolve this problem, this implementation is available and is present as an option within the A++/P++ array class library. However, compile times for expression templates are quite long.

- Internal debugging if turned on at compile time for A++/P++ will slow the execution speed. The effect on A++ is not very dramatic, but for P++ it is much more dramatic. This is because P++ has much more internal debugging code. The purpose of the internal debugging code is to check for errors as agressively as possible before they effect the execution as a segment fault of other mysterious error.
- Performance of P++ is slower if the array operations are upon array data that is distributed differently across the multiple processors. This case requires more communication and for arrays to be built internally to save the copies originally located upon different processors. P++ performance is most efficient if the array objects are aligned similarly across the multiple processors. This case allows the most efficient communication model to be used internally. This more efficient communication model introduces no more communication than an explicitly hand coded parallel implementation on a statement by statement basis.

The `ChangeLog` in the top level of the A++P++ distribution records all modifications to the A++/P++ library.

Chapter 11

Appendix

11.1 A++/P++ Booch Diagrams

Booch diagrams detail the object oriented design of a class library. The separate clouds represent different classes. Those which are shaded represent classes that are a part of the user interface, all others are those which are a part of the implementation. The connections between the ”clouds” represent that the class uses the lower level class (the one with out the associated ”dot”) within its implementation.

11.2 A++/P++ Error Messages

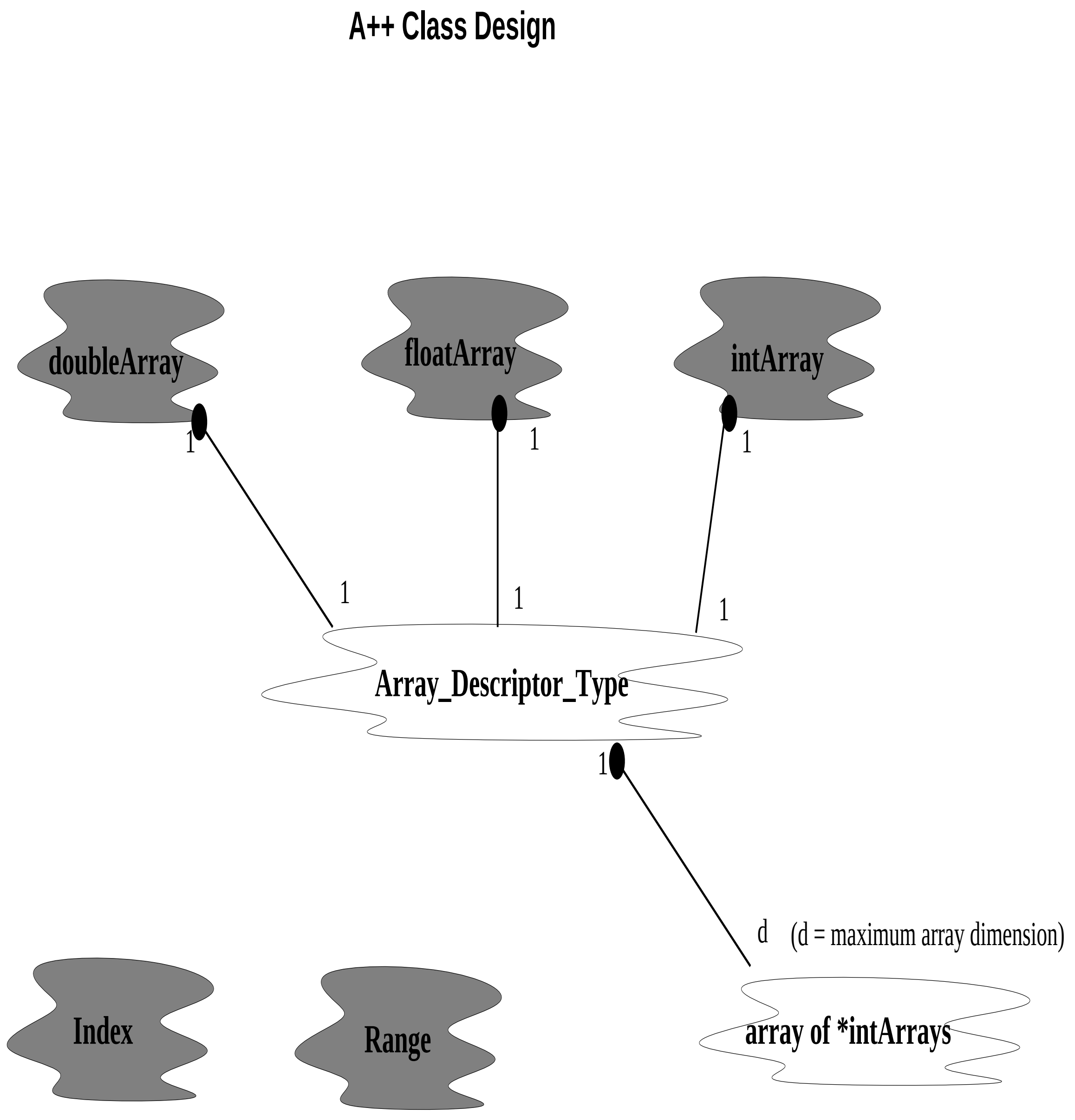
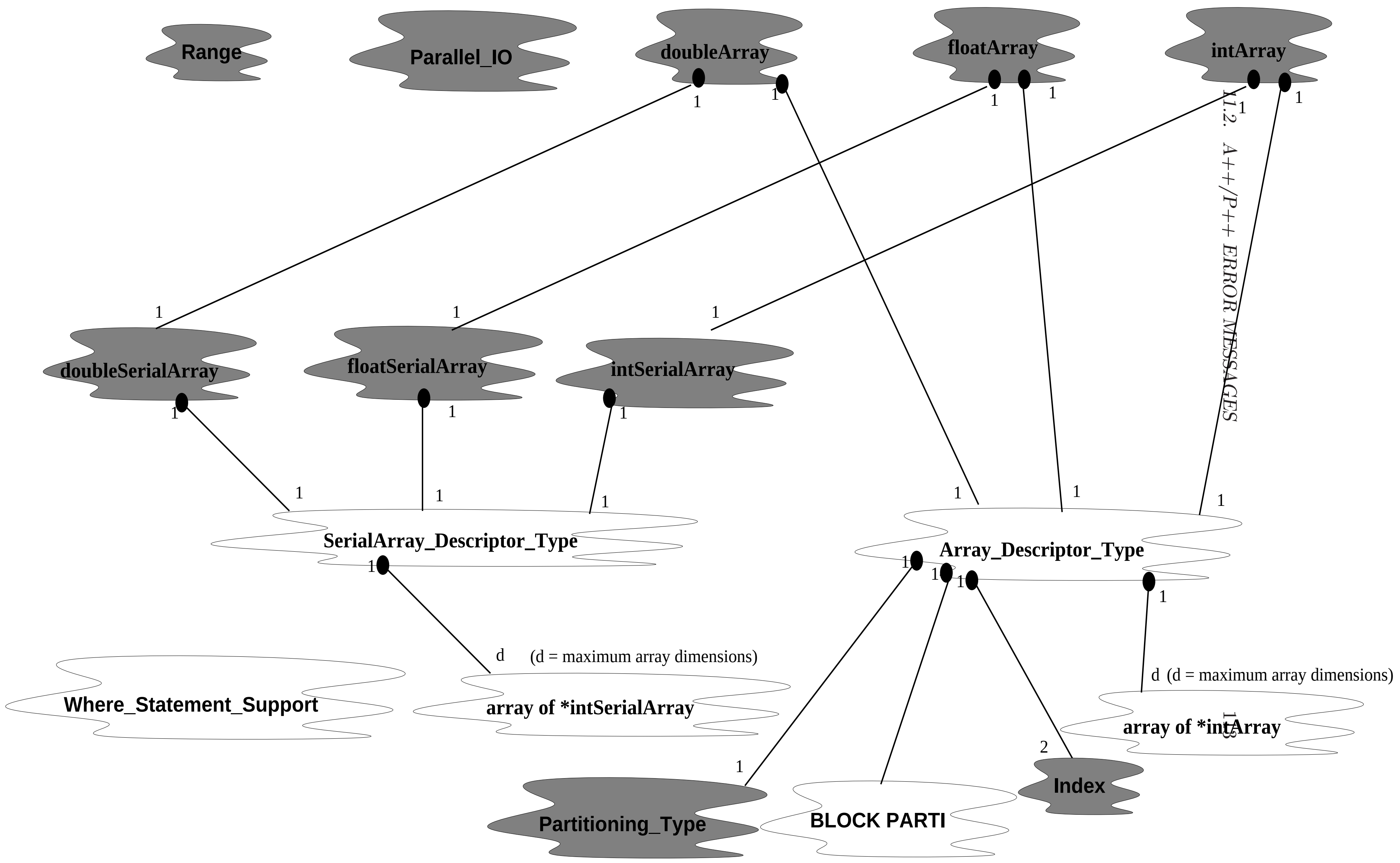


Figure 11.1: A++ Class Design.

P++ Class Design



1

Chapter 12

Glossary

We define terms used in the A++/P++ manual which might otherwise be unclear.

- **Array Object:** Any istantiation of an A++/P++ $< type >$Array.
- **Block Parti:** Low level library used by P++ for control of partitioning and communication. Parti uses any of several parallel communication libraries.
- **Conformal Operation:** An operation between arrays where the referenced sections manipulated are of the same size.
- **Data Parallelism:** Parallel execution of single expressions on data distributed over multiple processors.
- **Ghost Boundaries:** Internal data which replicate the edge of a partition (with some width) on the adjacent processor when arrays are partitioned across multiple processors. Ghost boundaries are only present if an array is partitioned and the ghost boundary with is specified to be greater then zero.
- **Index:** Fortran 90 like triple containing base, length, and stride.
- **Partition:** The division of array data across multiple processors.
- **Range:** Fortran 90 like triple containing base, bound, and stride.
- **Task Parallelism:** Parallel execution of multiple expressions on data on multiple processors. Operations may be different in each task. The control of task parallelism is more difficult than data parallelism. A++/P++ attempts to mix the two, but requires access to a task parallel model externally provided (as in use with a parallel C++ language). Since A++/P++ is a class library it can work easily with most research oriented parallel C++ compilers.

Bibliography

[1] **Angus I. G. and Thompkins W. T.:** Data Storage, Concurrency, and Portability: An Object Oriented Approach to Fluid Dynamics; Fourth Conference on Hypercubes, Concurrent Computers, and Applications, 1989.

[2] **Baden, S. B.; Kohn, S. R.:** Lattice Parallelism: A Parallel Programming Model for Non-Uniform, Structured Scientific Computations; Technical report of University of California, San Diego, Vol. CS92-261, September 1992.

[3] **Balsara, D., Lemke, M., Quinlan, D.:** AMR++, a C++ Object Oriented Class Library for Parallel Adaptive Mesh Refinement Fluid Dynamics Applications, Proceeding of the American Society of Mechanical Engineers, Winter Anual Meeting, Anahiem, CA, Symposium on Adaptive, Multilevel and Hierarchical Computational Stratagies, November 8-13, 1992.

[4] **Berryman, H.; Saltz, J. ; Scroggs, J.:** Execution Time Support for Adaptive Scientific Algorithms on Distributed Memory Machines; Concurrency: Practice and Experience, Vol. 3(3), pg. 159-178, June 1991.

[5] **Chandy, K.M.; Kesselman, C.:** CC++: A Declarative Concurrent Object-Oriented Programming Notation; California Institute of Technology, Report, Pasadena, 1992.

[6] **Chase, C.; Cheeung, A.; Reeves, A.; Smith, M.:** Paragon: A Parallel Programming Environment for Scientific Applications Using Communication Structures; Proceedings of the 1991 Conference on Parallel Processing, IL.

[7] **Forslund, D.; Wingate, C.; Ford, P.; Junkins, S.; Jackson, J.; Pope, S.:** Experiences in Writing a Distributed Particle Simulation Code in C++; USENIX C++ Conference Proceedings, San Francisco, CA, 1990.

[8] **High Performance Fortran Forum:** Draft High Performance Fortran Language Specification, Version 0.4, Nov. 1992. Available from titan.cs.rice.edu by anonymous ftp.

[9] **Lee, J. K.; Gannon, D.:** Object-Oriented Parallel Programming Experiments and Results; Proceedings of Supercomputing 91 (Albuquerque, Nov.), IEEE Computer Society and ACM SIGARCH, 1991, pg. 273-282.

[10] **Lemke, M.; Quinlan, D.:** Fast Adaptive Composite Grid Methods on Distributed Parallel Architectures; Proceedings of the Fifth Copper Mountain

Conference on Multigrid Methods, Copper Mountain, USA-CO, April 1991. Also in Communications in Applied Numerical Methods, Wiley, Vol. 8 No. 9 Sept. 1992.

[11] **Lemke, M.; Quinlan, D.:** P++, a C++ Virtual Shared Grids Based Programming Environment for Architecture-Independent Development of Structured Grid Applications; Arbeitspapiere der GMD, No. 611, 20 pages, Gesellschaft für Mathematik und Datenverarbeitung, St. Augustin, Germany (West), February 1992.

[12] **Lemke, M.; Quinlan, D.:** P++, a C++ Virtual Shared Grids Based Programming Environment for Architecture-Independent Development of Structured Grid Applications; accepted for CONPAR/VAPP V, September 1992, Lyon, France; to be published in Lecture Notes in Computer Science, Springer Verlag, September 1992.

[13] **Lemke, M., Quinlan, D., Witsch, K.:** An Object Oriented Approach for Parallel Self Adaptive Mesh Refinement on Block Structured Grids, Preceedings of the 9th GAMM-Seminar Kiel, Notes on Numerical Fluid Mechanics, Vieweg, Germany, 1993.

[14] **McCormick, S., Quinlan, D.:** Asynchronous Multilevel Adaptive Methods for Solving Partial Differential Equations on Multiprocessors: Performance results; Parallel Computing, 12, 1989, pg. 145-156.

[15] **McCormick, S.; Quinlan, D.:** Multilevel Load Balancing, Internal Report, Computational Mathematics Group, University of Colorado, Denver, 1987.

[Oliver] Ian Oliver.1993 *Programming Classics: Implementing the World's Best Algorithms.* Englewood Cliffs,N.J.: Prentice Hall.

[16] **Peery, J.; Budge, K.; Robinson, A.; Whitney, D.:** Using C++ as a Scientific Programming Language; Report, Sandia National Laboratories, Albuquerque, NM, 1991.

[17] **Schoenberg, R.:** M++, an Array Language Extension to C++; Dyad Software Corp., Renton, WA, 1991.

[18] **Stroustrup, B.:** The C++ Programming Language, 2nd Edition; Addison-Wesley, 1991.