A++/P++ Tutorial Guide (version 0.7.2)

Daniel Quinlan and Nehal Desai
Lawrence Livermore National Laboratory
L-560
Livermore, CA 94550
925-423-2668 (office)
925-422-6287 (fax)
dquinlan@llnl.gov
http://www.llnl.gov/casc/people/dquinlan
A++/P++ Web Page http://www.llnl.gov/casc/Overture/A++P++
A++/P++ Tutorial Guide (postscript version)
LACC Number: LA-CC-96-1
LAUR Number: LA-UR-??-????

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

0 Copyright **3**
0.1 In Plain English 3
0.2 NOT In Plain English 3

1 Tutorial **13**
1.1 Introducton 13
1.2 Examples 14
1.2.1 Example 1a. ”Hello, World” 14
1.2.2 Example 1b. “Parallel Hello World” 15
1.2.3 Example 2. 1-D Laplace Equation Solver 16
1.2.4 Example 3. Distribution of Arrays in P++ 18
1.2.5 Example 4. The Heat Equation 21
1.2.6 Example 5. Indirect Addressing 24
1.2.7 Example 6. Application of Indirect Addressing 26
1.3 Example Makefile 31
1.4 More example on the A++/P++ Home Page 33

Chapter 1

Tutorial

1.1 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.

1.2 Examples

1.2.1 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
```

1.2.2 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.

1.2.3 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
```

1.2.4 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();
}
```

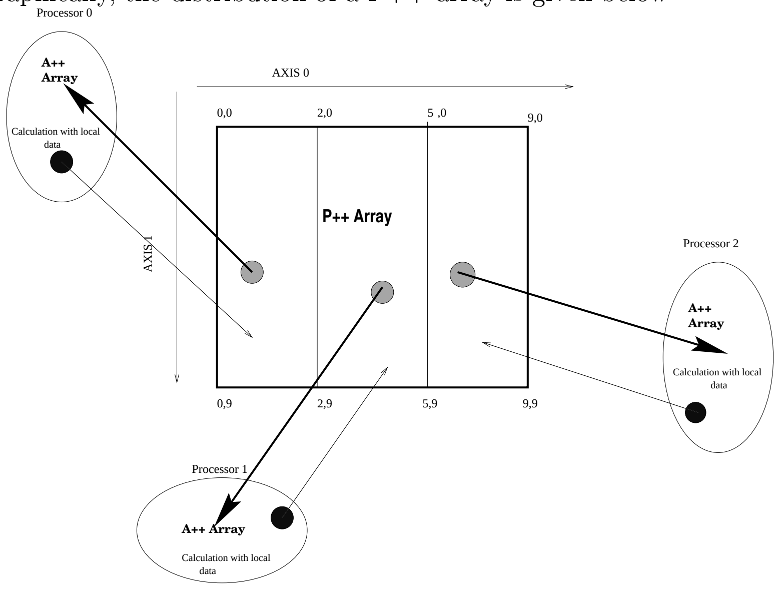
Output from example 3

```
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



1.2.5 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>
```

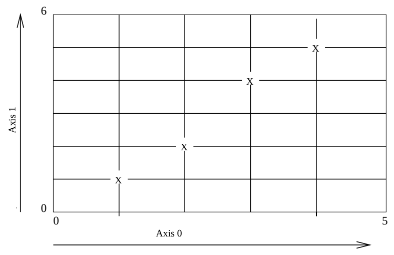
```
main(int argc,char** argv)
{
int num_of_process=5;
Optimization_Manager::Initialize_Virtual_Machine("",num_of_process,argc,argv);

// 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
```

1.2.6 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
```

1.2.7 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).

1.3 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
```

1.4 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.