# High Performance Fortran Interface to the Parallel C++ *

Shelby X. Yang, Dennis Gannon, Suresh Srinivas, Francois Bodin, and Paul Bode
Department of Computer Science, Indiana University
Bloomington, IN 47405
{yang,gannon,ssriniva,fbodin,bode}@rainier.cica.indiana.edu

## Abstract

*This paper describes the design of a Fortran interface to the parallel C++ (pC++) programming language. The pC++ Fortran interface provides a mechanism for users to link pC++ programs with Fortran subroutines so that they can take advantage of both the fast computing speed of Fortran and the object-oriented programming paradigm of C++. We will discuss the design of the Fortran interface and illustrate it with a galactic dynamics application. Our interface implementation is efficient. The galactic dynamics application can achieve a 14.4 Gflops sustained speed on a 512-node CM-5 massively parallel supercomputer.*

## 1 Introduction

One of the principal roles played by C++ in programming massively parallel computers is in the design of distributed data structures that are either too complex or simply inconvenient to program in Fortran. However, the mainstay of scientific and engineering programming is Fortran. The reasons for this are more than cultural. The restrictions in the Fortran type-system prohibit the use of pointers beyond the simple alias mechanisms allowed by Fortran90. Consequently, Fortran compilers can perform aggressive data-dependence-based optimizations which are not possible in C++. In addition, code generation technology and optimized intrinsic library functions present in Fortran are much better than their C++ counterparts. With the present generation of Fortran90 compilers, vector and array operations will be easy to compile into efficient code. We believe that it is essential to support a portable programming environment that mixes Fortran with C++ across the entire range of supercomputing platforms. [1]

There is no fundamental problem in linking Fortran to C++ programs. C++ has a standard facility for representing interfaces to other languages. However, with the founding of the High Performance Fortran Forum and the publication of the High Performance Fortran (HPF) [1] extensions to Fortran90, we have a new problem. In this paper, we explore and propose a mechanism for linking HPF programs to parallel C++ programs.

The paper is organized as follows. In section 2, we give a brief introduction to pC++. In section 3, 4, and 5, we discuss the design of the Fortran interface. In section 6, we discribe our experiments with a galactic dynamics application on the Thinking Machines CM-5 and the Intel Paragon. And finally in section 7, we draw our conclusions.

## 2 A brief introduction to pC++

pC++ [2] is an extension of C++ designed to allow programmers to compose "concurrent aggregate" classes which can be aligned and distributed over the memory of a parallel machine in a manner similar to the HPF compiler directives. pC++ allows users to write portable and efficient codes which can be run on a wide range of parallel computer systems. The first version of the compiler is a preprocessor which generates Single Program Multiple Data (SPMD) C++ code. Currently, it runs on the Thinking Machines CM-5, the Intel Paragon, the BBN TC2000, the Kendall Square Research KSR-1, the IBM SP-1, Parallel Virtual Machine (PVM) for workstation clusters, and the Sequent Symmetry. The basic concept behind

*This research is supported in part by ARPA under contract AF 30602-92-C-0135, the National Science Foundation Office of Advanced Scientific Computing under grant ASC-9111616, and the National Science Foundation and ARPA Grand Challenge Award, "The Formation of Galaxies and Large-scale Structure."

[1] This point of view is also supported in the research community. The soon to be published conclusions of the 1993 NSF/ARPA/DOE Grand Challenge Meeting include several strong recommendations for more investigation of mixing object-oriented style with the more traditional programming styles.

pC++ is the notion of a distributed data structure, called `Collection`, and parallel operations associated with it. A collection is a structured set of objects distributed across the processing elements of a computer. In a manner designed to be consistent with HPF, distribution of the objects in a collection over the processors is defined by programmers. pC++ provides a mechanism for building a "collection of objects" from a base *element* class. Member functions of this element class can be applied to the entire collection (or a subset) in parallel. This type of parallelism is similar to *data-parallelism*, though we call it *object-parallelism*. In addition, there is a mechanism for encapsulating the SPMD style in a thread based computing paradigm.

A base element class is a normal C++ class. For example, a class that represents a segment of a list of particles in an N-body simulation can be declared as

```
class Segment {
 public:
  double x[seg_nbodies],y[seg_nbodies],
         z[seg_nbodies],mass[seg_nbodies];
  Segment();
  void AccelerateParticles();
}
```

where `x`, `y`, and `z` represent the position of a particle in the $x$, $y$, and $z$ dimesions, and `mass` presents the mass of the particle. The member function `AccelerateParticles` accelerates the the particles according to the forces exerted on them.

A pC++ collection has the following component:

```
Collection NameOfCollection:ParentCollection
{
 private:
  <private data fields and functions>
 protected:
  <protected data fields and functions>
 public:
  <public data fields and functions>
 MethodOfElement:
  <data fields and functions
   added to each element>
};
```

The key points to note are:

- A collection declaration begins with the keyword `Collection` which is followed by the name of the collection and the name of the base collection the current collection inherits.
- A collection has private, protected, public data fields and member functions exactly as a normal C++ class. However, when a collection object is allocated, one copy of the collection structure is allocated in the memory of each processor. The member functions in a collection are invoked in the SPMD mode.
- A collection also has a set of data fields and member functions that will be copied into each base element class as public fields. These are labeled as `MethodOfElement` fields. The purpose of `MethodOfElement` fields is to allow each element to access or act on other elements in the collection. They are invoked in the object-parallel mode like other member functions in the base element class.

To construct a collection object one needs to create a processor object, a distribution object and an alignment object analogous to HPF. pC++ uses a special library class called `Processors` for the processor object. In the current implementation, it represents the set of all processors available to the program at run time. To build a distribution of some size, say 64, one can write

```
Processors P;
Distribution D(64,&P,CYCLIC);
```

where the $i$th distribution grid is mapped to processor `i%P`, because a cyclic distribution is used. To create an alignment object that defines the mapping of the collection elements to the distribution one writes

```
Align A(32,"[ALIGN(T[i],D[i])]");
```

where it is assumed that there are 32 elements in the collection. The collection constructor uses the alignment object to define the size and dimension of the collection. The mapping is described by a text string corresponding to the HPF alignment directive. It defines a mapping from a domain structure to a distribution structure using dummy index variables.

The intent of this two stage mapping, as it was originally conceived for HPF, is to allow the distribution coordinates to be a frame of reference so that different arrays could be aligned with each other in a manner that promotes memory locality.

With the distribution and alignment objects defined, a collection of `Segment` class can be constructed:

```
NameOFCollection&lt;Segment&gt; C(&D,&A);
```

and member function `AccelerateParticles` can be invoked in the object-parallel mode

```
C.AccelerateParticles();
```

## 3 Interfacing HPF and pC++

Let us consider the ways in which this interface might operate:

- HPF calling C++/pC++:

  *a.* A direct call to a C++ function from the main thread of a Fortran program. This presents no real problems that did not exist before, unless an HPF distributed array is passed as an argument.

  *b.* A C++ function is called as an HPF extrinsic function in a parallelized loop. Again this is only interesting if an HPF distributed array is passed as an argument.

  The HPF extrinsic procedure interface which allows calls to non-HPF subprograms as extrinsic procedures may well suffice for the above two situations provided the C++ subprogram is designed to be executed on a single processor. In both cases we assume that the C++ function will not attempt to return a value of any type other than a base type like `int` or `double`. It may be possible to view a returned C++ structure as a Fortran90 instance of a user defined type, but returning a C pointer would be a big problem.

- pC++ calling HPF:

  *a.* A call to a HPF parallel subroutine from a pC++ sequential main thread, *i.e.*, a pC++ main program.

  *b.* A call to a Fortran "node program", *i.e.*, a sequential or vector subroutine that is called from a parallel section of a pC++ program.

Of these we will focus on designing an interface for the case of pC++ calling Fortran. The data structure manipulation and encapsulation properties of C++ and the fast computing speed of Fortran make it natural for Fortran subroutines to be called as library routines and for C++ to act as the main controller.

Let us first consider case *a.* A pC++ element class which contains three arrays with two of the arrays being a special type `FArrayDouble` and one being a conventional C++ array is given below.

```
class MyElement {
  public:
   FArrayDouble x, y;
   double z[100];
   MyElement();
};
```

The special type `FArrayDouble` is used to contain double precision Fortran90 arrays and their descriptor structures. To define the constructor for the class we assume that `x` is a one dimensional array of a size of 100 and `y` is a two dimensional array of a size of $200 \times 50$.

```
MyElement::MyElement():x(100),y(200,50)
{...}
```

To use this class in a pC++ program that calls an HPF subroutine, we need to define the `Processors` object and to specify the size and shape of the collection. In the following example, a one dimensional collection of a size of 64 with elements of type `MyElement` is constructed and an HPF subroutine, `FFUN`, is called.

```
extern "HPF" void FFUN(FArrayDouble&,
               FArrayDouble&,double*);
main(){
 Processors P(64);
 Distribution D(64,&P,BLOCK);
 Align A(64,"[ALIGN(T[i],D[i])]");
 HPFCollection<MyElement> C(&D,&A);
 FFUN(C.x,C.y,C.z);
  ...
}
```

The important point to consider is what the passed arguments look like in terms of HPF distributed arrays. In this case, they can be defined by the HPF directives.

```
!HPF$ PROCESSORS P(64)
      double precision x(100*64),
     &  y(200*64,50),z(100*64)
!HPF$ DISTRIBUTE x(BLOCK),y(BLOCK,*) ONTO P
!HPF$ DISTRIBUTE z(BLOCK) ONTO P
```

It is relatively easy to see how the blocked decomposition corresponds to our simple collection of array blocks. When we have a two dimensional array of virtual processors, we would have a pC++ program as below:

```
main(){
 Processors P(64,32);
 Distribution D(64,32,&P,BLOCK,BLOCK);
 Align A(64,32,"[ALIGN(T[i][j],D[i][j])]");
 HPFCollection<MyElement> C(&D,&A);
 FFUN(C.x,C.y,C.z);
  ...
}
```

In this case, the HPF function would see these structures as defined by the following sequence of directives.

```
!HPF$ PROCESSORS P(64,32)
      double precision x(100*64,32),
     &  y(200*64, 50*32),z(100*64,32)
!HPF$ DISTRIBUTE x(BLOCK,BLOCK) ONTO P
!HPF$ DISTRIBUTE y(BLOCK,BLOCK) ONTO P
!HPF$ DISTRIBUTE z(BLOCK,BLOCK) ONTO P
```

The reader will note that we have not described how to create arrays that are distributed with anything other than a BLOCK distribution and array sizes are restricted to be multiples of the processor array size. There are ways around these restrictions, but they will not be discussed here.

## 4 Working with Connection Machines node Fortran on the CM-5

In this section we consider case *b* in which a Fortran node program is called from a parallel section of pC++. One of the problems in experimenting with HPF compilers is that only a few are now beginning to emerge. At the time of this writing, we do not have access to any of these systems, so we have conducted our experiments with Thinking Machines Co. CM Fortran (CMF) [3] on the CM-5. CMF is a reasonable approximation to the spirit of HPF.

There are three major issues that we want to address:

- Fortran array memory allocation,
- Fortran array element access, and
- Fortran array interprocessor communication.

As it can be seen later on in this section, the first two are addressed through new C++ classes, and the third through a new pC++ collection.

A CM-5 processing node consists of a Sparc microprocessor and four vector processor accelerators (vector units). To make use of these vector units, arrays have to be allocated in the memories of the vector units. The Connection Machine Run Time System (CMRTS) [4] provides functionalities for allocating space on the vector unit memories. However, the allocation is not so trivial since one also needs to be concerned about how array elements will be partitioned among the four vector units and being compatible with CMF. We use CMRTS function "CMRT_intern_detailed_geometry" to create a geometry and "CMRT_allocate_heap_array" to allocate memory. Once the arrays are allocated, array descriptors returned by "CMRT_allocate_heap_array" are passed to Fortran subroutines as arguments. Computations are done in Fortran subroutines. These Fortran arrays can also be directly manipulated by pC++ control programs through overloaded C++ operators.

We design a special C++ array class for each Fortran data type. The following class defines an array class for Fortran double precision arrays:

```
class FArrayDouble {
  // private part of class
 public:
  double *d;
  FArrayDouble() {d = 0.0;};
  FArrayDouble(int i);
  FArrayDouble(int i,int j);
  FArrayDouble(int i,int j,int k);
  FArrayDouble(int i,char layout1);
  FArrayDouble(int i,int j,char layout1,
   char layout2);
  FArrayDouble(int i,int j,int k,
   char layout1,char layout2,char layout3);
  ~FArrayDouble();
  double& operator()(int l);
  double& operator()(int l,int n);
  double& operator()(int l,int n,int m);
};
```

The private part of the class is used to store memory addresses of an array on the four vector units. The information will be used for accessing each individual array elements in pC++, and, in general, should not be of concern to the users. The class constructors that require arguments allocate memory on vector units when invoked. The arguments, i, j, and k specify the dimension sizes of the array and `layout1`, `layout2`, and `layout3` correspond to the CMF compiler directive for array layouts. A layout can be either "SERIAL" or "NEWS." A "NEWS" dimension means that this dimension will be distributed across the four vector units. A "SERIAL" dimension means this dimension will be packed into one vector unit's memory. An array with only "SERIAL" dimensions signals that it should be allocated in the Sparc chip's memory and thus is not allowed in the current implementation. Such an array can be allocated just like a normal C++ array which is always allocated in the Sparc chip's memory. The constructors that do not require the layout information use the default "NEWS" layout. The overloaded operators are used for accessing array elements.

The method we use to access array elements on the vector units from the Sparc chip differs from that of CMF. In the CMF assembly code, array element access involves computing "send addresses."

In our approach, we use "subgrid_dimension," "offchip_position," "subgrid_axis_increment," and "subgrid_size" provided by CMRTS to access each individual array element. It turns out that our method is as fast as CMF when all axes are declared "NEWS" and can be a few times faster than CMF when some axes are declared "SERIAL." For example, the speed for accessing a $64 \times 64$ double precision array is $5.68 \times 10^5$ bytes/second in pC++ and $3.81 \times 10^5$ bytes/second in CMF when the first dimension is declared "NEWS" and the second "SERIAL."

Besides memory allocation, we also provide methods for inter-element communication for the Fortran arrays. We design a special pC++ collection for this purpose:

```
Collection Fortran : public SuperKernel {
 public:
  Fortran(Distribution *D,Align *A);
  void GetFArray(int index,int offset,
                 FArrayDouble &buffer);
  void GetFArray(int index1,int index2,
       int offset,FArrayDouble &buffer);
};
```

The `Fortran` collection is derived from a special root collection called `SuperKernel`. The `SuperKernel` collection uses the distribution and alignment objects to initialize a collection and allocate the appropriate element objects on each processor's memory. The kernel also provides a global name space for the collection elements. The communication function `GetFArray` fetches an `FArray` array from a collection element denoted by the given index(es).

A Fortran subroutines is declared as a method of an element class. An invocation of a Fortran subroutine declared this way results in separate invocations of the Fortran subroutine on each processor. The Fortran subroutine is thus similar to a message-passing CMF "node program" though explicit message-passing is strongly discouraged in the Fortran subroutine. Communication between collection elements is accomplished in the pC++ control program. Since Fortran compilers usually add under scores to subroutine names, the pC++ compiler will need to generate a "wrapper function" for each Fortran subroutine.

By way of illustration, the following shows how a Fortran subroutine is called from pC++ utilizing the Fortran interface.

```
extern "C" {
 //Fortran subroutine is declared external
 void integrate_seg_(double*,double*,
                              int&,double&);
}
class Segment {
 public:
  double seg_sum;
  int length;
  FArrayDouble x, y;
  Segment();
  void integrate_seg();
};
Segment::Segment():x(length),y(length) {
  //FArrays are allocated in constructor
}
void Segment::integrate_seg() {
  // wrapper invokes Fortran subroutine
  integrate_seg_(x.d,y.d,length,seg_sum);
}
void main() {
  Processors P;
  Distribution D(64,&P,CYCLIC);
  Align A(64,"[ALIGN(T[i],D[i])]");
  Fortran&lt;Segment&gt; F(&D,&A);
  F.integrate_seg();
}
```

The Fortran subroutine is given below:

```
      subroutine integrate_seg(x,y,
     &   length,seg_sum)
      double precision seg_sum
      integer length
      double precision,array(length)::x,y
cmf   layout x(:news), y(:news)
      seg_sum = SUM(x*y)
      return
      end
```

In this example, the Fortran subroutine computes the dot-product between two one-dimensional arrays `x` and `y`. The `x` and `y` arrays are passed to the Fortran subroutine as "explicit-shape" arrays. However, the interface does not prevent us from declaring the arrays as "assumed-shape" arrays.

One shortcoming of the current Fortran interface implementation is that the pC++ control program cannot access Fortran COMMON blocks. This requires that all Fortran global variables be declared in the pC++ control program and passed to Fortran subroutines as arguments. Nevertheless, it does allow communication between Fortran subroutines through COMMON blocks. It is important to note, however, that pC++ programming model allows more than one pC++ collection element to be allocated on one processor, meaning Fortran subroutines called by different collection elements will access the same COMMON block. Data in the COMMON block are shared by elements on that processor. Potential race-condition may arise if a programmer is not careful. The best way to solve this problem is to allocate only one collection element on one processor.

# 5 Working with Intel node Fortran on the Paragon

We have also ported the Fortran interface to the Intel Paragon. Unfortunately, Fortran90 type compiler is not available on the Paragon, so we can only experiment with the Intel Fortran77 compiler. Since the Fortran interface is designed to be portable, most of what we have discussed for the CM-5 still applies for the Paragon. The pC++ program in the previous section can be directly ported to the Paragon without any change except that the layout specifications are ignored on the Paragon. FArrays become normal C++ arrays since there is no vector unit attached to the i860 processor. Nevertheless, the overloaded operators make array element access conceptually easier. The Fortran90 style subroutine has to be changed with vector expressions being converted to do-loops.

# 6 Expirement with a galactic dynamics application

We use a galactic dynamics application to test our Fortran interface. The application uses the self-consistent-field (SCF) method to evolve collisionless stellar systems [5]. The original code was written in Fortran77 by Lars Hernquist in 1991. In 1993, the code was converted to CMF by Greg Bryan. Expirements conducted on the 512-node CM-5 at the National Center for Supercomputing Applications (NCSA) by Greg Bryan indicate that with 10 million particles the CMF code can achieve 14.4 Gflops on 512 nodes of the CM-5 (private communication).

## 6.1 Brief summary of the SCF code

The algorithm used in the SCF code is described in detail in Hernquist and Ostriker 1992 [5]. The code solves the coupled Vlasov and Poisson equation for collisionless stellar systems using the N-body approximation approach. Poisson's equation for gravitational potential is solved by expanding the density $\rho$ and potential $\Phi$ in a set of basis functions. The basis set is constructed so that the lowest order members well-approximate a galaxy obeying the de Vaucouleurs $R^{1/4}$ projected density profile law.

The expansions of the density and potential take the following forms:

$$\rho(\vec{r}) = \sum_{nlm} A_{nlm}\rho_{nlm}(\vec{r}),$$

$$\Phi(\vec{r}) = \sum_{nlm} A_{nlm}\Phi_{nlm}(\vec{r}),$$

where $n$ is the radial "quantum" number and $l$ and $m$ are quantum numbers for the angular variables. Generally, the two sums will involve different expansion coefficients. But the assumption of bi-orthogonality ensures a one-to-one relationship between terms in the expansions for the density and potential. The basis sets $\rho_{nlm}$ and $\Phi_{nlm}$ also satisfy Poisson's equation:

$$\nabla^2\Phi_{nlm}(\vec{r}) = 4\pi\rho_{nlm}(\vec{r}).$$

The basis sets $\rho_{nlm}$ and $\Phi_{nlm}$ are given by

$$\rho_{nlm}(\vec{r}) = \frac{K_{nl}}{2\pi}\frac{r^l}{r(1+r)^{2l+3}}C_n^{2l+3/2}(\xi)\sqrt{4\pi}Y_{lm}(\theta,\phi),$$

$$\Phi_{nlm}(\vec{r}) = -\frac{r^l}{(1+r)^{2l+1}}C_n^{2l+3/2}(\xi)\sqrt{4\pi}Y_{lm}(\theta,\phi),$$

where $K_{nl}$ is a number related only to $n$ and $l$, and $C_n^{2l+3/2}(\xi)$ and $Y_{lm}(\theta,\phi)$ are ultrasperical polynomials and spherical harmonics, respectively. After some algebra, the expansion coefficients become

$$A_{nlm} = \frac{1}{I_{nl}}\sum_k m_k[\Phi_{nlm}(r_k,\theta_k,\phi_k)]^*,$$

where $I_{nl}$ is a number and $m_k$ is the mass of the $k$th particle. Once the gravitational potential is found, the gravitational force per unit mass can be obtained by taking the gradient of the potential and the particles can be accelerated accordingly.

## 6.2 The pC++/Fortran Version of the SCF Code

We design a C++ class called `Segment` to represent a subgroup of the N particles used in the simulation. As we have discussed earlier, the major procedure in the SCF code is to compute the sums for the expansion coefficients $A_{nlm}$. Our approach is to first compute local sums within each `Segment` object. After this, global sums are formed by a global reduction. The global sums are then broadcast back to each `Segment` object where the particles are accelerated by the gravitational force. Fortran subroutines in the original Fortran code now become member functions of the Segment class, although subroutines involving inter-element communication and I/O need to be modified. The Segment class is declared as follows with many less important variables and member functions omitted:

```
class Segment {
 public:
  FArrayDouble x,y,z,vx,vy,vz,ax,ay,az,mass,
               plm,clm,dlm,elm,flm,dplm;
  double sinsum[lmax+1][lmax+1][nmax+1],
         cossum[lmax+1][lmax+1][nmax+1];
  Segment();
  void compute_polynomial();
  void compute_acceleration();
  void update_position();
  void update_velocity();
};
```

In the above declaration, x, y, z, vx, vy, vz, ax, ay, az, mass are one-dimensional arrays that contain the positions, the velocities, the accelerations, and the masses of particles belonging to a Segment object, and plm, clm, dlm, elm, flm, dplm contain the expansion coefficients and the values of the polynomial. The sinsum and cossum contain the local sums and eventually the global sums of expansion coefficients. They are serial arrays on the CM-5 and are allocated in the Sparc chip's memory. Among the class member functions (Fortran subroutines), compute_polynomial computes the polynomials and local sums, compute_acceleration computes the acceleration for each particle, update_position and update_velocity update the positions and velocities of particles.

The collection that distributes the elements, allocates memory, and manages inter-element communication is declared as below. Again, many less important member functions are omitted for brevity:

```
Collection SelfConsistField:public Fortran {
 public:
  SelfConsistField(Distribution *D,Align *A);
  void InParticles();
  void InParameters();
  void OutParticles(int nsnap);
 MethodOfElement:
  virtual void compute_polynomial();
  virtual void compute_acceleration();
  virtual void update_position();
  virtual void update_velocity();
  void read_segment();
  void write_segment();
};
```

Functions declared here are pC++ functions. Their main functionalities are to handle the I/O and inter-element communication. InParticles, InParameters, and OutParticles read the input files and write to the output files, and read_segment and write_segment are called by InParticles and OutParticles to perform parallel I/O on both the CM-5 and Paragon. Functions that are already defined in element class Segment are declared as virtual functions in this collection declaration.

The main program is listed below:

```
void Processor_Main() {
 Processors P;
 Distribution D(elem_count,&P,BLOCK);
 Align A(elem_count,"[ALIGN(X[i],D[i])]");
 SelfConsistField&lt;Segment&gt; X(&D,&A);
 // read initial model
 X.InParameters();
 X.InParticles();
 X.compute_polynomial();
 X.ReduceDoubleAdd(offset,variable_count);
 X.compute_acceleration();
 // main loop
 for(n = 1; n <= nsteps; n++) {
  X.update_position();
  X.compute_polynomial();
  X.ReduceDoubleAdd(offset,variable_count);
  X.compute_acceleration();
  X.update_velocity();
  X.OutParticles(n);
 }
}
```

where ReduceDoubleAdd is a reduction function inherited from SuperKernel. offset is measured from the beginning of the class Segment to the beginning of the field sinsum, and variable_count is the total number of array elements is sinsum and cossum. A leapfrog integration scheme is used to advance particles.

### 6.3 Timing results

Our experiments with the pC++/CMF SCF code was conducted on the 32-node CM-5 at the University of Maryland and the 88-node Paragon at Indiana University. For comparison, we also ran the CMF SCF code on the CM-5. The results of these experiments are listed in Table 1.

| CPU time in seconds spent in main loop without I/O. | | | |
|---|---|---|---|
| No. of Particles | pC++/CMF on CM5 | CMF on CM5 | pC++/F77 on Paragon |
| 512 | 5.4 (76.6) | 6.6 (40.9) | 5.8 |
| 5120 | 8.6 (75.1) | 9.9 (40.9) | 25.0 |
| 51200 | 45.8 (89.1) | 50.3 (57.4) | 332.5 |
| Total CPU time in seconds with parallel I/O | | | |
| 51200 | 49.1 (89.1) | 58.2 (57.4) | 396.3 |

Table 1: *Execution time in seconds for 100 time steps on 32-node partitions with the expansions truncated at* $nmax = 6$ *and* $lmax = 4$. *Numbers in parentheses are amount of memory used in Mbytes. The total time includes reading the input model (2.87 Mbytes), initializing the system, evolving the system for 100 evolution steps, and writing two output files (2.87 Mbytes and 2.46 Mbytes).*

As can be seen in Table 1, the pC++/CMF SCF code is, in fact, slightly faster than the CMF SCF code in all cases. This can be explained by the fact that the pC++/CMF code is able to use a faster global vector reduction function. The timing results show that our Fortran interface implementation is very efficient. Given the results shown in Table 1 and experiments conducted on NCSA's CM-5 using larger partition sizes, we conclude that the pC++/CMF SCF code can achieve at least the 14.4 Gflop speed of the CMF SCF code. However, the memory overhead in pC++ is quite large, as the pC++/CMF code requires on an average about 1.8 times of memory to execute. The slow speed on the Paragon is due to the fact that no optimization was done by Intel Fortran77 compiler. For the Paragon porting, we simply translated Fortran90 vector expressions back to do-loops. If we use Kuck & Associates math library [6], which is optimized for the i860 processor, we can expect to gain more speedup. But using the Kuck & Associates math library would make the Fortran77 code less portable to other machines.

## 7 Summary and conclusions

In this paper, we have discussed the design of a Fortran interface to the parallel C++ (pC++) programming language. Our prototype Fortran interface which allows linking pC++ control programs to CM Fortran and Fortran77 subroutines proves to be efficient. Our experiments demonstrate that our portable mix of Fortran and C++ enables programmers to write efficient programs, and, at the same time, utilize both the fast vector and array computation properties of Fortran and the data structure manipulation and encapsulation properties of C++.

One of the shortcomings of our current implementation is that Fortran COMMON blocks cannot be accessed by pC++ control programs. Communication between Fortran subroutines through COMMON blocks is allowed, though potential race-condition may arise because pC++ programming model allows more than one pC++ collection element to be allocated on one processor. In addition, our current compiler cannot generate the wrapper functions required for Fortran subroutines. For the moment, the wrapper functions have to be hand-coded. The CM Fortran compiler layout directive also allows a communication weight to be assigned to an array axis. This feature is not implemented in our current Fortran interface design. We plan to add it in the future.

Efforts are under way to port pC++ and the Fortran interface to Cray T3D. We are studying Cray MPP Fortran specifications and experimenting with different Fortran interface designs for the T3D.

## References


[1] D. Loveman, "High Performance Fortran," *IEEE Parallel and Distributed Technology*, Vol. 1, 1993, pp. 25–42.

[2] D. Gannon, S. X. Yang, and P. Beckman, "User Guide for a Portable Parallel C++ Programming System, pC++," 1994. Available from moose.cs.indiana.edu:/pub/sage by ftp.

[3] Thinking Machines Corporation, "CM Fortran Reference Manual," 1992.

[4] Thinking Machines Corporation, "CMRTS Architectural Secification Version 0.85," 1993.

[5] L. Hernquist and J. P. Ostriker, "A Self-Consistent Field Method for Galactic Dynamics," *The Astrophysical Journal*, Vol. 386, 1992, pp. 375-397.

[6] Kuck & Associates, Inc., "CLASSPACK Basic Math Library User's Guide," 1992.