To appear in: Proceedings of the Supercomputing '93 Conference, Portland, Oregon, November 15–19, 1993.

Implementing a Parallel C++ Runtime System for Scalable Parallel Systems[1]

F. Bodin
Irisa
University of Rennes
Rennes, France
Francois.Bodin@cs.irisa.fr

P. Beckman, D. Gannon, S. Yang
Dept. of Comp. Sci.
Indiana University
Bloomington, Indiana 47405
{beckman,gannon,yang}@cs.indiana.edu

S. Kesavan, A. Malony, B. Mohr
Dept. of Comp. and Info. Sci.
University of Oregon
Eugene, Oregon 97403
{kesavans,malony,mohr}@cs.uoregon.edu

Abstract

pC++ is a language extension to C++ designed to allow programmers to compose "concurrent aggregate" collection classes which can be aligned and distributed over the memory hierarchy of a parallel machine in a manner modeled on the High Performance Fortran Forum (HPFF) directives for Fortran 90. pC++ allows the user to write portable and efficient code which will run on a wide range of scalable 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, and the Sequent Symmetry. In this paper we describe the implementation of the runtime system, which provides the concurrency and communication primitives between objects in a distributed collection. To illustrate the behavior of the runtime system we include a description and performance results on four benchmark programs.

1 Introduction

pC++ permits programmers to build distributed data structures with parallel execution semantics. For "distributed memory" machines, with a non-shared address space, the runtime system implements a shared name space for the objects in a distributed collection. This shared name space is supported by the underlying message passing system of the target machine. In the case of "shared memory" architectures, the runtime system uses the global addressing mechanism to support the name space. A thread system on the target machine is used to support the parallel tasks.

After a short introduction to pC++ we give a detailed description of each runtime system. To illustrate the behavior of the runtime system we include performance results for four benchmark programs.

2 A Brief Introduction to pC++

The basic concept behind pC++ is the notion of a distributed collection, which is a type of concurrent aggregate "container class" [7, 9]. More specifically, a *collection* is a structured set of objects distributed across the processing elements of the computer. In a manner designed to be completely consistent with HPF Fortran, the programmer must define a distribution of the objects in a collection over the processors and memory hierarchy of the target machine. As HPF becomes more available, future versions of the pC++ compiler will allow object level linking between distributed collections and HPF distributed arrays.

A collection can be an *Array*, a *Grid*, a *Tree*, or any other partitionable data structure. Collections have the following components:

- A collection class describing the basic topology of the set of elements.
- A size or shape for each instance of the collection class; e.g., array dimension or tree height.
- A base type for collection elements. This can be any C++ type or class. For example, one can define an Array of *Float*s, a Grid of *FiniteElement*s, a Matrix of *Complex*, or a Tree of *X*s, where *X* is the class of each node in the tree.
- A *Distribution* object. The distribution describes an abstract coordinate system that will be distributed over the available "processor objects" of the target by the runtime system. (In HPF [8],

[1] This research is supported by ARPA under Rome Labs contract AF 30602-92-C-0135, the National Science Foundation Office of Advanced Scientific Computing under grant ASC-9111616, and Esprit under the BRA APPARC grant.

the term *template* is used to refer to the coordinate system. We will avoid this so that there will be no confusion with the C++ keyword *template*.)

- A function object called the *Alignment*. This function maps collection elements to the abstract coordinate system of the *Distribution* object.

The pC++ language has a library of standard collection classes that may be used (or subclassed) by the programmer [10, 11, 12, 13]. This includes collection classes such as *DistributedArray*, *DistributedMatrix*, *DistributedVector*, and *DistributedGrid*. To illustrate the points above, consider the problem of creating a distributed 5 by 5 matrix of floating point numbers. We begin by building a *Distribution*. A distribution is defined by its number of dimensions, the size in each dimension and how the elements are mapped to the processors. Current distribution mappings include *BLOCK*, *CYCLIC* and *WHOLE* , but more general forms will be added later. For our example, let us assume that the distribution coordinate system is distributed over the processor's memories by mapping *WHOLE* rows of the distribution index space to individual processors using a *CYCLIC* pattern where the i^{th} row is mapped to processor memory $i \bmod P$, on a P processor machine.

pC++ uses a special implementation dependent library class called *Processors*. 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 7 by 7, with this mapping, one would write

```
Processors P;
Distribution myDist(7,7,&P,CYCLIC,WHOLE);
```

Next, we create an alignment object called `myAlign` that defines a domain and function for mapping the matrix to the distribution. The matrix `A` can be defined using the library collection class *DistributedMatrix* with a base type of *Float*.

```
Align myAlign(5,5,"[ALIGN(domain[i][j],
                          myMap[i][j])]");
DistributedMatrix<Float> A(myDist,myAlign);
```

The collection constructor uses the alignment object, `myAlign`, to define the size and dimension of the collection. The mapping function is described by a text string corresponding to the HPF alignment directives. 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.

2.1 Processors, Threads, and Parallelism

The processor objects used to build distributions for collections represent a set of threads. Given the declaration

```
Processors P;
```

one thread of execution is created on each processor of the system that the user controls. These new *processor object* (PO) threads exist independent of the main program control thread. (In the future, pC++ will allow processor sets of different sizes and dimensions.) Each new PO thread may read but not modify the "global" variables; i.e., program static data or data allocated on the heap by the main control thread. Each PO thread has a private heap and stack.

Collections are built on top of a more primitive extension of C++ called a *Thread Environment Class*, or `TEClass`, which is the mechanism used by pC++ to ask the processor object threads to do something in parallel. A `TEClass` is declared the same as any other class with the following exceptions:

- There must be a special constructor with a *Processors* object argument. Upon invocation of this constructor, one copy of the member field object is allocated to each PO thread described by the argument. The lifetime of these objects is determined by their lifetime in the control thread.
- A `TEClass` object may not be allocated by a PO thread.
- The `()` operator is used to refer to a single thread environment object by the control thread.
- A call to a `TEClass` member function by the main program control thread represents a transfer of control to a parallel action on each of the threads associated with the object. (Consequently, member functions of the `TEClass` can read but cannot modify global variables.) The main control thread is suspended until all the processor threads complete the execution of the function. If the member function returns a value to the main control thread, it must return the same value from each PO thread or the result is undefined.
- If a `TEClass` member function is invoked by one of the processor object threads, it is a sequential action by that thread. (Hence, there is no way to generate nested parallelism with this mechanism.)

These issues are best illustrated by an example.

```
int x;          // c++ global
float y[1000]; // c++ global

TEClass MyThreads{
   int id;           // private thread data
 public:
   float d[200];    // public thread data
   void f(){id++;}  // parallel functions
   int getX(int j){return x;}
};

main() {
 Processors P;   // the set of processors
 MyThreads T(P); // implicit constructor
                 // one thread object/proc.
 // a serial loop
 for(int i=0; i<P.numProcs(); i++)
   T(i).id=i; // main control thread can
              // modify i-th thread env.
 T.f(); // parallel execution on each thread
 // an implicit barrier after parallel call
}
```

In this example, the processor set `P` is used as the parameter to the thread environment constructor. One copy of the object with member field `id` is allocated to each PO thread defined by `P`. The lifetime of `T` is defined by the main control thread in which it was created. (However, in the current implementation the storage is not automatically reclaimed.) Figure 1 illustrates the thread and memory model that the language provides.

The main control thread can access and modify the public member fields of the `TEClass` object. To accomplish this, one uses the `()` operator, which is implicitly overloaded. The reference `T(i).id` refers to the `id` field in the i^{th} `TEClass` object. Note that the value of the expression `T.id` within the main control thread may not be well defined because each thread may have a different value for `id`. However the assignment `T.id = 1` is valid and denotes an update to all members named `id`.

An individual PO thread cannot modify the local fields of another PO thread, but it can access them by means of the `()` operator. The only other way for PO threads to communicate is by means of native system message passing, but this is not encouraged until a C++ binding for the standard message passing interface is defined.

The call `T.f()` indicates a branch to a parallel operation on each PO thread. After the parallel execution

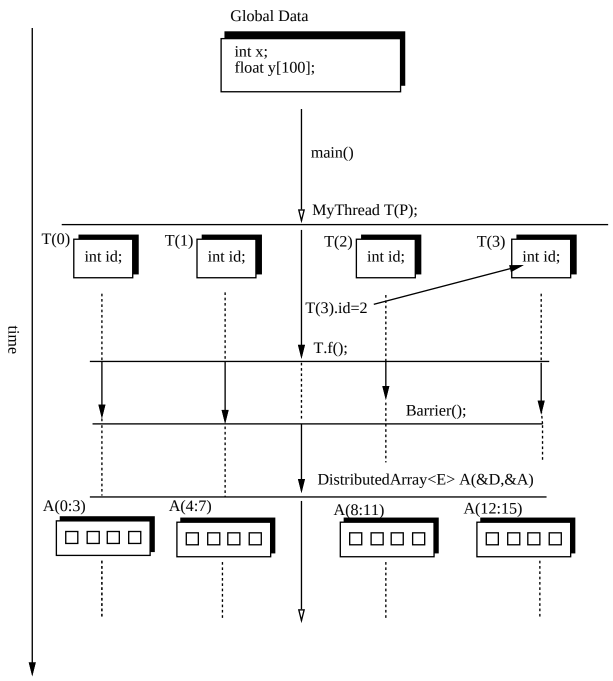

Figure 1: TEClass Objects, Collections and Processor Threads

of the method, there is a barrier synchronization before returning to the main control thread. In the case of invoking an object such as `T.getX()`, which has a non-void return value, it is essential that the function returns an identical value for each main PO thread for a given main thread invocation.

Note that the `TEClass` mechanisms provide a simple and direct way to "wrap" message passing SPMD style C++ routines inside a standard sequential main program. In this way, many of the special libraries of parallel C++ code already designed can be easily integrated into this model [16, 17].

2.2 Collections and Templates

The C++ language provides the *templates* mechanism to build parameterized classes. Collections are very similar to templates defined over `TEClass`es. In fact, it is *almost* sufficient to describe a collection class as

```
template <class ElementType>
TEClass MyCollection: Kernel<ElementType> {
  MyCollection(Distribution &D,Align &A)::
    Kernel(D,A) { ...}
      ...
};
```

where Kernel is a super-template class that defines the concurrent aggregate structure of the collection. Indeed, as will be shown in the following sections of this paper, it is the structure of Kernel and its basic constructors and member functions that is at the core of the runtime system for pC++.

While the construction above is nearly sufficient to define collections, it does not give us everything we want. In particular, collections define a generalization of data parallelism as follows. Let C be a collection type and E an element type. Assume E has the form

```
class E {
  int a;
  void f();
  E & operator +(E&);
};
```

and let x and y be declared to be of type C<E>. Because + is defined on the elements, one can write x+y and this means the "pointwise" addition of the elements and the result is a new collection of type C<E>. In a similar manner the expression x.a + y.a is a new collection of type C<int>. In addition, the expression x.f() means the parallel application of the element member function f to each element of the collection. These operations, together with collection specific reductions form the main parallel control in pC++.

To accomplish this we provide a special form of the C++ template construct with a distinguished class parameter, called ElementType, which defines the type of the element in the collection. The exact syntax is shown below:

```
Collection<class ElementType>
TEClass CollName: ParentCollection {
  public:
   CollName(Distribution &D, Align &A);
  private:
  protected:
   // TEClass member functions and fields.
   // Methods declared here are executed in
   // parallel by the associated PO thread.
  MethodOfElement:
   // Fields declared here are added to each
   // element, Methods to the element class.
}
```

Data fields defined in the public, private and protected areas are components of the underlying TEClass object. The size and shape of the collection and the way in which the elements of the collection are distributed to the processor object threads is defined by the Distribution and Alignment objects supplied to the constructor for the collection. The set of elements that are mapped to a single PO thread object is called the "local collection" for that thread. The data structure that contains and manages these elements is part of the Kernel class which is a required ancestor of any collection.

2.3 Communication Between Elements in a Collection

One of the key ideas behind collection classes is to enable the programmer to build a distributed data structure where data movement operators can be closely linked to the semantics of the data type. For example, elements of a *DistributedGrid* need to access their neighbors. A *Tree* node will often reference its children or parent. Each application has a topology of communication and global operators that must be supported efficiently by the collection structure.

If c is a collection of type C<E>, then any processor thread may access the i^{th} element of c by the Kernel functions

```
c->Get_Element(i);
c->Get_ElementPart(i, offset, size);
```

The first function accesses the i^{th} element and places a copy in a local system buffer and returns a pointer of type (ElementType *). The second function accesses part of an element at a given offset and size and make a copy in the buffer. In other collections, such as distributed arrays, matrices and grids, the operator (...) has been overloaded. For example, if c is a two dimension collection, then expressions like

```
x = c(i,j) + c(i+1,j)
```

work by calling the Get_Element function if a reference is not to a local element.

3 The pC++ Runtime System

Get_Element and Get_ElementPart are the only communication primitives in pC++ that allow processor threads to access the shared name space and remote elements. Notice that this differs significantly from the "message passing" style that is common in SPMD programming. In the latter model, all synchronization is accomplished by matching a send operation with a corresponding receive. In pC++, any processor object thread may read any element of any collection, but only the owner object thread may modify the element; this is equivalent to the "owner computes" semantics found in HPF. All synchronization is in terms of barrier operations that terminate each collection operation.

The runtime system for pC++ must manage three distinct tasks.

1. *The allocation of collection classes.* This involves the interpretation of the alignment and distribution directives to build the local collection for each processor object.

More specifically, each processor object must have a mechanism whereby any element of the collection can be identified. In a shared address space environment, this may be a table of pointers or a function that computes the address of an element. In the non-shared address space model, this may be the name of a processor that either has the object or knows where to find it. Depending upon the execution model of the target, this task may also involve the initialization of threads associated with each processor object.

2. *The management of element accesses.* In particular, access management requires an effective, efficient implementation of `Get_Element` and `Get_ElementPart` functions.

This activity can be seen as a compiler-assisted "local caching" of a remote element to the local processor thread's address space. In a shared address space environment, alternative memory management schemes may be used to improve memory locality. If there is no shared address space, these functions require a "one-sided" communication protocol – if processor X needs an element from processor Y, processor X must wake up an agent that has access to the address space of Y which can fetch the data and return it to X.

3. *Termination of parallel collection operations.* All parallel collection operations are barrier synchronized before returning to the main thread.

Note that only the processor objects involved in the computation of the collection operation must be synchronized and not every processor in the system need be involved. However, as we shall see, this may be required for some implementations.

Some restrictions are imposed by the current pC++ compiler that are important to note for the runtime system implementation. The current version of the pC++ compiler generates SPMD code in which the set of processor objects for each collection is the same as the set of processors in the user's execution partition. There is one execution thread per processor and, on one processor, all local collection objects share this thread. In true SPMD style, the main thread is duplicated and is run with the single local thread on each processor.

This model of execution is consistent with all current HPF implementations, but imposes some limitations on the programmer. For example, it is not possible to have two different collection operations going on in different subsets of processors concurrently. Furthermore, this limits the system from having nested concurrency by building collections of collections. Even with these limitations this is still a powerful programming model. These limitations will be removed in future versions of the compiler.

In the paragraphs that follow, we will look at the shared address and distributed address space implementations of the current pC++ execution model.

3.1 Distributed Memory Systems

Distributed memory parallel systems consist of a set of processing nodes interconnected by a high-speed network. Each node consists of a processor and local memory. In the case of a non-shared, distributed memory system, each processor only has access to its local memory and a message system is used to move data across the network between processors.

One common approach to building a shared memory system on top of a non-shared, distributed memory computer is called *shared virtual memory* (SVM). In SVM, the message passing system is used to move pages from one processor to another in the same way a standard VM system moves pages from memory to disk. Though these systems are still experimental [14, 15], they show great promise for providing a support environment for shared memory parallel programming models. Because pC++ is based on a shared name space of collection elements, we use SVM techniques to build the runtime system for the Intel Paragon and the Thinking Machines CM-5.

More specifically, our model is based on ideas from Koan[15]. The basic idea is that each collection element has a *manager* and a *owner*. The owner of the element is the processor object that contains the element in its local collection. As with pages in an SVM, we assume that an element may be moved from one local collection to another at run time, for load balancing reasons, or, in the case of dynamic collections, may be created and destroyed at run time. In other words, we assume that the distribution map may be dynamic. Although this dynamic feature of the system is not being used by the current compiler,[2] it is a design

[2]It will be supported in future releases.

requirement for the runtime system implementation. The purpose of the element manager is to keep track of which processor object owns the element. The manager is static and every processor thread knows how to find the manager. Elements are assigned managers by a simple cyclic distribution. The algorithm for implementing the function `Get_Element(i)` is given as follows:

1. Let p be the processor thread that is executing `Get_Element(i)` for an element i of some collection. Let P be the number of processors. The index m of the manager processor thread is given by `m = i mod P`. Thread p sends a message to m requesting the identity of the owner.
2. The manager q is "interrupted" and looks in a table for the owner o of element i and sends this value to p.
3. The requester p then sends a message to o asking for element i.
4. The owner o is "interrupted" and sends a copy of element i to the requester p.

Hence, the primary implementation issues for a given machine reduce to:

- How is a processor interrupted when a request for an element or owner is received?
- How is the table of owner identifiers stored in the manager?
- How is the barrier operation implemented?

The current pC++ compiler assumes no mechanism exists for interrupting a processor thread. Instead, the compiler generates calls to a function called `Poll()` in the element and collection methods. By calling `Poll()`, a thread can check a queue of incoming messages and reply to requests in a reasonably timely manner. Unfortunately, calling `Poll()` periodically is not sufficient to prevent starvation. If no interruption mechanism exists on the target, it is necessary to make sure that the `Barrier()` function also calls `Poll()` while waiting for the barrier to complete.

The final issue to consider in the runtime environment is that of allocating the collection elements. In the current version a large table is created in each processor object that stores pointers to the local collection. A second table in each processor object stores the index of the owner of each element that is managed by that processor. Because all of the elements and both tables can be allocated and created based on the distribution and alignment data, it is straightforward to parallelize this task in the distributed memory environment. (This is in contrast to the shared memory situation where synchronization is necessary to ensure that each processor has access to pointers to all the elements.)

3.1.1 The Thinking Machines CM-5

The CM-5 is a distributed memory machine based on a fat tree network. Each processor is a Sparc CPU together with four vector units. (In the experiments described here the vector units are not used.) The basic communication library is CMMD 3.0 Beta which is based on a re-implementation of the Berkeley CMAM Active Message layer [5, 6]. The active message layer provides very good support for short messages that consist of a pointer to a local function to execute and three words of argument. In addition, the system can be made to interrupt a processor upon the receipt of a message, or it can be done by polling. One can switch between these two modes at run time. Our experience indicates that using a combination of polling and interrupts works the best. During barrier synchronization, the interrupt mechanism is used. The CMMD barrier operation is very fast (4 microseconds).

3.1.2 The Intel Paragon

The Intel Paragon is a distributed memory machine based on a grid network. Each processor contains two i860s. One i860 is used to run the user code and one handles the message traffic and talks to the special mesh router. (Unfortunately, our testbed Paragon system is running "pre-release" software which only uses one of the i860s.) The basic communication library is the NX system that has been used for many years on the Intel machines. NX only provides a very primitive interrupt driven message handler mechanism; consequently, only the polling strategy can be used. Furthermore, NX is not well optimized for very short messages, such as locating the owner of an element. In addition, implementing a barrier function that must also poll for messages is non-trivial and results in slow operation. Barrier execution takes approximately 3 milliseconds. However, the native NX barrier which does not do polling is not much faster (about 2 milliseconds). Combined with the effect of pre-release software, the performance of the pC++ runtime system on the Intel Paragon is non-optimal.

3.2 Shared Memory Systems

There are three main implementation differences in the pC++ runtime system on a shared memory versus a distributed memory machine. The most obvious difference is that message communication is not required for accessing remote collection elements. All collection elements can be accessed using address pointers into the shared memory space. A related difference is that collection element tables need only be allocated once, since all processors can directly reference tables using their base address. However, it may be beneficial to have multiple copies of the tables to improve memory locality during `Get_Element` operations. In contrast, it is necessary to have a separate collection element table on each processor node in a distributed memory machine. The third difference is in how collections are allocated. In a distributed memory machine, the owner of elements of a collection allocates the space for those elements in the memory of the processor where it (the owner process) will execute. In a shared memory machine, the space for an entire collection is allocated out of shared memory space. Care must be taken in memory allocation to minimize the contention between local processor data (i.e., the data "owned" by a processor) and remote data. Achieving good memory locality in a shared memory system, using processor cache or local memory, will be important for good performance.

3.2.1 General Strategy

The current pC++ runtime system that we have implemented for shared memory machines has the following general, default properties:

- Collection element tables: Each processor has its own copy of the element table for each collection.
- Collection allocation: Each processor object allocates all the space for its local elements. The processor objects then exchange local element addresses to build the full collection element table.
- Barrier synchronization: The barrier implementation is chosen from optimized hardware/software mechanisms on the target system.

3.2.2 The BBN TC2000

The BBN TC2000 [1] is a scalable multiprocessor architecture which can support up to 512 computational nodes. The nodes are interconnected by a variant of a multistage cube network referred to as the *butterfly switch*. Each node contains a 20 MHz Motorola 88100 microprocessor and memory which can be configured for local and shared access. The contribution of each node to the interleaved shared memory pool is set at boot time.

The parallel processes are forked one at a time via the nX system routine `fork_and_bind`. This routine creates a child process via a UNIX `fork` mechanism and attaches the child to the specified processor node. The collection element tables and local collection elements are allocated in the local memory space on each node of the TC2000. There are several choices under nX for allocating collection elements in shared memory: across node memories (e.g., interleaved or random) or on a particular node's memory with different caching policies (e.g., uncached or cached with copy-back or write-through cache coherency). Currently, the TC2000 pC++ runtime system allocates collection elements in the "owner's" node memory with a write-through caching strategy. The TC2000 does not have special barrier synchronization hardware. Instead, we implemented the logarithmic barrier algorithm described in [2]. Our implementation requires approximately 70 microseconds to synchronize 32 nodes. This time scales as the log of the number of processors.

3.2.3 The Sequent Symmetry

The Sequent Symmetry [3] is a bus-based, shared memory multiprocessor that can be configured with up to 30 processors. The Symmetry architecture provides hardware cache consistency through a copy-back policy and user-accessible hardware locking mechanisms for synchronization. For our prototype implementation, we used a Symmetry S81 machine with 24 processors (16 MHz Intel 80386 with a Weitek 1167 floating point coprocessor) and 256 MBytes of memory across four memory modules interleaved in 32 byte blocks.

Using Sequent's parallel programming library, the implementation of the pC++ runtime system was straightforward. Because all memory in the Sequent machine is physically shared in the hardware, the local allocation of the collection element tables on each processor is only meaningful relative to the virtual memory space of the process. All collection element tables are allocated in the local data segment of each process, making them readable only by the process that created them. In contrast, collection elements must be allocated in a shared segment of the virtual address space of each process; a shared memory allocation routine is used for this purpose. Unfortunately, there is no way to control the caching policy in software; copy-back is the hardware default. Barrier synchronization is implemented using a system-supplied barrier routine which takes advantage of the hardware locking facilities of the Sequent machine. It is very efficient – the barrier performance on 8, 12, 16, and 20 processors is 34, 47, 58, and 70 microseconds, respectively.

3.2.4 The Kendall Square Research KSR-1

The KSR-1 is a shared virtual memory, massively-parallel computer. The memory is physically distributed on the nodes and organized as a hardware-coherent distributed cache [4]. The machine can scale to 1088 nodes, in clusters of 32. Nodes in a cluster are interconnected with a pipelined slotted ring. Clusters are connected by a higher-level ring. Each node has a superscalar 64-bit custom processor, a 0.5 Mbyte local sub-cache, and 32 Mbyte local cache memory.

For the pC++ runtime system implementation, we used the POSIX thread package with a KSR-supplied extension for barrier synchronization. The collection allocation strategy is exactly the same as for the Sequent except that no special shared memory allocation is required; data is automatically shared between threads. However, the hierarchical memory system in the KSR is more complex than in the Sequent machine. Latencies for accessing data in the local sub-cache and the local cache memory are 2 and 18 cycles, respectively. Latencies between node caches are significantly larger: 150 cycles in the same ring and 500 cycles across rings. Although our current implementation simply calls the standard memory allocation routine, we suspect that more sophisticated memory allocation and management strategies will be important in optimizing the KSR performance.

4 Performance Measurements

To exercise different parallel collection data structures and to evaluate the pC++ runtime system implementation, four benchmark programs covering a range of problem areas were used. These benchmarks are briefly described below. The results for the benchmarks on each port of pC++ follow.

BM1: Block Grid CG. This computation consists of solving the Poisson equation on a 2-dimensional grid using finite difference operators and a simple conjugate gradient method without preconditioning. It represents one type of PDE algorithm.

BM2: A Fast Poisson Solver. This benchmark uses FFTs and cyclic reductions along the rows and columns of a two dimensional array to solve PDE problems. It is typical of a class of CFD applications.

BM3: The NAS Embarrassingly Parallel Benchmark. Four NAS benchmark codes have been translated to pC++; we report on two. The BM3 program generates 2^{24} complex pairs of uniform (0, 1) random numbers and gathers a small number of statistics.

BM4: The NAS Sparse CG Benchmark. A far more interesting benchmark in the NAS suite is the random sparse conjugate gradient computation. This benchmark requires the repeated solution to $A * X = F$, where A is a random sparse matrix.

4.1 Distributed Memory Systems

The principal runtime system factors for performance on non-shared, distributed memory ports of pC++ are message communication latencies and barrier synchronization. These factors influence performance quite differently on the TMC CM-5 and Intel Paragon. For the CM-5, experiments for 64, 128, and 256 processors were performed. Because of the large size of this machine relative to the others in the paper, we ran several of the benchmarks on larger size problems. For the BM1 code running on a 16 by 16 grid with 64 by 64 sub-blocks, near linear speedup was observed, indicating good data distribution and low communication overhead relative to sub-block computation time. Execution time for BM2 is the sum of the time for FFT transforms and cyclic reduction. Because the transforms require no communication, performance scales perfectly here. In contrast, the cyclic reduction requires a communication complexity that is nearly equal to the computational complexity. Although the communication latency is very low for the CM-5, no speedup was observed in this section even for Poisson grid sizes of 2,048. For the benchmark as a whole, a 25 percent speedup was observed from 64 to 256 processors. As expected, the BM3 performance showed near linear speedup. More importantly, the execution time was within 10 percent of the published manually optimized Fortran results for this machine. For the BM4 benchmark, we used the full problem size for the CM-5. While the megaflop rate is low, it matches the performance of the Cray Y/MP un-tuned Fortran code.

Results for the Paragon show a disturbing lack of performance in the messaging system, attributed primarily to the pre-release nature of this software. Experiments were performed for 4, 16, and 32 processors. The BM1 benchmark required a different block size choice, 128 instead of 64, before acceptable speedup performance could be achieved, indicative of the effects of increased communication overhead. At first glance, the speedup improvement from BM2 contradicts what was observed for the CM-5. However, using a smaller number of processors, as in the Paragon case, has the effect of altering the communications / computation ratio. Collection elements mapped to the same processor can share data without communication, while if the collection is spread out over a large number of processors almost all references from one element to another involves network traffic. Speedup behavior similar to the Paragon was observed on the CM-5 for equivalent numbers of processors. For the BM3 benchmark, a 32 node Paragon achieved a fraction of 0.71 of the Cray uniprocessor Fortran version; speedup was 19.6. However, the most significant results are for the BM4 benchmark. Here, the time increased as processors were added. This is because of the intense communication required in the sparse matrix vector multiply. We cannot expect improvements in these numbers until Intel finishes their "performance release" of the system.

4.2 Shared Memory Systems

The shared memory ports of the pC++ uncover different performance issues from the distributed memory ports regarding the language and runtime system implementation performance. Here, the ability to achieve good memory locality is the key to good performance. Clearly, the choice of collection distribution is important, but the memory allocation schemes in the runtime system will play a big part. To better isolate the performance of runtime system components and to determine the relative influence of different phases of the entire benchmark execution where the runtime system was involved, we used a prototype tracing facility for pC++ for shared memory performance measurement. In addition to producing the same performance results reported above for the distributed memory systems, a more detailed execution time and speedup profile was obtained from the trace measurements. Although space limitations prevent detailed discussion of these results, they will be forthcoming in a technical report.

In general, we were pleased with the speedup results on the Sequent Symmetry, given that it is a bus-based multiprocessor. For all benchmarks, speedup results for 16 processors were good: BM1 (14.84), BM2 (14.15), BM3 (15.94), and BM4 (12.33). Beyond 16 processors, contention on the bus and in the memory system stalls speedup improvement. Although the Sequent implementation serves as an excellent pC++ testbed, the machine architecture and processor speed limits large scalability studies. The Symmetry pC++ runtime system implementation is, however, representative of ports to shared memory parallel machines with equivalent numbers of processors; e.g., the shared memory Cray YM/P or C90 machines. Using the four processor Sequent speedup results (3.7 to 3.99) as an indication, one might expect similar speedup performance on these systems. (Note, we are currently porting pC++ to a Cray YM/P and C90.)

The performance results for the BBN TC2000 reflect interesting architectural properties of the machine. Like the Sequent, benchmark speedups for 16 processor were encouraging: BM1 (14.72), BM2 (14.99), BM3 (15.92), and BM4 (11.59). BM1 speedup falls off to 23.89 and 32.36 at 32 and 64 processors, respectively, but these results are for a small 8 by 8 grid of subgrids, reflecting the small problem size performance encountered in the CM-5. BM2 speedup continues at a fairly even clip, indicating a better amortization of remote collection access costs that resulted in high communication overhead in the distributed memory versions. BM3 speedup was almost linear, achieving 31.48 for 32 processors and 58.14 for 64 processors. Unlike the Sequent, the BM4 speedup beyond 16 processors did not show any significant architectural limitations on performance.

The pC++ port to the KSR-1 was done most recently and should still be regarded as a prototype. Nevertheless, the performance results demonstrate the important architectural parameters of the machine. Up to 32 processors (1 cluster), speedup numbers steadily increase. BM1 to BM3 speedup results are very close to the TC2000 numbers; BM3 speedup for 64 processors was slightly less (52.71). However, BM4's speedup at 32 processors (9.13) is significantly less than the TC2000's result (17.29), highlighting the performance interactions of the choice of collection distribution and the hierarchical, cache-based KSR-1 memory system. Beyond 32 processors, two or more processor clusters are involved in the benchmark computations; we performed experiments up to 64 processors (2 clusters). As a result, a portion of the remote collection references must cross cluster rings; these references encounter latencies 3.5 times as slow as references made within a cluster. All benchmark speedup results reflect this overhead, falling to less than their 32 processor values.

5 Conclusion

Our experience implementing a runtime system for pC++ on five different parallel machines indicates that it is possible to achieve language portability and performance scalability goals simultaneously using a well-defined language/runtime system interface. The key, we believe, is to keep the number of runtime system requirements small and to concentrate on efficient implementations of required runtime system functions.

The three main pC++ runtime system tasks are collection class allocation, collection element access, and barrier synchronization. The implementation approach for these tasks is different for distributed memory than for shared memory architecture.

In the case of the distributed memory machines, the critical factor for performance is the availability of low latency, high bandwidth communication primitives. (Note that we have not made use of the CM-5 vector units or of highly optimized i860 code in the benchmarks.) While we expect the performance of these communication layers to improve dramatically over the next few months, we also expect to make changes in our compiler and runtime system. One important optimization will be to use barriers as infrequently as possible. In addition, it will be important to overlap more communication with computation.

In the case of shared memory machines, the performance focus shifts to the memory system. Although the BBN TC2000 architecture was classified as a shared memory architecture for this study, the non-uniform times for accessing collection elements in this machine result in runtime system performance characteristics similar to the distributed memory system. The more classic shared memory architecture of the Sequent Symmetry will require a closer study of memory locality trade-offs. Clearly, the choice of where to allocate collections in the shared memory can have important performance implications. In a hierarchical shared memory system, such as the KSR-1, the goal should be to allocate collection elements in a way that maximizes the chance of using the faster memory closer to the processors and that minimizes the possible contention and overhead in accessing remote memory. The problem for the runtime system becomes what memory allocation attributes to chose. The default choice is not guaranteed to always be optimal. Future versions of shared memory runtime systems may use properties of the collection classes to determine the appropriate element layout.

References

[1] BBN Advanced Computer Inc., Cambridge, MA. *Inside the TC2000*, 1989.

[2] D. Hensgen, R. Finkel, and U. Manber. Two Algorithms for Barrier Synchronization. *Int'l. Journal of Parallel Programming*, 17(1):1–17, 1988.

[3] Sequent Computer Systems, Inc. *Symmetry Multiprocessor Architecture Overview*, 1992.

[4] S. Frank, H. Burkhardt III, J. Rothnie, *The KSR1: Bridging the Gap Between Shared Memory and MPPs*, Proc. Compcon'93, San Francisco, 1993, pp. 285–294.

[5] T. von Eiken, D. Culler, S. Goldstein, K. Schauser, *Active Messages: a Mechanism for Integrated Communication and Computation*, Proc. 19th Int'l Symp. on Computer Architecture, Australia, May 1992.

[6] D. Culler, T. von Eiken, *CMAM - Introduction to CM-5 Active Message communication layer*, man page, CMAM distribution.

[7] A. Chien and W. Dally. *Concurrent Aggregates (CA)*, Proceedings of the Second ACM Sigplan Symposium on Principles & Practice of Parallel Programming, Seattle, Washington, March, 1990.

[8] High Performance Fortran Forum, *Draft High Performance Fortran Language Specification*, Version 1.0, 1993. Available from titan.cs.rice.edu by ftp.

[9] J. K. Lee, *Object Oriented Parallel Programming Paradigms and Environments For Supercomputers*, Ph.D. Thesis, Indiana University, Bloomington, Indiana, June 1992.

[10] J. K. Lee and D. Gannon, *Object Oriented Parallel Programming: Experiments and Results* Proc. Supercomputing 91, IEEE Computer Society and ACM SIGARCH, 1991, pp. 273–282.

[11] D. Gannon and J. K. Lee, *Object Oriented Parallelism: pC++ Ideas and Experiments* Proc. 1991 Japan Society for Parallel Processing, pp. 13-23.

[12] D. Gannon, J. K. Lee, On Using Object Oriented Parallel Programming to Build Distributed Algebraic Abstractions, *Proc. CONPAR/VAPP*, Lyon, Sept. 1992.

[13] D. Gannon, *Libraries and Tools for Object Parallel Programming*, Proc. CNRS-NSF Workshop on Environments and Tools For Parallel Scientific Computing, 1992, St. Hilaire du Touvet, France. Elsevier, Advances in Parallel Computing, Vol.6.

[14] K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*, Ph.D. Thesis, Yale University, 1986.

[15] Z. Lahjomri and T. Priol, KOAN: a Shared Virtual Memory for the iPSC/2 Hypercube, *Proc. CONPAR/VAPP*, Lyon, Sept. 1992.

[16] M. Lemke and D. Quinlan, a Parallel C++ Array Class Library for Architecture-Independent Development of Numerical Software, *Proc. OON-SKI Object Oriented Numerics Conf.*, pp. 268–269, Sun River, Oregon, April 1993.

[17] J. Dongarra, R. Pozo, D. Walker, An Object Oriented Design for High Performance Linear Algebra on Distributed Memory Architectures, *Proc. OON-SKI Object Oriented Numerics Conf.*, pp. 257–264, Sun River, Oregon, April 1993.