On Using Object Oriented Parallel Programming to Build Distributed Algebraic Abstractions

Dennis Gannon, Jenq Kuen Lee, Srinivas Narayana

CICA and Department of Computer Science
Indiana University

Abstract. This paper considers the problem of designing numerical linear algebra computations with a parallel object oriented programming language. An extension to C++ based on a *Concurrent Aggregate* concept is used to describe a hierarchy of distributed structures including dense matrices, sparse matrices, banded matrices, vectors and the associated linear algebra. We show that these abstractions are ideal for describing BLAS3 style blocked computation. As an example we show how to build a distributed conjugate gradient algorithms for sparse matrix (based on the NAS sparse benchmark and we show results for four different parallel machines.

1 Introduction

There are two major problems with parallel computation as it applies to scientific programming. First, automatic parallelization of old Fortran programs is not showing the types of speed-ups necessary to scale to hundreds or thousands of processors. Second, the alternative to compiling old programs is either to rewrite them in a machine specific, low level language or to wait for solutions like the HPC Fortran90-D [5] effort to become available. While the later alternative is not bad, one is still left with a Fortran program with a structure that is a long way from the mathematical models that go into the design of the computations.

In this paper we propose a methodology for using object-oriented style to build portable parallel programs that allow users to express computation completely in terms of the basic mathematical operators that describe the underlying algorithms.

Our approach is to extend the C++ language with a new construct called the distributed collection. This new language is called pC++ [3, 4] and it currently runs on several different parallel processing platforms. To illustrate the ideas behind pC++ in the context of scientific computation we will outline a design for a *distributed linear algebra collection-class hierarchy.* By “distributed”, we mean structures that have been partitioned among the available processors in such a way that local memory and cache have been used to optimal advantage. In the last section of this paper we describe a large benchmark that we have implemented with this ”object parallel” library and show some early results.

2 Parallel C++ (pC++) and the Collection Construct

The *Collection* construct is similar to class construct in the C++ language[6]. The difference is that a collection is a structured set of objects from some standard C++ class type that has been distributed over a set of processors. The idea is very simple. Let C be a class of the form

```
Class C{
     float data;
     ....
  public:
     C operator +(C x);
     float foo();
  }
```

C has an overloaded binary operator + and two functions. The collection class *DistributedArray* is part of the standard pC++ library. To define a 2-dimensional m by n array X of objects of type C which are distributed over a "vector" of processors of size P, one writes

```
     DistributedArray<C> X([P], [m, n], [Block, Whole]);
```

The key words *Block* and *Whole* describe how each dimension of the array is mapped to the 1-D array of processors. (This is identical to the Fortran-D, HP-Fortran model.)

Parallelism now comes from two possible sources. First, the operation

```
   X.foo();
```

means the parallel invocation of the C class function foo on each element of the collection. Because foo returns a float, the value returned by this call is of type $DistributedArray < float >$ with the same shape and distribution as X. Similarly, the operator $X + X$ is a parallel, pointwise application of the + operator to every member of the collection.

The second source of parallelism is the invocation of parallel operators associated with the global Collection structure. For distributed arrays this includes global data movement and reduction operators.

3 Inheritance and A Distributed Matrix Class Hierarchy

The most powerful feature of C++, or any other object oriented language, is the ability to define a class hierarchy where one class is defined in terms of properties that are inherited from another. Our implementation for a parallel linear algebra system begins with such a C++ class hierarchy for standard sequential linear algebra. In what follows we assume that there is a class *vector* which can be defined over the reals or complex numbers (single or double precision). In addition, we will make use of a class *matrix* with subclasses *dense_matrix*, *sparse_matrix* and *banded_matrix* and all the associated operators. (For the complete details, the full version of this paper is available from gannon@iuvax.cs.indiana.edu.)

In the same manner as C++ classes, pC++ collections form an inheritance hierarchy. To build a distributed vector collection we derive it from the DistributedArray class as

```
Collection Dvector: DistributedArray{
   public:
      Dvector operator +(Dvector x);
      ....
   };
```

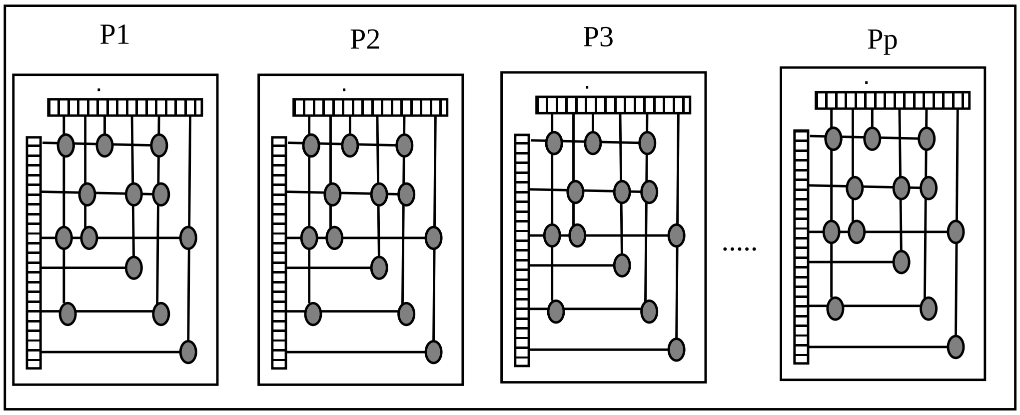
The collection *Dvector* has all the basic operators associated with a *vector* in the standard linear algebra. Consequently we can define a distributed vector V as

```
Dvector<float> V(...);
```

In our case we create a large-grain distributed vector by the declaration

```
Dvector<vector> V(...);
```

We use a very simple method to build a distributed matrix. We view the processor set as a 1-D array and partition the c columns into c/P blocks where P is the total number of available processors in exactly the same way as we have described for distributed vectors. as shown in Figure 1. The collection that implements this is called a Dcol_matrix.



Dcol_matrix< sparse_matrix > M(..)

Fig. 1. A Distributed Column matrix of Sparse Matrices.

By using the algebraic abstractions provided by this hierarchy a programmer can easily specify the type and structure for distributed vectors and matrices. For example, to define a distributed sparse matrix A, and a pair of distributed vectors, x and y and to do a matrix vector multiply, one writes

```
Dcol_matrix<sparse_matrix> A(n,n);
Dvector<vector> x(n), y(n);

y = A*x;
```

4 A Benchmark Experiment

To illustrate the utility of these concepts we have rewritten the NASA Ames NAS Sparse Conjugate Gradient Benchmark to use these distributed matrix and vector abstractions. This benchmark uses the CG algorithm to compute an approximation of the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. According to the NASA team that designed the benchmark, this computation is typical of unstructured grid computations in that it tests irregular long distance communication in parallel machines by employing unstructured matrix vector multiplication.

The benchmark code builds a large randomly sparse matrix and then makes frequent calls to a CG solver, which when translated into pC++ is shown below.

Our implementation uses one distributed sparse matrix and six distributed vectors. There is a great deal more to this benchmark, but the following routine is the key component.

```
void cgsol(int n, Dcol_matrix<sparse_matrix>& S, Dvector<vector>& b,
       Dvector<vector> & x, int nitcg, Dvector<vector> & r,
       Dvector<vector> & p, Dvector<vector> & q) {
       int  i, iter;
       double   alpha, beta, rho, rho0, dinv;
       x = 0;  r = b; p = r; rho = r*r;
       for(iter = 1; iter <= nitcg; iter++){
           q = S*p;
           alpha = rho /(p*q);
           x += alpha*p;
           rho0 = rho;
           r -= alpha*q;
           rho = r*r;
           beta = rho / rho0;
           p = (beta)*p + r;
           }
      r = S*x;
      r = b - r;
      resid =  r.norm();
}
```

While we would like to say that our compiler can handle this routine as shown, there are still a few basic transformations that we have not completed. The most important of these involves the use of expressions of the form

```
p = (beta)*p + r;
```

A standard C++ compiler would allocate two temporary arrays: one to compute the product (beta)*p and another to return the value of the sum. In pC++ we must be more careful, because such an allocation would generate a distributed allocation and a lot of unnecessary copying. We plan to have these optimization implemented in the next version of the compiler.

We have run the pC++ version of the program on four different machines. The BBN GP1000, the BBN TC2000, the Alliant FX/2800 and the Thinking Machines CM-5. With the exception of the current CM-5 implementation (discussed below), the program is completely portable, so no no changes in the source code was needed in moving from one machine to another. The problem size can be set by a parameter. We selected the largest problem that would run on all four machines. The matrix size 2400 by 2400 with a total of 135238 random zeros. Execution time and speed-up numbers are given in the table below. The times are in seconds and the horizontal axis is the number of processors.

Machine	P= 5	P= 10	P= 12	P= 20	P= 25	P= 40
GP1000 time	439	244	215	146	135	128
speed-up	4.98	8.95	10.16	14.95	16.19	17.07
FX/2800 time	76.5	43.6	40.8			
speed-up	4.6	8.01	8.6			
TC2000 time	91	54	49	36.5	35	
speed-up	4.8	8.1	8.9	12	12.5	

As is clear from the experiments, the program is far from showing unbounded speed-up. One reason for this is that the problem size is still relatively small. Consequently, the large number of inner products in the CG algorithm will cause it to have a sublinear speed-up if there is not enough computation to mask the $log(p)$ cost of the reductions. But we can still put these results in perspective In the case of the Alliant FX/2800, we have access to an automatic parallelizing and vectorizing compiler for Fortran. In the table below, we have included the results of running the optimized Fortran program on the benchmark test data size of 1400 by 1400 with 78148 non-zeros. The Fortran program was compiled with full vector concurrent optimization.

Machine	P= 2	P= 5	P= 10	P= 14
FX/2800 pC++ time	112.8	47.9	27.0	22.33
FX-Fortran time	78.38	73.45	73.63	72.98
ratio Fort/pC++	0.69	1.5	2.7	3.2

As can be seen, the Alliant Fortran compiler is not able to extract any significant parallelism from this computation. On the other hand,the pC++ program, which has no vectorization within code that runs on each I860, runs slower for small numbers of processors but over three times faster than the Fortran code on 14 processors. We must also add one more caveat. While the pC++ performance may look good, the absolute performance was less than 10the original benchmark when run on a Cray YMP.

More recently we have completed a partial port of our compiler to the Thinking Machines CM-5 and we have tested this computation on a machine with 128 processors. In this case we have executed the benchmark at its full size of 14000 by 14000 with 1853104 non-zeros. The speed of the computation is 18.9 Mflops which is equal to 0.27 times the speed of a 1 processor Cray Y/MP on the Fortran version of this benchmark. We emphasize that the machine is an early version without the floating point vector hardware and a very early version of the communications library. By a "partial port" of the compiler we mean that the main vector and matrix routines had to be modified by hand to use message passing primitives provided by TMC. On the other hand, the CGM application code did not need any modification. (On the other machines discussed here none of the basic library code for vector and matrix operations had to be modified in going from one machine to another.) We also note that 80% of the time consumed by CGM on the CM-5 is spent in the data communication routines in the Matrix times vector computation. We hope to report better performance numbers as the Thinking Machines Software communications software improves.

References

1. Andrew A. Chien and William J. Dally. *Concurrent Aggregates (CA)*, Proceedings of the Second ACM Sigplan Symposium on Principles & Practice of Parallel Programming, Seattle, Washington, March, 1990.
2. Charles Koelbel, Piyush Mehrotra, John Van Rosendale. *Supporting Shared Data Structures on Distributed Memory Architectures*, Technical Report No. 90-7, Institute for Computer Applications in Science and Engineering, January 1990.
3. J. K. Lee and D. Gannon, *Object Oriented Parallel Programming: PC++*, Proceedings of Supercomputing 91 (Albuquerque, Nov.), IEEE Computer Sciety and ACM SIGARCH, 1991, pp. 273–282.
4. D. Gannon and J. K. Lee, *Object Oriented Parallelism: pC++ Ideas and Experiments*, Proceedings of 1991 Japan Society for Parallel Processing, pp. 13–23, 1991. Also Bit V.24, no. 1 (1992) (in Japanese).
5. G. Fox and S. Hiranandani and K. Kennedy and C. Koelbel and U. Kremer and C. Tseng and M. Wu, *Fortran D language specification*, Rice University, Department of Computer Science TR 90079, 1991
6. Bjarne Stroustrup. *The C++ programming Language*, Addison Wesley, Reading, MA, 1986

This article was processed using the LaTeX macro package with LLNCS style