A DAta-Parallel Programming Library for Education (DAPPLE)

David Kotz

Technical Report PCS-TR94-235
Department of Computer Science
Dartmouth College
Hanover, NH 03755-3510
dfk@cs.dartmouth.edu

November 7, 1994

Abstract

In the context of our overall goal to bring the concepts of parallel computing into the undergraduate curriculum, we set out to find a parallel-programming language for student use. To make it accessible to students at all levels, and to be independent of any particular hardware platform, we chose to design our own language, based on a data-parallel model and on C++. The result, DAPPLE, is a C++ class library designed to provide the illusion of a data-parallel programming language on conventional hardware and with conventional compilers. DAPPLE defines *Vectors* and *Matrices* as basic classes, with all the usual C++ operators overloaded to provide elementwise arithmetic. In addition, DAPPLE provides typical data-parallel operations like scans, permutations, and reductions. Finally, DAPPLE provides a parallel if-then-else statement to restrict the scope of the above operations to partial vectors or matrices.

1 Introduction

Parallel computing, having been considered an advanced topic suitable only for graduate students, is slowly migrating into the undergraduate curriculum [Mil94]. We believe parallelism should be introduced early in the curriculum, before the habits of sequential thinking are ingrained. Indeed, we are preparing to teach it to freshmen in CS2 [JKM94]. We use a *data-parallel* programming model, whose single thread of control allows students to explore issues in parallel algorithms without the complexities of asynchrony, deadlock, and communication. (While these are important issues in parallel computing, we feel that it is best to allow the students to focus on the underlying parallelism first, and to postpone these other issues to a later course.)

We wanted a programming language that allowed students to experiment with parallel computing *concepts* without being distracted by the *mechanics* of parallel programming. In addition,

This research was supported under grant DUE-9352796 by the National Science Foundation ILI-LLD program.
A revised version of this report will appear in SIGCSE '95.

Copyright 1994 by the authors

we wanted a parallel programming language that was essentially the same as the language used by students for their sequential programming (preferably C++), was available on the computers they use, was easy to learn by beginners, and was usable by students at all levels in many kinds of courses. Although many data-parallel languages exist, including C*, Fortran90, NESL [Ble93], and HPF [Lov93], they are difficult to use, are not similar to C++, or are not easily portable to student computers.

We found many research projects designing parallel C++ variants. C** [LRV92] is perhaps the closest candidate, in that it supports a data-parallel model, but it requires a new compiler and is not yet available. pC++ [BBG+93] can also provide a data-parallel model, using only a preprocessor and library, but its syntax is a little complicated for beginners. Other data-parallel options like Presto++ [Kil92] and Compositional C++ [CK92] are also rather complex for beginners. Others, like Mentat [Gri93], CHARM++ [KK93], and COOL [CGH94], are more task-parallel than data-parallel.

Finding no suitable existing language, we decided to design and implement our own language as a set of macros and classes that extended C++. The result is DAPPLE, a DAta-Parallel Programming Library for Education. DAPPLE gains its strength from its simplicity, portability, and versatility, rather than from performance or ease of implementation on real parallel hardware. In other words, DAPPLE was optimized for pedagogical use.

In this paper, after a quick review of the data-parallel programming model, we give an overview of DAPPLE through three programming examples.

2 Data-parallel programming

The data-parallel programming model gives the programmer a single thread of control, much as in sequential programming languages, but allows certain operations to be applied to large collections of data simultaneously. For example, the sum of two arrays may be assigned to a third array by using many *virtual processors* in parallel, each responsible for computing one (scalar) sum and storing it in the appropriate element of the result array.

When the condition expression of an `if()` statement refers to collections, the expression is independently evaluated by every virtual processor. Those virtual processors where the condition is true execute the "then" clause (simultaneously), and those where the condition is false execute the "else" clause (simultaneously). Within each clause, only a subset of the processors are *active*, and only active processors participate in operations on collections. In other words, a parallel `if()`

reduces the *context* of collection operations within each clause. Finally, there are other operations on entire collections, such as reducing a collection to a scalar by summing all the elements, or printing the collection.

3 DAPPLE programming

DAPPLE adds data-parallel concepts to C++ programming, allowing the programmer to manipulate collections of data (vectors and matrices) as described above. To illustrate these concepts and the language, we present three examples.

3.1 Pascal's triangle

Pascal's triangle is a set of rows, where the first row contains one "1" followed by an infinite number of "0"s. Each entry in the next row is the sum of the entry above it and the entry above and to the left. Inductively, row i has i non-zero entries. The result (one row per line, not showing the zeros) is

```
1
1       1
1       2       1
1       3       3       1
1       4       6       4       1
1       5       10      10      5       1
```

and so forth. Here is part of a DAPPLE program to compute Pascal's triangle:

```
const int N = 6;   // we will compute N rows of Pascal's triangle
intVector arow(N); // N elements, uninitialized
```

The second statement defines an integer vector called `arow`, with N elements numbered $0, 1, \ldots, N-1$. (DAPPLE supports new classes `intVector`, `charVector, floatVector, doubleVector,` and `booleanVector`).[1] This vector will soon contain one row of the triangle, but for now the elements are uninitialized. Vectors may also be initialized when defined, to a scalar, an array, another vector, or a function of the index. For example,

```
extern int Identity(int i);    // defined by DAPPLE; returns i
const intVector VP(N, Identity);
```

[1] We chose not to use templates because current compilers vary in their ability to support templates, and because templates were not sufficiently expressive.

defines an N-element integer vector called `VP`, initialized so that element i has value i. We next use a parallel-if statement, `ifp()`, to initialize `arow` (for comparison, we present the equivalent sequential code):

```
// DAPPLE                    //    Sequential equivalent
ifp (VP == 0) {              //
    arow = 1;                //    arow[0] = 1;
    cout << arow << endl;    //    cout << arow[0] << endl;
} else                       //    for (int i = 1; i < N; i++)
  arow = 0;                  //      arow[i] = 0;
```

The "then" clause executes only for those virtual processors where the condition (VP == 0) is true, in this case, only virtual processor 0. Thus, it assigns and prints only `arow[0]`. This one element is of course the entire first row of Pascal's triangle. The "else" clause executes for the remaining virtual processors. To compute and print $N-1$ more rows, we loop:

```
// DAPPLE                          //  Sequential equivalent
for (int i = 1; i < N; i++) {      //  for (int i = 1; i < N; i++) {
                                   //      for (int j = i; j > 0; j--)
    arow += shift(arow, 1);        //        arow[j] += arow[j-1];
    ifp (VP <= i)                  //      for (int j = 0; j < i; j++)
      cout << arow << endl;        //        cout << arow[j] << '\t';
                                   //      cout << arow[i] << endl;
}                                  //  }
```

Each time through the loop we compute a new row of the triangle, in parallel, by adding the current row to itself, shifted one to the right (a zero is shifted in at the left side).[2] Then, we print out the vector, but only elements 0 through i, i.e., the non-zero elements of this row.

3.2 Matrix-matrix multiply

In addition to vectors, DAPPLE supports a set of Matrix classes.[3] Figure 1 shows most of a program to multiply two integer matrices.[4] Three matrices are defined as type `intMatrix(`r`,`c`)`, where integers r and c specify the number of rows and columns. Note that `A` and `B` are initialized from user input using the standard `iostream` operator `>>`, overloaded by DAPPLE for matrix (or vector) input.

A nested loop computes each element of the result matrix C as an inner product (dot product) of the appropriate row of A and the appropriate column of B, demonstrating DAPPLE's capability

[2] Purists of object-oriented programming note that we chose a functional rather than object-oriented style for most operations. The functional style makes it easier to compose operations, e.g., `B = shift(B,1) + B + shift(B,-1)`, than if `shift()` modified `B`. Recommended by the ARM [ES90, page 249], the functional syntax `shift(B,1)` makes it clear that the operand `B` is not modified, while in `B.shift(1)` it is not as clear. Similarly, we believe that `x =`

```
// we'll multiply a PxQ matrix by a QxR matrix to get a PxR matrix
int P, Q, R;
cin >> P >> Q >> R;

// we'll compute C = A * B
intMatrix A(P,Q), B(Q,R), C(P,R);

// load matrices; row-major order, whitespace-separated integers
cin >> A;
cin >> B;

// loop through the result locations
for (int r = 0; r < P; r++)
  for (int c = 0; c < R; c++)
    C[r][c] = inner(A[r][_], B[_][c]);

cout << C;

intMatrix D(P,R);           // D is what C should be
cin >> D;

if (any(C != D))
  cout << "The answers are different!" << endl;
else
  cout << "The answers are the same." << endl;
```

Figure 1: A matrix-matrix multiplication program in DAPPLE.

to work with *matrix slices* [LRV92]. Here, `A[r][_]` is a *row slice*, representing row r of matrix A, and `B[_][c]` is a *column slice*, representing column c of matrix B. Slices may be used anywhere vectors may be used, including on the left-hand side of an assignment operator.

The function `inner(v1, v2)` is provided by DAPPLE, but the same operation could also be expressed as `sum(v1 * v2)`, using DAPPLE's built-in reduction function called `sum()`.

The final `if()` statement demonstrates a handy reduction, `any()`, which returns (scalar) true if and only if some element of its vector or matrix argument is non-zero. Here, its argument is the boolean matrix representing the condition `(C != D)`, so `any(C != D)` is true if there is any position (i, j) where $C_{ij} \neq D_{ij}$.

3.3 Quicksort

To demonstrate DAPPLE's ability to manipulate data within a vector, and in particular its ability to dynamically narrow context to a subset of the virtual processors, we devised a simple recursive implementation of quicksort (Figure 2).[5] The `quicksort` procedure recursively sorts the active portion of its vector argument. (Initially, quicksort is called with all processors active.) It begins by using the reduction `n_active()` to find the size of the subvector it is to sort. Then, it dispenses with two special cases: subvectors of size 0 or 1 are trivially sorted, and a subvector of size 2 may require a swap. (We use reductions `min_value()`, `max_value()`, and `first()`, to compute the minimum and maximum values and assign them to the appropriate element.) Otherwise, we partition and recurse. To partition, it chooses a splitter value (here, the value at the first active processor), builds a permutation subvector that specifies the destination of every element in the repartitioned subvector, and then permutes. It restricts the context to the left partition and recurses, and then restricts the context to the right partition and recurses.

The quicksort example demonstrates one weakness of DAPPLE, its inability to support nested data parallelism [Ble93]. The two recursive calls to quicksort() must be done sequentially, each with only a small subset of the virtual processors active. Given this model, other sorting algorithms would be more appropriate. Exploring this issue would be a valuable lesson for students.

`sum(A*B+C)` is clearer than `x = (A*B+C).sum()`.

[3] For consistency, we decided that all overloaded operators would be elementwise operators, so `C=A*B` for three matrices A, B, and C does an elementwise multiplication and not a matrix multiplication.

[4] Of course, there are better algorithms, but this serves to demonstrate DAPPLE. Also, there are more efficient ways to program quicksort in DAPPLE (not shown).

[5] In a classroom setting, of course, we ensure the students are familiar with sequential quicksort before exposing them to parallel quicksort.

```
void quicksort(intVector& X) // the sort is done in place, ie, X is updated
{
    // check the number of active processors (ie, size of our sublist)
    int n = n_active(X);        // how big is this sublist?
    if (n <= 1) ;               // do nothing
    else if (n == 2) {          // possibly swap them
        int largest = max_value(X);
        int smallest = min_value(X);

        ifp (VP == first(VP))
          X = smallest;           //  first one get smallest
        else
          X = largest;            // second one gets largest
    } else {                      // n >= 3
        intVector P(N);           // permutation vector
        const intVector ONE(N,1); // constant vector of all 1s
        int splitter;             // splitter value
        int left, middle, right;  // first VP# in each subset

        // pick a splitter; I'll just use the first value
        splitter = first(X);
        left = first_index(X);  // which VP holds the splitter?

        // find the left half, those less than or equal to splitter
        // (except for that first one)...
        ifp (X <= splitter && VP != left) {
            // compute our destination in the result vector
            P = left + plus_scan(ONE); // i.e., left, left+1, left+2...
            middle = left + n_active(X); // the rest will begin here
        }

        // move the splitter into the middle
        ifp (VP == left) {
            P = middle;         // route it there later
            right = middle + 1; // the rest will begin here
        }

        // do the right half, those greater than the splitter
        ifp (X > splitter) {
            // compute our destination in the result vector
            P = right + plus_scan(ONE); // i.e., right, right+1, right+2...
        }

        X = permute(X, P);      // partition the data
        ifp (VP < middle)
          quicksort(X);         // sort the left half
        ifp (VP > middle)
          quicksort(X);         // sort the right half
    }
}
```

Figure 2: A quicksort function in DAPPLE.

4 Summary and status

The DAPPLE extensions to C++ are summarized in Table 1. We are fine-tuning the language and implementation for use in a parallel-computing course later this year [JKM94]. DAPPLE should be useful beyond that course, however, in other courses and in other institutions.

DAPPLE currently runs on DECstation 5000 workstations with Ultrix and the g++ compiler, and we are porting it to other Unix workstations (Sun, SGI, and DEC Alpha) and to the Macintosh (using Symantec C++). DAPPLE is not yet publically available, but the complete package (code, documentation, tutorial, and examples) will be available by ftp and WWW before SIGCSE. Watch the URL `http://www.cs.dartmouth.edu/ILI/dapple/`.

Acknowledgements

Many thanks to all of those who made suggestions about the language or this paper, or helped with subtle points of C++ technique, including Owen Astrachan, Tom Cormen, Fillia Makedon, Takis Metaxas, Nils Nieuwejaar, Sam Rebelsky, Scott Silver, and Cliff Stein.

References

[BBG+93] François Bodin, Peter Beckman, Denis Gannon, Srinivas Narayana, and Shelby X. Yang. Distributed pC++: basic ideas for an object parallel language. *Scientific Programming*, 2(3), Fall 1993.

[Ble93] Guy E. Blelloch. NESL: a nested data-parallel language. Technical Report CMU-CS-93-129, Carnegie Mellon University, April 1993.

[CGH94] Rohit Chandra, Anoop Gupta, and John L. Hennessey. COOL: an object-based language for parallel programming. *IEEE Computer*, 27(8):14–26, August 1994.

[CK92] K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. Technical Report CS-TR-92-13, California Institute of Technology, 1992.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990. Ninth printing.

[Gri93] Andrew S. Grimshaw. Easy-to-use object-oriented parallel processing with Mentat. *IEEE Computer*, 26(5):39–51, May 1993.

[JKM94] Donald Johnson, David Kotz, and Fillia Makedon. Teaching parallel computing to freshmen. In *Conference on Parallel Computing for Undergraduates.* Colgate University, June 1994.

[Kil92] Michael F. Kilian. *Parallel Sets: An Object-oriented Methodology for Massively Parallel Programming.* PhD thesis, Harvard University, 1992.

[KK93] L.V. Kale and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, 1993.

[Lov93] David B. Loveman. High Performance Fortran. *IEEE Parallel and Distributed Technology*, 1(1):25–42, February 1993.

Table 1: Summary of DAPPLE extensions to C++.

	Vectors	**Matrices**
Types	int, char, float, double, boolean	same
Initializations	(none), scalar, array, function	same
	another vector	another matrix
Subscripting	`V[i]`	`M[i][j], M[i][_], M[_][j]`
Vector products	scalar = `inner(VA,VB)`	
	matrix = `outer(VA,VB)`	
Elementwise:		
Arithmetic operators	`+ - * / %`	same
Relational operators	`< <= == != >= >`	same
Boolean operators	`&& || !`	same
Assignment operators	`= += -= *= /= %= ++ --`	same
Function application	`apply`(*function, vector*)	`apply`(*function, matrix*)
Reductions:		
sum	x = sum(V);	same
are any nonzero?	b = any(V);	same
are all nonzero?	b = all(V);	same
number of nonzeros	n = n_nonzeros(V);	same
number active	n = n_active(V);	same
value of first active	x = first(V);	N/A
index of first active	n = first_index(V);	N/A
maximum value	x = max_value(V);	same
minimum value	x = min_value(V);	same
index of max	n = max_index(V);	N/A
index of min	n = min_index(V);	N/A
Scans:	VA = plus_scan(VB);	plus_scan_rows, plus_scan_cols
	VA = max_scan(VB);	max_scan_rows, max_scan_cols
	VA = min_scan(VB);	min_scan_rows, min_scan_cols
	VA = or_scan(VB);	or_scan_rows, or_scan_cols
	VA = and_scan(VB);	and_scan_rows, and_scan_cols
Moving data:	VA = shift(VB, distance);	MA = shift(MB, rows, cols);
		MA = shift_rows(MB, *distance per row*);
		MA = shift_cols(MB, *distance per column*);
	VA = rotate(VB, distance);	MA = rotate(MB, rows, cols);
		MA = rotate_rows(MB, *distance per row*);
		MA = rotate_cols(MB, *distance per column*);
	VA = pack(VB);	N/A
	VA = permute(VB, P);	N/A
	VA = permute(VB, *function*);	N/A
Input and output:		
input	`cin >> V;`	same
output	`cout << V;`	same
	`cerr << V;`	same
Parallel if statement:		
`ifp() ... else ...`		

[LRV92] James R. Larus, Brad Richards, and Guhan Viswanathan. C**: A large-grain, object-oriented, data-parallel programming language. Technical Report #1126, University of Wisconsin-Madison, November 1992.

[Mil94] Russ Miller. The status of parallel processing education. *IEEE Computer*, pages 40–43, August 1994.