A DAta-Parallel Programming Library for Education (DAPPLE)

David Kotz
Department of Computer Science
Dartmouth College
Hanover, NH 03755-3510
dfk@cs.dartmouth.edu

Running head: 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. Thanks to Naval Ravikant for developing the wonderful animations, and to Prasad Jayanti and Cliff Stein for letting me experiment with their classes.

This research was supported under grant DUE-9352796 by the National Science Foundation ILI-LLD program.

Abstract

In the context of our goal to bring parallel computing into the undergraduate curriculum, we needed a parallel-programming language that was accessible to students and independent of any particular hardware platform. Finding nothing appropriate, we chose to design our own language. 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 *Vector* and *Matrix* classes, with most C++ operators overloaded to provide elementwise arithmetic, and supports data-parallel operations like scans, permutations, and reductions. DAPPLE also provides a parallel if-then-else statement to restrict the scope of the above operations to partial vectors or matrices. In this paper we describe the DAPPLE language, the pedagogical decisions that went into its design, and our experience using DAPPLE in the classroom. DAPPLE is freely available on the Internet.

1 Introduction

Parallel computing, having been considered an advanced topic suitable only for graduate students, is slowly migrating into the undergraduate curriculum [1]. We believe parallelism should be introduced early in the curriculum, before the habits of sequential thinking are ingrained. Indeed, some have had success teaching the elementary concepts to high-school students [2]. When limited resources constrained our original plan to replace our CS2 data-structures and programming course with a course centered on parallel computing [3], we focused on the addition of a week-long module about parallel computing to the existing CS2 course. We used the techniques described in this paper to teach parallel computing to first-year undergraduates in CS2.

When teaching parallel computing to first-year undergraduates, one must carefully consider the approach. We believe that it is important for the students to study parallel programs as well as parallel algorithms, and to have hands-on experience with parallel programming. We used a *data-parallel* programming model, whose single thread of control allowed students to explore issues in parallel algorithms without the complexities of asynchrony, deadlock, and communication. (While these are important issues in parallel computing, we felt that it was 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, 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 [4], and HPF [5], 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** [6] 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++ [7] 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++ [8] and Compositional C++ [9] are also rather complex for beginners. Others, like Mentat [10], CHARM++ [11], and COOL [12], are more task-parallel than data-parallel. Recent efforts [13, 14, 15] are only in early stages of development.

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.

After a quick review of the data-parallel programming model in Section 2, we give an overview of the DAPPLE language in Section 3, including a discussion of the pedagogical decisions involved in the language's design. Section 4 comments on some interesting implementation issues. Section 5 describes our experience using DAPPLE in the classroom. We conclude in Section 6 and point out how to obtain DAPPLE for your own classroom.

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 conditional 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). Because non-parallel code may also be part of these clauses, the semantics say that the "then" clause executes first, and then the "else" clause executes. 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 four examples and a summary of the language.

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. Figure 1 shows part of a DAPPLE program to compute Pascal's triangle. 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. The third and fourth statements of Figure 1 define an N-element integer vector called `VP`, initialized so that element i has value i.

Figure 1 uses a parallel-if statement, `ifp()`, to initialize `arow` (for comparison, it also presents the equivalent sequential code). 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.

The `for` loop of Figure 1 computes and prints $N-1$ more rows. 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.

---

[1] We chose not to use templates because current compilers vary in their ability to support templates.

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

3.2 Computing π

Another simple example shows students the power of data-parallel computing on a familiar problem: numerical integration. In Figure 2 we use the rectangle rule to estimate π as $\int_0^1 \frac{4}{1+x^2}$. There are N rectangles of width $\frac{1}{N}$, with rectangle i ($0 \leq i < N$) located at position $x_i = (i + \frac{1}{2})\frac{1}{N}$. After computing the x values in parallel, it is then easy to compute the function $y_i = \frac{4}{1+x_i^2}$ in parallel. Each rectangle's height y is multiplied by its width $\frac{1}{N}$ to get its area, and then the `sum()` reduction provides the total area, our estimate for π. This program prints "`pi ~= 3.14159`".

3.3 Matrix-matrix multiply

In addition to vectors, DAPPLE supports a set of Matrix classes. Figure 3 shows most of a program to multiply two integer matrices.[3] 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 to work with *matrix slices* [6]. 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 $C_{ij} \neq D_{ij}$. Although one might be tempted to write `ifp(C != D)` instead, that would have a different effect: the first message would be printed once for every virtual processor where $C_{ij} \neq D_{ij}$, and the second message would be printed once for every virtual processor where $C_{ij} = D_{ij}$.

[3] Of course, there are better algorithms, but this serves to demonstrate DAPPLE. 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.

3.4 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 4).[4] 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 [4]. 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 makes a valuable lesson for students.

3.5 DAPPLE design

The DAPPLE extensions to C++, most of which are exhibited in the above examples, are summarized in Table 1. In the course of designing the DAPPLE language, we encountered many decisions, small and large, that affect the syntax and semantics of the language. Some of the more interesting design decisions follow.

Early on, we decided to expose the notion of *virtual processors* to the students. In this way DAPPLE is more like C*, which defines operations on parallel variables with fixed "shapes", than like NESL, which defines abstract operations on variable-length lists. Although NESL provides a higher-level abstraction, we felt that it was important to give the students a small connection to the hypothetical parallel execution environment. DAPPLE's conceptual model makes a virtual processor responsible for executing elementwise operations at corresponding positions of two operands

[4] In a classroom setting, of course, we ensure the students are familiar with sequential quicksort before exposing them to parallel quicksort. Also, there are more efficient ways to program quicksort in DAPPLE (not shown).

with the same shape. This concept is reflected more strongly in the semantics of the `ifp` statement, which makes some virtual processors inactive during the execution of its "then" and "else" clauses. This construction makes it clear to students that there are times when some processors are inactive, and makes students aware of the inefficiencies that occur in some parallel programs.

Once we decided to base our conceptual model on virtual processors that were either active or inactive, we needed to define the semantics of all operations over active subsets of vectors and matrices. Because an active subset is defined by a `ifp` statement using a boolean collection (usually the result of a boolean expression), the active subset itself has a shape. Within the dynamic context of the `ifp` statement, only collections whose shape matches that of the active set may be manipulated. Although occasionally inconvenient, it is a direct result of semantics based on "active" and "inactive" virtual processors, and is a feature of all languages like C*.

As mentioned earlier, we chose a functional notation for our operators, rather than an object notation. Functions of collections always return new collections, a notion that is convenient for common expressions like the following:

```
intMatrix A(n);
...
// add each elements' neighbors to itself
A += shift(A, -1, -1) + shift(A, -1, 1) + shift(A, 1, -1) + shift(A, 1, 1);
```

This functional notation makes the functional behavior more obvious, and consistent with operators, than does the pure object form below:

```
intMatrix A(n);
...
// add each elements' neighbors to itself
A += A.shift(-1, -1) + A.shift(-1, 1) + A.shift(1, -1) + A.shift(1, 1);
```

We always tried to choose the semantics that would be intuitive to most students. In some cases, intuition clashes with consistency. While it would be consistent for all collection operations to be only affect elements corresponding to active virtual processors, we found a few operations that should be insensitive to the active set. For example, initialization, `shift()`, and `rotate()`:

```
floatVector A(n);
...
ifp (A > 0) {
    // average of three neighbors, but do it only where A > 0
    A = (shift(A, -1) + A + shift(A, 1)) / 3;
}
ifp (A > 0) {
    floatVector X(n, 0);  // a vector of n integers, initialized to 0
    ...
    A = rotate(X, -1);
}
```

In the first `ifp` statement, each `shift()` shifts the entire vector left or right, regardless of the active set. Of course, the addition and assignment only occur on active elements of the vector. The alternative would have `shift(A,i)` move each *active* element over by i active positions. We found little use for these semantics, and plenty of use for those above. In the second `ifp` we see the new vector `X` used as the argument to a `rotate`. Since inactive elements of `X` may be rotated into active positions, and assigned to `A`, it was important to initialize all elements of a collection. This decision had the undesired effect of making initialization different from assignment, but makes it harder for uninitialized variables to sneak into student's programs.

The permutation operator required a similar decision. We decided that the result of `permute(X,P)` would be initialized to `X`, and then the active elements of `X` would be copied according to the permutation vector `P`. Thus, the active set determines which elements are sent, not which elements are written, in the result vector.

In making all of these decisions, we found it valuable to compare existing languages, to consult experienced data-parallel programmers, and to write many example programs.[5]

4 DAPPLE implementation notes

There were also several interesting implementation decisions made during the design of DAPPLE. While a complete description of the DAPPLE implementation is beyond the scope of this paper, there are some highlights worth reporting here.

[5]See the examples at URL `http://www.cs.dartmouth.edu/ILI/dapple/examples/`.

Templates. It seems clear that C++ templates would have been perfect for implementing generic Vector and Matrix classes, allowing easy generalization over primitive element classes like `int` and `float`. At the time we developed DAPPLE (May 1994), however, templates were poorly supported by C++ compilers available to us. Furthermore, the compilation model for templates would have required that the entire DAPPLE implementation code (4500 lines of C++) be recompiled for each small student example. Even in our non-template implementation, the overhead of compiling include files and of linking makes DAPPLE slow to use on the Macintosh. Furthermore, there are several operators that are defined for one elemental class and not another; for example, boolean operators are defined for booleanVectors but not for floatVectors, and the *mod* operator (`%`) is only defined for intVectors and intMatrices. These variations would be difficult to express in a template class.

Active sets and ifp. The active set is represented internally by a stack of booleanVectors, each representing the active set at that level of `ifp` nesting. We implement `ifp` as a macro that expands into a two-iteration loop, which updates the active-set stack before and after each iteration. The loop contains an `if` statement that chooses the "then" clause on the first iteration, and the "else" clause on the second iteration. As a result, the active set remains meaningful across function calls embedded in `ifp` clauses, so `ifp` statements may be dynamically nested:

```
main()
{
    intVector A(n);
    ...
    ifp (A > 0)
        do(A);
}

do(intVector A)
{
    intVector B(n);
    ...
    ifp (B == 0)
        cout << A;   // only where A > 0 and B == 0
    else
        cout << B;   // only where A > 0 and B != 0
}
```

Shapes. Every Vector and Matrix is given a *shape* when declared, through parameters to the constructor. In our implementation, Vector and Matrix are both subclasses of Collection, so that

the simpler methods (such as the elementwise operators) can be defined once for Collection and be usable either for Vectors or Matrices. The Collection contains a description of its shape. Each Collection method checks to make sure its operands are the same shape, and creates a result Collection of the same shape.

Matrix slices. A matrix slice (e.g., `M[_][4]` representing column 4 of matrix M) behaves exactly like a vector. We overloaded the `[]` operator for matrices to return a new collection that had the shape of a vector (so that slice-vector operations were possible), but which contained a pointer to the original matrix so that assignment to a slice would update the original matrix. We defined a global variable of a special type and the name `_` (underscore), allowing us to define three types of matrix subscripting: [*int*][*int*] (element), [*int*][_] (row slice), and [_][*int*] (column slice), through different overloads of the `[]` operator.

5 Classroom experience

We taught parallel-computing concepts and DAPPLE to a group of first-year students near the end of Dartmouth's second course in computer science (i.e., CS2). The students had learned C++ programming in this and the preceding course, and had learned the basic data structures (dictionaries, trees, lists, arrays, *etc.*) and algorithms (sorting, searching, *etc.*). The four lectures are outlined in Figure 5. In the first lecture, we motivated parallel computing and introduced some basic concepts, including the notions of parallel complexity, work, speedup, and efficiency. In the second lecture we introduced DAPPLE through a combination of description and simple examples. The third and fourth lectures examined two larger examples in more detail. The first introduced the topic of image processing, specifically edge finding. The second visited the familiar topic of quicksort, by presenting a parallel version of quicksort.

The program to use Canny's algorithm for edge finding demonstrated an embarrassingly parallel program with nearest-neighbor communication. Many students were excited to see a simple program with obvious real-world applications. They were able to appreciate the parallelism and the fact that computer science applies to the real world, often forgotten in the collection of abstract programming problems typical in this course. We developed a colorful animation of this inherently visual application that helped to explain some of the mathematics in the program.

The quicksort program, much like that in Figure 4, demonstrated the complexity of parallel programming, and more sophisticated techniques like scans, permutations, and recursively nested

active sets. We used two demonstrations to help students understand the algorithm. First, we used live demonstrations where the students acted as virtual processors, and second, we used a detailed computer animation to follow the algorithm through an example.

For homework the students were asked to do two small exercises. First, the students were asked to comment on the efficiency of the quicksort algorithm, which clearly loses parallelism in the lower levels of the recursion. Then the students were asked to write a new DAPPLE program to simulate Conway's Game of Life [17, 18], as in Figure 6. As this program has a structure similar to Canny's algorithm, most students had little difficulty. Students used DAPPLE on the Macintosh, in the same programming environment they had used for all of their previous programming.

We presented our class lectures using an overhead projection of a Macintosh computer screen, allowing us to show the animations, view code, compile and run programs, and show lecture notes. All of the lecture notes and examples were formatted in HTML and viewed with Netscape, so they were available to students after class.[6]

Student response. We surveyed the students at the end of the lectures, and examined the DAPPLE programs they submitted. Of 26 students in the class, 20 returned the survey. All found the module to be interesting and useful. They particularly enjoyed the way it made them look at computing in a different way, and the way we were able to relate it to real-world computing needs. Several felt that it would have been better if we had used a real parallel computer, despite the fact that the language and concepts would have been no different.

Most found our pace too slow; we clearly underestimated their ability to follow complex material.

The animations were "extremely" useful, even "vital," to understanding the algorithms. While it is not surprising that these animations were useful to help students visualize the algorithms, a student assistant spent about fifty hours building the animations. More interestingly, the class was divided on whether the "live" demonstrations, where students acted out algorithms, were useful. Other researchers have successfully used such live demonstrations to teach parallel computing [19], even to high-school students [2]. Most of our students liked the demonstrations, but one even said they were "degrading," yet another example of how students prefer to learn in different ways.

Students were also mixed on the use of HTML/Netscape lecture notes. Some found it useful to have the lecture notes easily available after class from anywhere on campus. Others requested printed notes. Several pointed out that what works best in class (outline form with little detail)

[6] And to you: see URL `http://www.cs.dartmouth.edu/ILI/dapple/lectures/`.

is not what is needed outside of class (details). Perhaps an outline format for lectures, with more links to details that can be viewed outside class, would be best.

Several students were excited by the material and requested that we expand the module into a full class. Others recommended that we spread the same amount of material over several weeks, with small DAPPLE exercises interspersed so they could get a better feel for the concepts and language. Finally, some wanted to use real parallel computers and to see performance charts of speedup on real applications.

We also examined the "Game of Life" DAPPLE programs completed by 23 students, most of which were similar to Figure 6. They had no trouble using the `<<` and `>>` operators to input and output their boards, to use nested `if`s, and to use a reduction to compute the number of live cells. Some used more advanced features, such as type casting and matrix slices. Unfortunately, four students seemed to miss the point, using a doubly-nested `for` loop to sequentially update individual matrix elements. One student used recursion instead of iteration to step through the generations (!), although it seems this problem was not related to parallel concepts. The only major complaint had to do with the long compilation times (due to the large DAPPLE "include" files).

6 Summary and conclusions

We set out to teach parallel computing to first-year students, because we believe that it is important to teach the concepts of parallel computing throughout the curriculum. Since we felt that it was important to provide a hands-on programming experience, we needed a parallel-programming language that was conceptually simple, easy to learn, not too different from the students' existing language, free, and available on the students' computers. In this paper we survey some of the potential languages, and then describe our efforts to design and implement a suitable language. The object-oriented capabilities of C++, which was the language students were using, allowed us to build a reasonable parallel-programming language from macros and a class library. We describe some of the design decisions involved in the language and its implementation, and then our experiences using the language in the classroom.

We decided that it was important to expose some aspects of parallelism to the students, but not all of the complexities. Thus, we decided to use a data-parallel programming model that was based on the concept of virtual processors, but (due to the single thread of control) does not require students to deal with asynchrony, deadlock, or even explicit communication. Our language, DAPPLE, is based on C++ (which the students know) and provides parallel operations

on collections and a parallel `if` statement.

The students universally found the material interesting and useful, and were able to complete simple parallel-programming assignments after four lectures. We found that computer animations and live demonstrations contributed significantly to student understanding and interest. Since many students commented on how much they enjoyed the opportunity to look at computing from a different perspective, we feel that the addition of this module to the curriculum may actually increase the entry of students into the computer-science major. Since our biggest mistake in the classroom experiments was in underestimating the ability of the students to pick up new material quickly, we are confident that first-year students could use DAPPLE for deeper explorations of parallel computing.

Availability

DAPPLE currently runs on Unix workstations (at least Sun, SGI, DEC Ultrix, and DEC Alpha) using the g++ compiler, and on the Macintosh using Symantec C++ 7.04. The animations run only on the Macintosh. The complete package (code, documentation, tutorial, examples, lecture notes, and animations) is available on the WWW at URL `http://www.cs.dartmouth.edu/ILI/dapple/`.

References

[1] R. Miller, "The status of parallel processing education" *IEEE Computer*, Vol. 27, No. 8, August 1994, pp. 40–43.

[2] A. Rifkin, "Teaching parallel programming and software engineering concepts to high school students", *SIGCSE Technical Symposium on Computer Science Education*, 1994, pp. 26–30.

[3] D. Johnson, D. Kotz, and F. Makedon, "Teaching parallel computing to freshmen", *Conference on Parallel Computing for Undergraduates*. Edited by C. Nevison, Colgate University, June 1994.

[4] G. E. Blelloch, "NESL: a nested data-parallel language", Technical Report CMU-CS-93-129, Carnegie Mellon University, Pittsburgh, PA, April 1993.

[5] D. B. Loveman, "High Performance Fortran" *IEEE Parallel and Distributed Technology*, Vol. 1, No. 1, February 1993, pp. 25–42.

[6] J. R. Larus, B. Richards, and G. Viswanathan, "C**: A large-grain, object-oriented, data-parallel programming language", Technical Report #1126, University of Wisconsin-Madison, November 1992.

[7] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. X. Yang, "Distributed pC++: basic ideas for an object parallel language" *Scientific Programming*, Vol. 2, No. 3, Fall 1993, pp. 7–22.

[8] M. F. Kilian, *Parallel Sets: An Object-oriented Methodology for Massively Parallel Programming*, PhD thesis, Harvard University, Cambridge, MA, 1992.

[9] K. M. Chandy and C. Kesselman, "Compositional C++: Compositional parallel programming", Technical Report CS-TR-92-13, California Institute of Technology, Pasadena, CA, 1992.

[10] A. S. Grimshaw, "Easy-to-use object-oriented parallel processing with Mentat" *IEEE Computer*, Vol. 26, No. 5, May 1993, pp. 39–51.

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

[12] R. Chandra, A. Gupta, and J. L. Hennessey, "COOL: an object-based language for parallel programming" *IEEE Computer*, Vol. 27, No. 8, August 1994, pp. 14–26.

[13] E. A. West and A. S. Grimshaw, "Braid: Integrating task and data parallelism", *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, February 1995, pp. 211–219.

[14] T. J. Sheffler and S. Chatterjee, "An object-oriented approach to nested data parallelism", *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, February 1995, pp. 203–210.

[15] T. J. Sheffler, "A portable MPI-based parallel vector template library", Technical Report 95.04, RIACS, NASA Ames Research Center, Moffett Field, CA, 1995.

[16] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA, 1990, Ninth printing.

[17] M. Gardner, "Mathematical games" *Scientific American*, Vol. 223, No. 10, October 1970, pp. 120–123.

[18] M. Gardner, "Mathematical games" *Scientific American*, Vol. 224, No. 2, February 1971, pp. 112–117.

[19] A. T. Kitchen, N. Shaller, and P. Tymann, "Game playing as a technique for teaching parallel computing concepts" *SIGCSE Bulletin*, September 1992, pp. 35–38, 44, 50.

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

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

// first row                        //    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;

// N-1 remaining rows
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;
}                                   //  }
```

Figure 1: A DAPPLE program to compute Pascal's triangle.

```
// we divide the x range up into N intervals,
// computing the function at N points
const int N = 1000;

// which virtual processor am I (0, 1, 2, ... N-1)?
const intVector VP(N, Identity);

const double width = 1.0 / N; // width of each interval
doubleVector X(N), Y(N);      // X points, Y=f(X)

// we compute pi as the integral of 4 / (1 + x^2) for x from 0 to 1
// we do this by using the rectangle rule to approximate the integral
X = (floatVector(VP) + 0.5) * width; // find the midpoint of each interval
Y = 4.0 / (1.0 + X * X);             // compute the function there

// sum the area of each rectangle (Y is the height of the rectangle)
cout << "pi ~= " << sum (width * Y) << endl;
```

Figure 2: A DAPPLE program to estimate π as $\int_0^1 \frac{4}{1+x^2}$.

```
// 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 3: A matrix-matrix multiplication program in DAPPLE.

```
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 4: A quicksort function in DAPPLE.

Lecture 1: Parallel computing

- Motivation for parallel computing
  - high-performance computing
  - limitations of current hardware development trends
- What kinds of problems are naturally parallel?
- What kinds of problems are *not* parallel?
- Live demo: students as processors
- Data parallelism and task parallelism
- Parallel complexity
  - work, speedup, efficiency
- Amdahl's Law

Lecture 2: DAPPLE

- Vector and Matrix
- virtual processors
- assignment, arithmetic, reductions
- `ifp`
- many examples, including Figure 1 and Figure 2

Lecture 3: Canny's algorithm example

- importance of image processing
- digital representation of an image
- edge finding
- Canny's algorithm for edge finding
  - explanation
  - visual animation
  - DAPPLE program
- preparation for next lecture: odd-even sorting
  - live class demo

Lecture 4: Quicksort

- review sequential quicksort
- parallel quicksort method
  - live class demo
  - visual animation
  - DAPPLE program, Figure 4

Homework:

- Conway's Game of Life

Figure 5: A four-lecture module on parallel computing and DAPPLE for Dartmouth's second computer-science course.

```
cout << "Enter number of rows:" << endl;
int n;  cin >> n;
cout << "Enter number of columns:" << endl;
int m;  cin >> m;

cout << "Enter initial state (" << n << " rows by " << m << " columns, ";
cout << "using 'O' for a live cell, '.' for an empty cell):" << endl;
charMatrix printable(n,m);
cin >> printable;

// note that because we use shift(), we assume boundaries of 0
// if we used rotate, we would get periodic boundary conditions (torus)
int gen = 0;
while (true) {
    // 1 means life there, 0 means nothing
    intMatrix cells(n,m), neighbors(n,m);

    ifp (printable == '.')
      cells = 0;
    else
      cells = 1;

    neighbors = (  shift(cells,  0, -1) + shift(cells,  0,  1)
                 + shift(cells, -1, -1) + shift(cells, -1,  0)
                 + shift(cells, -1,  1) + shift(cells,  1, -1)
                 + shift(cells,  1,  0) + shift(cells,  1,  1));

    // The Rules:
    // If an organism has 0 or 1 neighbors, it dies of loneliness.
    // If an organism has 2 or 3 neighbors, it survives.
    // If an organism has 4 or more neighbors, it dies of overcrowding.
    // If an empty cell has exactly 3 neighbors, a new organism is born.
    ifp (neighbors != 2)
      ifp (neighbors == 3)
        cells = 1;              // either existing survives, or new is born
      else
        cells = 0;              // either lonely or overcrowded
    else
      ; // no change with two neighbors

    cout << "Generation " << ++gen << ", " << sum(cells) << " alive" << endl;
    ifp (cells == 1)
      printable = 'O';
    else
      printable = '.';
    cout << printable << endl;
}
```

Figure 6: A DAPPLE program for Conway's game of Life. This game simulates a simple cellular automaton, in which each cell in a 2-d matrix is either occupied by an organism or unoccupied, and in each generation organisms are born or killed according to simple localized rules. The program here is a typical solution, similar to most student programs.

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 ...`		