27. Shan Y, Klepeis JL, Eastwood MP, Dror RO, Shaw DE (2005) Gaussian split Ewald: a fast Ewald mesh method for molecular simulation. J Chem Phys 122:054101
28. Shaw DE (2005) A fast, scalable method for the parallel evaluation of distance-limited pairwise particle interactions. J Comput Chem 26:1318–1328
29. Shaw DE, Deneroff MM, Dror RO, Kuskin JS, Larson RH, Salmon JK, Young C, Batson B, Bowers KJ, Chao JC, Eastwood MP, Gagliardo J, Grossman JP, Ho CR, Ierardi DJ, Kolossváry I, Klepeis JL, Layman T, McLeavey C, Moraes MA, Mueller R, Priest EC, Shan Y, Spengler J, Theobald M, Towles B, Wang SC (2007) Anton: a special-purpose machine for molecular dynamics simulation. In: Proceedings of the 34th annual international symposium on computer architecture (ISCA '07). ACM, New York
30. Shaw DE, Dror RO, Salmon JK, Grossman JP, Mackenzie KM, Bank JA, Young C, Deneroff MM, Batson B, Bowers KJ, Chow E, Eastwood MP, Ierardi DJ, Klepeis JL, Kuskin JS, Larson RH, Lindorff-Larsen K, Maragakis P, Moraes MA, Piana S, Shan Y, Towles B (2009) Millisecond-scale molecular dynamics simulations on Anton. In: Proceedings of the conference for high performance computing, networking, storage and analysis (SC09). ACM, New York
31. Shaw DE, Maragakis P, Lindorff-Larsen K, Piana S, Dror RO, Eastwood MP, Bank JA, Jumper JM, Salmon JK, Shan Y, Wriggers W (2010) Atomic-level characterization of the structural dynamics of proteins. Science 330:341–346
32. Stone JE, Phillips JC, Freddolino PL, Hardy DJ, Trabuco LG, Schulten K (2007) Accelerating molecular modeling applications with graphics processors. J Comput Chem 28:2618–2640
33. Taiji M, Narumi T, Ohno Y, Futatsugi N, Suengaga A, Takada N, Konagaya A (2003) Protein Explorer: a petaflops special-purpose computer system for molecular dynamics simulations. In: Proceedings of the ACM/IEEE conference on supercomputing (SC '03), Phoenix, AZ. ACM, New York
34. Toyoda S, Miyagawa H, Kitamura K, Amisaki T, Hashimoto E, Ikeda H, Kusumi A, Miyakawa N (1999) Development of MD Engine: high-speed accelerator with parallel processor design for molecuar dynamics simulations. J Comput Chem 2:185–199
35. Young C, Bank JA, Dror RO, Grossman JP, Salmon JK, Shaw DE (2009) A 32 × 32 × 32, spatially distributed 3D FFT in four microseconds on Anton. In: Proceedings of the conference for high performance computing, networking, storage and analysis (SC09). ACM, New York

Application-Specific Integrated Circuits

►VLSI Computation

Applications and Parallelism

►Computational Sciences

Architecture Independence

►Network Obliviousness

Area-Universal Networks

►Universality in VLSI Computation

Array Languages

Calvin Lin
University of Texas at Austin, Austin, TX, USA

Definition

An array language is a programming language that supports the manipulation of entire arrays – or portions of arrays – as a basic unit of operation.

Discussion

Array languages provide two primary benefits: (1) They raise the level of abstraction, providing conciseness and programming convenience; (2) they provide a natural source of data parallelism, because the multiple elements of an array can typically be manipulated concurrently. Both benefits derive from the removal of control flow. For example, the following array statement assigns each element of the B array to its corresponding element in the A array:

```
A := B;
```

This same expression could be expressed in a scalar language using an explicit looping construct:

```
for i := 1 to n
   for j := 1 to n
      A[i][j] := B[i][j];
```

The array statement is conceptually simpler because it removes the need to iterate over individual array elements, which includes the need to name individual elements. At the same time, the array expression admits more parallelism because it does not over-specify the order in which the pairs of elements are evaluated; thus,

the elements of the B array can be assigned to the elements of the A array in any order, provided that they obey array language semantics. (Standard array language semantics dictate that the righthand side of an assignment be fully evaluated before its value is assigned to the variable on the lefthand side of the assignment.)

For the above example, the convenience of array operators appears to be minimal, but in the context of a parallel computer, the benefits to programmer productivity can be substantial, because a compiler can translate the above array statement to efficient parallel code, freeing the programmer from having to deal with many low-level details that would be necessary if writing in a lower-level language, such as MPI. In particular, the compiler can partition the work and compute loop bounds, handling the messy cases where values do not divide evenly by the number of processors. Furthermore, if the compiler can statically identify locations where communication must take place, it can allocate extra memory to cache communicated values, and it can insert communication where necessary, including any necessary marshalling of noncontiguous data. Even for shared memory machines, the burden of partitioning work, inserting appropriate synchronization, etc., can be substantial.

Array Indexing

Array languages can be characterized by the mechanisms that they provide for referring to portions of an array. The first array language, APL, provided no method of accessing portions of an array. Instead, in APL, all operators are applied to all elements of their array operands.

Languages such as Fortran 90 use array *slices* to concisely specify indices for each dimension of an array. For example, the following statement assigns the upper left 3×3 corner of array `A` to the upper left corner of array `B`.

```
B(1:3,1:3) = A(1:3,1:3)
```

The problem with slices is that they introduce considerable redundancy, and they force the programmer to perform index calculations, which can be error prone. For example, consider the Jacobi iteration, which computes for each array element the average of its four nearest neighbors:

```
B(2:n,2:n) = (A(1:n-1,2:n)+
              A(3:n+1,2:n)+
              A(2:n,  1:n-1)+
              A(2:n,  3:n+3)+
```

The problem becomes much worse, of course, for higher-dimensional arrays.

The C* language [13], which was designed for the Connection machine, simplifies array indexing by defining indices that are relative to each array element. For example, the core of the Jacobi iteration can be expressed in C* as follows, where `active` is an array that specifies the elements of the array where the computation should take place:

```
where (active)
{
  B = ([.-1][.]A + [.+1][.]A
       + [.][.-1]A + [.][.+1]A)/4;
}
```

C*'s relative indexing is less error prone than slices: Each relative index focuses attention on the differences among the array references, so it is clear that the first two array references refer to the neighbors above and below each element and that the last two array references refer to the neighbors to the left and right of each element.

The ZPL language [4] further raises the level of abstraction by introducing the notion of a *region* to represent index sets. Regions are a first-class language construct, so regions can be named and manipulated. The ability to name regions is important because – for software maintenance and readability reasons – descriptive names are preferable to constant values. The ability to manipulate regions is important because – like C*'s relative indexing – it defines new index sets relative to existing index sets, thereby highlighting the relationship between the new index set and the old index set.

To express the Jacobi iteration in ZPL, programmers would use the At operator (`@`) to translate an index set by a named vector:

```
[R]   B := (A@north + A@south +
            A@west +  A@east)/4;
```

The above code assumes that the programmer has defined the region `R` to represent the index set $[1{:}n][1{:}n]$, has defined `north` to be a vector whose value is $[-1, 0]$, has defined `south` to be a vector

whose value is [+1,0], and so forth. Given these definitions, the region `R` provides a base index set [1:*n*][1:*n*] for every reference to a two-dimensional array in this statement, so `R` applies to every occurrence of `A` in this statement. The direction `north` shifts this base index set by −1 in the first dimension, etc. Thus, the above statement has the same meaning as the Fortran 90 slice notation, but it uses named values instead of hard-coded constants.

ZPL provides other region operators to support other common cases, and it uses regions in other ways, for example, to declare array variables. More significantly, regions are quite general, as they can represent sparse index sets and hierarchical index sets.

More recently, the Chapel language from Cray [5] provides an elegant form of relative indexing that can take multiple forms. For example, the below Chapel code looks almost identical to the ZPL equivalent, except that it includes the base region, R, in each array index expression:

```
T[R] = (A[R+north] + A[R+south]+
        A[R+east] +  A[R+west])/4.0;
```

Alternatively, the expression could be written from the perspective of a single element in the index set:

```
[ij in R] T(ij) = (A(ij+north) +
                   A(ij+south) +
                   A(ij+east) +
                   A(ij+west))/4.0;
```

The above formulation is significant because it allows the variable `ij` to represent an element of an arbitrary tuple, where the tuple (referred to as a *domain* in Chapel) could represent many different types of index sets, including sparse index sets, hierarchical index sets, or even nodes of a graph.

Finally, the following Chapel code fragment shows that the tuple can be decomposed into its constituent parts, which allows arbitrary arithmetic to be performed on each part separately:

```
[(i,j) in R]  A(ij) = (A(i-1,j)+
                       A(i+1,j)+
                       A(i,j-1)+
                       A(i,j+1))/4.0;
```

The FIDIL language [9] represents an early attempt to raise the level of abstraction, providing support for irregular grids. In particular, FIDIL supports index sets, known as *domains*, which are first-class objects that can be manipulated through union, intersection, and difference operators.

Array Operators

Array languages can also be characterized by the set of array operators that they provide. All array languages support elementwise operations, which are the natural extension of scalar operators to array operands. The Array Indexing section showed examples of the elementwise + and = operators.

While element-wise operators are useful, at some point, data parallel computations need to be combined or summarized, so most array languages provide reduction operators, which combine – or reduce – multiple values of an array into a single scalar value. For example, the values of an array can be reduced to a scalar by summing them or by computing their maximum or minimum value.

Some languages provide additional power by allowing reductions to be applied to a subset of an array's dimensions. For example, each row of values in a two-dimensional array can be reduced to produce a single column of values. In general, a partial reduction accepts an *n*-dimensional array of values and produces an *m*-dimensional array of values, where $m < n$. This construct can be further generalized by allowing the programmer to specify an arbitrary associative and commutative function as the reduction operator.

Parallel prefix operators – also known as scan operators – are an extension of reductions that produce an array of partial values instead of a single scalar value. For example, the prefix sum accepts as input *n* values and computes all sums, $x_0 + x_1 + x_2 + \ldots + x_k$ for $0 \leq k \leq n$. Other parallel prefix operations are produced by replacing the + operator with some other associative operator. (When applied to multi-dimensional arrays, the array indices are linearized in some well-defined manner, e.g., using Row Major Order.)

The parallel prefix operator is quite powerful because it provides a general mechanism for parallelizing computations that might seem to require sequential iteration. In particular, sequential loop iterations that accumulate information as they iterate can typically be solved using a parallel prefix.

Given the ability to index an individual array element, programmers can directly implement their own reduction and scan code, but there are several benefits of language support. First, reduction and scan are common abstractions, so good linguistic support makes these abstractions easier to read and write. Second, language support allows their implementations to be customized to the target machine. Third, compiler support introduces nontrivial opportunities for optimization: When multiple reductions or scans are performed in sequence, their communication components can be combined to reduce communication costs.

Languages such as Fortran 90 and APL also provide additional operators for flattening, re-shaping, and manipulating arrays in other powerful ways. These languages also provide operators such as matrix multiplication and matrix transpose that treat arrays as matrices. The inclusion of matrix operations blurs the distinction between array languages and matrix languages, which are described in the next section.

Matrix Languages

Array languages should not be confused with matrix languages, such as Matlab [7]. An array is a programming language construct that has many uses. By contrast, a matrix is a mathematical concept that carries additional semantic meaning. To understand this distinction, consider the following statement:

```
A = B * C;
```

In an array language, the above statement assigns to A the element-wise product of B and C. In a matrix language, the statement multiples B and C.

The most popular matrix language, Matlab, was originally designed as a convenient interactive interface to numeric libraries, such as EISPACK and LINPACK, that encourages exploration. Thus, for example, there are no variable declarations. These interactive features make Matlab difficult to parallelize, because they inhibit the compiler's ability to carefully communication the computation and communication.

Future Directions

Array language support can be extended in two dimensions. First, the restriction to flat dense arrays is too limiting for many computations, so language support for sparsely populated arrays, hierarchical arrays, and irregular pointer-based data structures are also needed. In principle, Chapel's domain construct supports all of these extensions of array languages, but further research is required to fully support these data structures and to provide good performance. Second, it is important to integrate task parallelism with data parallelism, as is being explored in languages such as Chapel and X10.

Related Entries

►Chapel (Cray Inc. HPCS Language)
►Fortran 90 and Its Successors
►HPF (High Performance Fortran)
►NESL
►ZPL

Bibliographic Notes and Further Reading

The first array language, APL [10], was developed in the early 1960s and has often been referred to as the first write-only language because of its terse, complex nature.

Subsequent array languages that extended more conventional languages began to appear in the late 1980s. For example, extensions of imperative languages include C* [13], FIDIL [9], Dataparallel C [8], and Fortran 90 [1]. NESL [3] is a functional language that includes support for nested one dimensional arrays.

In the early 1990s, High Performance Fortran (HPF) was a data parallel language that extended Fortran 90 and Fortran 77 to provide directives about data distribution. At about the same time, the ZPL language [12] showed that a more abstract notion of an array's index set could lead to clear and concise programs.

More recently, the DARPA-funded High-Productivity Computing Systems project led to the development of Chapel [5] and X10 [6], which both integrate array languages with support for task parallelism.

Ladner and Fischer [11] presented key ideas of the parallel prefix algorithm, and Blelloch [2] elegantly demonstrated the power of the scan operator for array languages.

Bibliography

1. Adams JC, Brainerd WS, Martin JT, Smith BT, Wagener JL (1992) Fortran 90 handbook. McGraw-Hill, New York
2. Blelloch G (1996) Programming parallel algorithms. Comm ACM 39(3):85–97

3. Blelloch GE (1992) NESL: a nested data-parallel language. Technical Report CMUCS-92-103, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1992
4. Chamberlain BL (2001) The design and implementation of a region-based parallel language. PhD thesis, University of Washington, Department of Computer Science and Engineering, Seattle, WA
5. Chamberlain BL, Callahan D, Zima HP (2007) Parallel programmability and the Chapel language. Int J High Perform Comput Appl 21(3):291–312
6. Ebcioglu K, Saraswat V, Sarkar V (2004) X10: programming for hierarchical parallelism and non-uniform data access. In: International Workshop on Language Runtimes, OOPSLA 2004, Vancouver, BC
7. Amos Gilat (2003) MATLAB: an introduction with applications, 2nd edn. Wiley, New York
8. Hatcher PJ, Quinn MJ (1991) Data-parallel programming on MIMD computers. MIT Press, Cambridge, MA
9. Hilfinger PN, Colella P (1988) FIDIL: a language for scientific programming. Technical Report UCRL-98057, Lawrence Livermore National Laboratory, Livermore, CA, January 1988
10. Iverson K (1962) A programming language. Wiley, New York
11. Ladner RE, Fischer MJ (1980) Parallel prefix computation. JACM 27(4):831–838
12. Lin C, Snyder L (1993) ZPL: an array sublanguage. In: Banerjee U, Gelernter D, Nicolau A, Padua D (eds) Languages and compilers for parallel computing. Springer-Verlag, New York, pp 96–114
13. Rose JR, Steele Jr GL (1987) C*: an extended C language for data parallel programming. In: 2nd International Conference on Supercomputing, Santa Clara, CA, March 1987

Array Languages, Compiler Techniques for

Jenq-Kuen Lee[1], Rong-Guey Chang[2], Chi-Bang Kuan[1]
[1]National Tsing-Hua University, Hsin-Chu, Taiwan
[2]National Chung Cheng University, Chia-Yi, Taiwan

Synonyms

Compiler optimizations for array languages

Definition

Compiler techniques for array languages generally include compiler supports, optimizations and code generation for programs expressed or annotated by all kinds of array languages. These compiler techniques mainly take advantage of data-parallelism explicit in array operations and intrinsic functions to deliver performance for parallel architectures, such as vector processors, multi-processors and VLIW processors.

Discussion

Introduction to Array Languages

There are several programming languages providing a rich set of array operations and intrinsic functions along with array constructs to assist data-parallel programming. They include Fortran 90, High Performance Fortran (HPF), APL and MATLAB, etc. Most of them provide programmers array intrinsic functions and array operations to manipulate data elements of multidimensional arrays concurrently without requiring iterative statements. Among these array languages, Fortran 90 is a typical example, which consists of an extensive set of array operations and intrinsic functions as shown in Table 1. In the following paragraphs, several examples will be provided to bring readers basic information about array operations and intrinsic functions supported by Fortran 90. Though in the examples only Fortran 90 array operations are used for illustration, the array programming concepts and compiler techniques are applicable to common array languages.

Fortran 90 extends former Fortran language features to allow a variety of scalar operations and intrinsic functions to be applied to arrays. These array operations and intrinsic functions take array objects as inputs, perform a specific operation on array elements concurrently, and return results in scalars or arrays. The code fragment below is an array accumulation example, which involves two two-dimensional arrays and one array-add operation. In this example, all array elements in the array *S* is going to be updated by accumulating by those of array *A* in corresponding positions. The accumulation result, also a two-dimensional 4×4 array, is at last stored back to array *S*, in which each data element contains element-wise sum of array *A* and array *S*.

```
integer  S(4,4),  A(4,4)

S = S + A
```

Besides primitive array operations, Fortran 90 also provides programmers a set of array intrinsic functions to manipulate array objects. These intrinsic functions listed in Table 1 include functions for data movement,