Parallel Programming Languages

Keshav Pingali,
Department of Computer Science,
Cornell University,
Ithaca, NY 14853.

September 1, 1998

Contents

1 Introduction

A parallel programming language is a formal notation for algorithm expression whose meaning can be defined by appealing to a parallel computational model.

Parallel programming languages have more complicated data and control models than sequential programming languages. The data model in sequential languages is the Random Access Memory (RAM) model in which there is a single address space of memory locations that can be read and written by the processor. The analog in parallel languages is the *shared-memory* model in which all memory locations reside in a single address space and are accessible to all the processors (the word "processors" in this article always refers to the logical processors in the underlying parallel execution model of the language, and not to hardware processors). A more decoupled data model is provided by the *distributed-memory* model in which each processor has its own address space of memory locations inaccessible to other processors. The choice of the data model determines how processors communicate with each other — in a shared-memory model, they communicate by reading and writing shared locations, but in a distributed-memory model, they communicate by sending and receiving messages.

The control model in a parallel programming language determines how processors are co-ordinated. The simplest parallel control model is *lock-step* (vector) *synchronization*. At every step of program execution, each processor is either turned off or is required to perform the same operation as all other processors. The active processors at each step work on different data items, so this control model is also called the *Single Instruction Multiple Data (SIMD)* model. SIMD-style parallel execution can be exploited in performing vector operations like adding or multiplying the elements of a set of vectors. *Bulk synchronization* is a more decoupled control model in which processors synchronize occasionally by using a *barrier* instruction. No processor is allowed to execute a statement past a barrier until all processors have arrived at that barrier. In between the execution of successive barrier statements, processors are autonomous and may execute different operations. Bulk synchronization can be used to exploit the *data parallelism* that arises when a function f is applied to each of the elements of a data structure such as a vector. All evaluations of f can be performed in parallel, so processors synchronize only at the beginning and end of this computation. Since f may have conditionals inside it, the processors may end up performing different computations, which is permitted in the bulk synchronous model. The most decoupled form of synchronization is *fine-grain synchronization* in which two or more processors can synchronize on their own whenever they need to, without involving other processors. This form of parallel execution is sometimes called MIMD (Multiple Instruction Multiple Data) parallelism. The MIMD model is appropriate for exploiting *task* parallelism which arises when autonomous tasks execute concurrently, synchronizing only for exclusive access to resources or for co-ordinating access to data that is being produced and consumed concurrently by different tasks.

Figure 1 classifies the languages discussed in this article according to their control and data models. A survey of parallel programming languages can be found in [39].

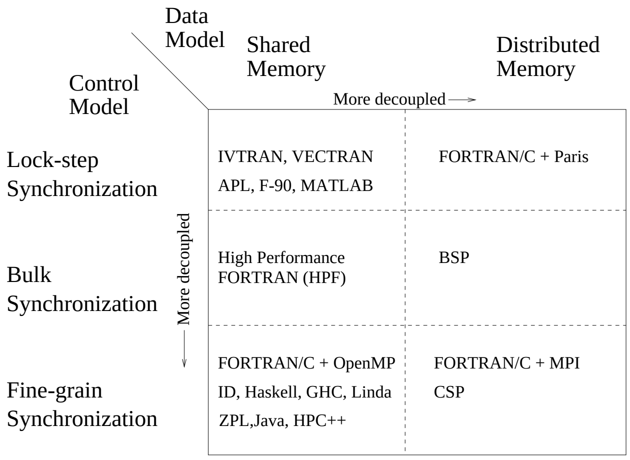

Figure 1: A Classification of Parallel Programming Languages

2 Lock-step Synchronous Parallel Languages

Lock-step (SIMD) parallel languages are used mainly to program vector and array processors for performing scientific computations in which dense matrices are the primary data structures. Not surprisingly, most of these languages are extensions of FORTRAN. Programs in these languages contain a combination of scalar and vector operations. On array processors like CM-2 [41], the scalar operations are usually performed by a front-end high performance work-station, while the vector operations are performed on the array processor. Vector processors like the CRAY [11] can execute both scalar and vector instructions. Therefore, the key problem in designing a SIMD language is to design constructs that expose as many vector operations as possible to the compiler.

2.1 Shared-Memory SIMD Languages

The simplest vector operations involve the application of an arithmetic or boolean function to each element of an array (or arrays), such computing the elementwise sum of two arrays. These operations can be expressed quite simply by overloading arithmetic and boolean operators. For example, the FORTRAN-90 [4] statement `C = A + B` specifies that the elementwise sum of arrays `A` and `B` is to be stored into array `C`.

In many applications however, vector operations must performed on some but not all of the elements of an array. For example, in solving partial differential equations, it may be necessary to apply one operator to points in the interior of the domain and a different one to points at the boundaries. Operator overloading is not sufficient to permit the expression of conditional vector operations, so a variety of constructs for describing sparse index sets have been invented.

Many SIMD languages provide the programmer with constructs for specifying the array section on which the vector operation is to be performed. One approach is to use *control vectors*, first introduced in the Burroughs Illiac IV FORTRAN language [32] — a value of *true* in a control vector indicates that the vector operation should be performed for the corresponding data element. A `*` indicates a control vector of arbitrary length in which all elements are *true*. The following code shows the use of control vectors in this language. The first array statement adds the elements of the `A` and `C` arrays pointwise and assigns the results to `A`. Because only the odd elements of `B` are true, the second array assignment adds only the odd elements of `A` and `C` and assigns the results to odd elements of `A`.

```
   do 10 i = 1, 100,2
     B(i) = .true.
     B(i+1) = .false.
10 continue
   A(*) = A(*) + C(*)
   A(B(*)) = A(B(*)) + C(B(*))
```

An important special case of conditional vector operations is *constant stride* vector operations in which the elementary operations are applied to every k^{th} element of the vector(s) for some integer k. On many vector computers, it is difficult to generate efficient code for these operations if control vectors are used. The operands and results of vector operations are usually stored in memory, so it is usually not worth performing an arithmetic operation in vector mode unless the loads and stores can also be done in vector mode. However, some computers (such as the CRAY-1 and CRAY-2 [11]) permit only constant-stride loads and stores from memory. Unless the compiler can determine that the *true* entries in a control vector occur with a fixed stride, it is forced to generate scalar loads and stores.

The IBM VECTRAN language [36] addressed this problem by introducing *array triplets* which can describe many constant stride array sections. An array triplet consists of three expressions separated by colons, and specifies the start, end and stride of the range of execution of a statement. If the stride is 1, the last expression and its preceding colon can be omitted. There is an obvious similarity between triplets and the specification of DO loop index sets in FORTRAN. The following code shows a use of triplets. After the last statement is executed, `A(2)` contains 1, `A(4)` contains 2 etc. Multi-dimensional arrays can be handled by using a triplet for each dimension of the array. Array triplet notation is also used in other array languages like MATLAB [30] and FORTRAN-90 [4]

```
   do 10 i = 1, 10
10    A(i) = i
   A(2:10:2) = A(1:5)
```

Although triplet notation is powerful, it is not a replacement for control vectors since it cannot describe arbitrary index sets. Therefore, Vectran supplemented the triplet notation with `where` statements, an example of which is given below.

```
   where (A(1:100) .LT. 0)
          A(1:100) = - A(1:100)
   otherwise
          A(1:100) = 0.0
   endwhere
```

The `where` statement first evaluates a logical array expression. Statements in the body of `where` are executed for each index for which the logical array expression is true, while statements in the `otherwise` clause are executed for indices for which the logical expression is false. The clauses can contain only assignment statements.

The approaches described so far for expressing conditional vector operations are data-oriented in the sense they require the programmer to specify the array section on which the vector operation must be performed. A complementary approach is to embellish the control constructs in the language. One such construct, which was introduced in the IVTRAN language [33], is the `forall` statement in which the sparse index set is specified in terms of the control variables of the loop. The following code shows an example of its use. The loop has a two-dimensional index space in which all iterations can be performed in parallel, and in each iteration `(i,j)`, the assignment is performed if `A(i,j)` is less than zero. Note that the `forall` construct permits assignment to constant-stride array sections like diagonals which cannot be described using triplet notation.

```
forall (i = 1:100:2, j = 1:100, A(i,j) .LT. 0)
   A(i,j) = B(i,j)
```

Although reduction operations like adding all the elements of a vector can also be done in vector mode, shared-memory SIMD languages have traditionally not had constructs to support these operations. However, most of them provide library routines that can be invoked by the programmer to perform reduction operations in vector mode.

Other shared-memory vector languages are LRLTRAN [45] from Lawrence Livermore Laboratories, BSP FORTRAN [5] from Burroughs, and Cedar FORTRAN [21] from the University of Illinois, Urbana. Cedar FORTRAN permitted the expression of both SIMD and MIMD parallelism.

2.2 Distributed-Memory SIMD Languages

The CM-2 machine [41] from Thinking Machines was a distributed-memory array processor and its assembly language, called Paris (Parallel Instruction Set) [40], had FORTRAN, C and Lisp interfaces that permit programmers to write high-level language programs with Paris commands embedded in them. The resulting languages were called FORTRAN/Paris, C/Paris and Lisp/Paris, and they are examples of distributed-memory SIMD languages.

The programming model of Paris has an unbounded number of *virtual processors* (VP's) which can be configured into Cartesian grids of various sizes. Each VP has *local memory* for storing data, a *context-flag* that controls instruction execution, and a unique *address* that can be used by other VP's to send it messages. Each VP can perform the usual arithmetic and logical operations, taking operands from its local memory and storing the result back there (one of the operands can be an immediate constant which is broadcast from the front-end processor). The execution of these operations can be made conditional on the context-flag. Inter-processor communication is performed by executing the `send` instruction. Since processors operate in lock-step, a separate instruction for receiving messages is not required; rather, the execution of the `send` instruction results in data transfer from the source VP to the destination VP. Therefore, the `send` instruction has to specify the address of the receiving processor, and the memory addresses of the source and destination locations of the message. A given VP may receive messages from several other VP's during a `send` operation. If so, the data in these messages can be combined using a reduction operator specified in the `send` instruction. A noteworthy feature of Paris is that it was the first SIMD language to include a rich set of instructions for performing global reductions and parallel prefix operations on data stored in the VP's.

3 Bulk Synchronous Parallel Languages

Lock-step synchronization provides a simple programming model but it can be inefficient for programs with many data-dependent conditionals. Since processors operate in lock-step, every processor must participate in the execution of *both* clauses of a conditional statement even though it performs computations in only *one* of the clauses. Bulk synchronization is a more relaxed synchronization model in which processors execute instructions autonomously but must rendezvous at intervals by executing a *barrier* instruction. No processor can execute an instruction past a barrier until *all* processors have arrived at that barrier. The interval between two successive barriers is called a *super-step*.

The requirement that all processors rendezvous at all barriers means that the most natural approach to programming in this model is to require all processors to execute the same program even though they can take different paths through that program to arrive at the same sequence of barrier instructions. This approach is sometimes called Single Program Multiple Data (SPMD) parallelism, but this term has been abused sufficiently that we will not use it any further in this article.

Bulk synchronization is appropriate for exploiting *data parallelism* in programs. The simplest kind of data parallelism arises when a function is applied to each element of a data structure (like *mapcar* in LISP). A more subtle form of data parallelism arises when an associative operation like addition or multiplication is used to combine all the elements of a data structure together. There is a well-known parallel algorithm ("tree reduction") for performing this operation in time proportional to the logarithm of the number of elements in the data structure [28]. Data parallelism is also present in the computation of parallel prefix operations.

3.1 Shared-Memory Bulk Synchronous Languages

We use High Performance FORTRAN (HPF) [27] as our example. HPF is somewhat unique among parallel languages in that it was designed by a group of no less than 50 researchers. It has two parallel loop constructs called the `FORALL` loop and the `INDEPENDENT` directive for expressing bulk synchronous parallelism. The body of the `FORALL` must consist of a sequence of assignments without conditionals or invocations of general procedures, although side-effect functions, declared to be `PURE` functions, can be invoked in a `FORALL`. These functions can contain conditionals. There is an implicit barrier at the end of every statement in a `FORALL`. The semantics of the `FORALL` is that all iterations of the first statement can be executed concurrently, and when these are completed, all iterations of the second statement can be executed concurrently, and so on.

The INDEPENDENT directive before a DO loop tells the compiler that the iterations of the loop can be done in parallel since they do not effect each other. There is an implicit barrier at the end of the loop but loop iterations do not have to be synchronized in any way. This directive is often used to expose opportunities for parallel execution to the compiler, as shown in the code below. The NEW clause asserts that J is local to the outer loop. Iterations of the outer loop can be executed concurrently if the values of IBLACK(I) are distinct from the values of IRED(J), and the IBLACK array does not have repeated values. This information cannot be deduced by a compiler, so the INDEPENDENT directive is useful for conveying this information.

```
!HPF$ INDEPENDENT, NEW(J)
  DO I = 1, N
    DO J = IBEGIN(I), IEND(J)
     X(IBLACK(I)) = X(IBLACK(I)) + X(IRED(J))
    END DO
  END DO
```

In HPF, the assignment of computational work to processors is not directly under the control of the programmer. Instead, it relies on a combination of *data distribution* directives and compiler technology to produce code with good locality, as described in [6, 37]. The two basic distributions are *block* and *cyclic* distributions. Block distributing an array gives each processor a set of contiguous elements of that array; if there are p processors and n array elements, each processor gets a contiguous block of n/p elements. In a cyclic distribution, successive array elements are mapped to successive processors in a round-robin manner; therefore, element i is mapped to processor $i \ mod \ p$. HPF also supports a *block-cyclic* distribution in which blocks of elements are dealt to processors in a round-robin manner. The compiler can exploit data distributions in assigning work by assigning an iteration to a processor if that processor has most of the data required by that iteration. Alternative strategies like the *owner-computes* rule [37] are also popular.

An HPF program for computing π is shown below. It approximates the definite integral $\int_0^1 4/(1+x^2)dx$ by using the rectangle rule, computing the value of $1/n * \sum_{i=1}^{n} 4/(1+((i-0.5)/n)^2)$. In this program, n is chosen to be 1000. SUM is an intrinsic for computing the sum of the elements of a distributed array.

```
      PURE REAL FUNCTION F(X)
      REAL, INTENT(IN) :: X
      F = 4.D0/(1.D0 + X*X)
      END FUNCTION F

      PROGRAM COMPUTE_PI
      REAL TEMP(1000)
!HPF$ DISTRIBUTE TEMP(BLOCK)
      WIDTH = 1.D0/1000
      FORALL (I = 1:1000)
      TEMP(I) = WIDTH * F((I - 0.5D0)*WIDTH)
```

```
END FORALL
T = SUM(TEMP)
END
```

A second version of HPF called HPF-2 with support for irregular computations and task parallelism has been defined. IBM, PGI, DEC and other companies have HPF compilers targeted to distributed-memory computers like the IBM SP-2. However, the quality of the compiler-generated code is relatively poor in comparison to hand-written parallel code, and source-level performance prediction has proved to be difficult since performance depends greatly on decisions about inter-processor communication made by the compiler [23]. For these reasons, interest in HPF is on the wane.

3.2 Distributed-Memory Bulk Synchronous Languages

The first theoretical study of bulk synchronous models was done by Valiant who proposed the Bulk Synchronous Parallel (BSP) model [43] as a bridging model between parallel hardware and software. A parallel machine in the BSP model has some number of processors with local memories, interconnected by a routing network. The computation consists of a sequence of supersteps; in each superstep, a processor receives data sent by other processors in the previous superstep, performs local computations, and sends data out to other processors which receive this data in the following superstep. A processor may send and receive any number of messages in each superstep. Consecutive supersteps are separated by barrier synchronization of all processors. Communication is therefore separated from synchronization.

Although BSP is a model and not a language, a number of libraries that implement this model on a variety of parallel platforms have been written [19, 31]. In this article, we describe the BSP Green library [19] which provides the following functions:

1. `void bspSendPkt(int pid,const bspPkt *pktPtr)`: Send a packet to the process whose address is `pid`; the data to be sent is at address `pktPtr`.
2. `bspPkt *bspGetPkt()`: Receive a packet sent in the previous superstep; returns NULL if all such packets have already been received.
3. `void bspSynch()`: Barrier synchronization of all processors.
4. `int bspGetPid()`: Return the process id.
5. `int bspGetNumProcs()`: Return the number of processes.
6. `bspGetNumPkts()`: Return the number of packets sent in the previous superstep to this process that have not yet been received.
7. `bspGetNumStep()`: Return the number of the current superstep.

The first three functions are called *fundamental* functions since they implement the core functionality of the BSP model, and the last three are called *supplemental* functions. This set of functions is somewhat limited, and a more user-friendly library would provide other supplemental functions such as one to perform reductions, while remaining true to the BSP spirit. For example, the BSPLib project (http://www.BSP-Worldwide.org/) includes support for one-sided communication and high-performance unbuffered communication.

The following program (from [19]) uses the BSP Green library functions to perform a trivial computation with three processors connected logically in a ring. Each processor sends the value of a local variable `A` to its neighbor in the ring, and then performs a local computation with the value it receives. This takes two two supersteps. Note that some of the code (such as the calls to `memcpy`) is at a fairly low level of abstraction. The philosophy behind the decision to expose such details to the programmer is that all expensive operations should be evident when reading the program text.

```
void program(void)
{ int pid, numProcs, A,B,C;
  bspPkt pkt, *pktPtr;

  pktPtr = &pkt;
  pid = bspGetPid();            //get process ID
  numProcs = bspGetNumProcs();  //get number of processes
  if (pid == 0) {A = 3; B = 12;} //initialize A and B
  if (pid == 1) {A = 1; B = 18;}
  if (pid == 2) {A = 5; B = 7;}

  memcpy((void *)pktPtr, (void *)&A, 4); //Store A into packet buffer
  bspSendPkt((pid+1)%numProcs, pktPtr);  //send data to neighbor in ring
  bspSynch();       //superstep synchronization

  pktPtr = bspGetPkt();   //receive packet
  memcpy((void *)&C, (void *)pktPtr, 4);  //store data in C
  C = C + B;
  fprintf(stdout, ‘‘Process %d, C = %d\n’’, pid, C);
  bspSynch();    // superstep synchronization
}
```

One of the goals in the design of BSP is to permit accurate performance prediction of parallel program. Performance prediction of BSP programs is made using a model with three parameters: (i) the number of processors p, (ii) the gap g which reflects the network bandwidth available to each processor, and (iii) the latency L which is time required to send a packet through the network and perform a barrier synchronization. If a BSP program consists of S supersteps, the execution time for superstep i is $w_i + g * h_i + L$ where w_i is the longest computation time required by any processor in that superstep and h_i is the largest number of packets sent or received by any processors in that superstep. This performance model assumes that communication and computation are not overlapped. The execution time for the program is W + gH + LS where $W = \Sigma w_i$ and $H = \Sigma h_i$.

A major contribution of BSP has been to highlight what can be accomplished with its minimalist approach to communication and synchronization. However, the exchange of a single message between just two processors requires the co-operation of all processors in the machine! The BSP counter-argument is that worrying about optimizing individual messages makes parallel programming too difficult, and that the focus should be on getting the large-scale structure of the parallel program right.

4 Fine-grain Synchronous Parallel Languages

The most relaxed form of synchronization is *fine-grain* synchronization in which two or more processors can synchronize whenever they need to without the involvement of other processors. This style of programming is usually called Multiple Instruction Multiple Data (MIMD) programming. Fine-grain synchronization is appropriate for exploiting *task parallelism* in which autonomous tasks need to synchronize either to obtain exclusive access to shared resources or because they are organized as a pipeline in which data structures are produced and consumed concurrently.

4.1 Shared-Memory MIMD Languages

We discuss FORTRAN/OpenMP [35] which is a new industry standard API for shared-memory parallel programming, and contrast it with the more "expression-oriented" approach of Multilisp [22].

4.1.1 OpenMP

OpemMP is a set of compiler directives and runtime library routines that can be used to extend FORTRAN and C to express shared-memory parallelism. It is an evolution of the now-moribund ANSI X3H5 effort. An OpenMP FORTRAN program for computing π is shown below. A single thread of control is created at the start, and this thread executes all statements till the `PARALLEL` directive is reached. The `PARALLEL` directive and its corresponding `END PARALLEL` directive delimit a *parallel section*. At the top of the parallel section, a certain number of slave threads are created which co-operate with the master to perform the work in the parallel section and then die at the bottom of the parallel section. In our example, the only computation in the parallel section is the `do` loop. Furthermore, the `DO` directive asserts that the iterations of the loop can be performed in parallel. Optional clauses in this directive permit the programmer to specify how iterations should be assigned to threads. For example, the `SCHEDULE(DYNAMIC,5)` clause specifies that iterations are assigned to threads in blocks of 5 iterations; when a thread completes its iterations, it returns to ask for more work, and so on. There is an implicit barrier synchronization at the bottom of the parallel `DO` loop, as well as at the end of the parallel region. The barrier synchronization may be avoided by specifying the clause `NOWAIT` at these points. Once the parallel region is done, all threads except the master die. The master completes the execution of the rest of the program.

By default, all variables in a parallel region are shared by all the threads. Declaring a variable to be `PRIVATE` gives each thread its own copy of that variable. By default, loop control variables like `i` in our example are `PRIVATE`. Note that all the threads in our program write to the `sum` variable. Declaring `sum` to be a `REDUCTION` variable permits the compiler to generate code for updating this variable atomically. The compiler may also generate more elaborate code such as performing the reduction in a tree of processors.

```
program compute_pi
integer n,i
```

```
  double precision w,x,sum,pi,f,a
  f(a) = 4.d0/(1.d0 + a*a)
  print *, 'Enter the number of intervals'
  read *,n
  w = 1.0d0/n
  sum = 0.0d0
!$OMP PARALLEL
!$OMP DO  SCHEDULE(DYNAMIC,5), PRIVATE(x), REDUCTION(+: SUM)
  do i = 1, n
    x = w * (i - 0.5d0)
    sum = sum + f(x)
  enddo
!$OMP END PARALLEL
  pi = w*sum
  print *,
  computed pi = ', pi
  stop
  end
```

Fine-grain synchronization in OpenMP is accomplished by the use of critical sections. The `CRITICAL` and `END CRITICAL` directives restrict access to the enclosed region to one thread at a time. For example, instead of declaring `SUM` to be a reduction variable as above, we can use a critical section to update it atomically as shown below.

```
!$OMP PARALLEL
!$OMP DO  SCHEDULE(DYNAMIC,5), PRIVATE(x,temp)
  do i = 1, n
    x = w * (i - 0.5d0)
    temp = f(x)
!$OMP CRITICAL
    sum = sum + temp
!$OMP END CRITICAL
  enddo
!$OMP END PARALLEL
```

OpenMP also a *parallel sections* directive. Each section contains computations that can be performed in parallel with the computations in the other sections of this construct.

OpenMP is being supported by SGI, KAI and other companies.

4.1.2 Multilisp

It is instructive to contrast OpenMP with Multilisp [22] which is also a shared-memory MIMD parallel language, but one in which synchronization between producers and consumers of data is folded quite elegantly into the data accesses themselves. Multilisp is a parallel extension of Scheme which was intended for writing parallel programs for the MIT Concert multiprocessor. There are two parallel constructs, one for evaluating the arguments to a function in parallel (`pcall`), and another for computing a value in parallel with executing code that will eventually use that value (`future`).

The expression `(pcall F A)` is equivalent to the Scheme procedure call `(F A)` except that the expressions `F` and `A` are evaluated in parallel. The function that expression `F` evaluates to is invoked after that evaluation of the argument `A` is complete. The `pcall` construct can be nested; for example, the expressions `F` and `A` may themselves contain `pcall` constructs.

The `future` construct can be used to fork off a computation that is performed in parallel with execution of code that may ultimately need the value of that computation. For example, the expression `(pcall cons A B)` evaluates `A` and `B` in parallel, and builds the cons cell when the evaluations are complete. The construction of the data structure need not wait for the termination of the computations of `A` and `B` since these computations can immediately return "place-holders" for the ultimate values, replacing these place-holders with the actual values when those become available. This can be accomplished by the invocation `(pcall cons (future A) (future B))`. While the computation of `A` and `B` is taking place, the `cons` cell can be used to build other data structures or be passed to other procedure invocations. An operation like addition that tries to use the value of `A` or `B` before that value is available is blocked till the corresponding place-holder is replaced with the value; when that value becomes available, that computation is allowed to continue. This is a form of fine-grain dataflow synchronization at the level of data structure elements.

The following program shows a Multilisp version of Quicksort(taken from [22]). The `partition` procedure uses the first element `elt` of list `l` to divide the rest of `l` into two lists, one containing only elements less than `elt` and the other containing elements greater than or equal to `elt`. These lists are themselves sorted in parallel recursively, and the resulting lists, together with `elt`, are appended together to form the output. To reduce the overhead of explicitly appending lists, `qs` takes an additional argument `rest` that is the list of elements that should appear after the elements of `l` in the sorted list.

```
(defun qsort (l) (qs l nil))

(defun qs (l rest)
   (if (null l) rest
       (let ((parts (partition (car l) (cdr l))))
         ; sort the two partitions in parallel recursively
         (qs (left-part parts)
             (future (cons (car l) (qs (right-part parts) rest)))))))

(defun partition (elt lst)
   (if (null lst)
       (bundle-parts nil nil)
       (let ((cdrparts (future partition elt (cdr lst))))
         (if (> elt (car lst))
             (bundle-parts (cons (car lst)
```

```
                                (future (left-part cdrparts)))
                          (future (right-part cdrparts)))
            (bundle-parts (future (left-part cdrparts))
                          (cons (car lst)
                                (future (right-part cdrparts)))))))

(defun bundle-parts (x y) (cons x y))

(defun left-part (p) (car p))

(defun right-part (p) (cdr p))
```

It can be seen that this Multilisp program is a functional program to which `future`'s have been added. The problem of deciding where it is safe and profitable to insert `future`'s in a general Multilisp program is a non-trivial one since Multilisp is an imperative language in which expression evaluation can have side-effects. The suggested programming style is to write mostly functional code and look for opportunities to evaluate data structure elements as well as function arguments in parallel.

As in Scheme, the linked list is the key data structure in Multilisp. The role of lists in parallel programming is somewhat controversial because unlike arrays, lists are sequential access data structures and this sequentiality can limit speed-up in some programs. For example, consider applying a function f in parallel to each data item in a list. The list must traversed sequentially to spawn the parallel tasks, so parallel speed-up will be limited especially if the time for each function evaluation is small. If an array is used instead, the time required for the entire computation may be as small as the maximum of the times required for the individual function evaluations. Although linked lists are not used very often in parallel programming, note the `future` construct and its associated dataflow synchronization can be used in the context of other data structures.

The Linda language [17] also folds synchronization into data accesses, although in the case of Linda, synchronization is done during associative access of a shared *tuple space*.

4.2 Object-oriented MIMD Languages

We describe HPC++ [13] and Java [18]. These are both shared-memory languages.

4.2.1 HPC++

HPC++ [13] is a C++ library and language extension framework. For exploiting loop level parallelism, HPC++ has compiler directives called *pragmas* which are similar to the OpenMP directives. For example, parallel loops are exposed to the compiler by the `HPC_INDEPENDENT` directive, used in the following code to compute the ComputePi function.

```
double ComputePi(int n) {
  double w = 1.0/n;
  double sum = 0.0;
```

```
  #pragma HPC_INDEPENDENT, PRIVATE x
  for (int i = 1; i < n; i++) {
      double x = w * (i - 0.5);
      #pragma HPC_REDUCE
      sum += f(x);}
  return sum;}

double f(double a) {
  return 4.0/(1.0 + a*a);
 }
```

One of the innovative aspects of HPC++ is its extension of the Standard Template Library (STL) to support data parallelism. The Standard Template Library (STL) in C++ provides (i) *containers* that define aggregate data structures like vector, lists and queues, (ii) *iterators* for enumerating over the contents of containers, and (iii) *algorithms* that allow element-wise operations to be applied to containers. HPC++ has a Parallel Standard Template Library (PSTL) that provides parallel versions of these.

The most important container class in PSTL is the `Array` container (STL does not have multi-dimensional arrays which are crucial for scientific programming). By default, array containers are block-distributed but the programmer can specify a custom distribution by providing a *distribution object* containing a function that maps array indices to processors. The *par_for_each* iterator in PSTL is the parallel analog of the *for_each* iterator in STL. HPC++ also has a number of parallel algorithms such as *par_apply* for applying a function to each element of a container, and *par_reduction* which is a parallel apply followed by a reduction with an associative binary operation. The following code shows HPC++ code for summing all the positive elements of a vector. The vector `v` is block distributed. The parameters to the `par_reduction` algorithm are the associative combining operation, the function to be applied to each element of the container and the starting and ending parallel iterators for the reduction.

```
BlockDistribution d(100, 100/numcontexts());
distributed_vector<double> v(100, &d);

class GreaterThanZero{
   public:
    double operator() (double x){
       if (x > 0) return x;
       else return 0;
    }
};
...
double total = par_reduction(plus<double>(), GreaterThanZero(),
                             v.parbegin(), v.parend());
...
```

HPC++ is under active development. Planned enhancements to the existing implementation include a library for distributed active objects and an interface to CORBA via the IDL mapping. Another approach to extending C++ for parallel computing is the Charm++ effort [26].

4.2.2 Java

Java is a new object-oriented programming language which has a library of classes that support programming with threads. The thread library is intended primarily for writing multi-threaded uniprocessor programs like GUI managers. A parallel Java program consists of a number of threads executing in a single global object namespace. These threads are instances of user-defined classes that are usually subtypes of the `Thread` class in the Java library which override the `run` method of the `Thread` class to define what threads must do once they are created. Threads are first-class objects that can be named, passed as parameters to methods, returned from methods etc. In addition, methods inherited from the `Thread` class permit a thread to be suspended, resumed, put to sleep for specified intervals of time, etc. Java also supports the notion of *thread groups*. Threads in a group can be suspended and resumed collectively.

Synchronization in Java is implemented using *monitors*. A monitor is associated with every object that contains a method declared to be *synchronized*. Whenever control enters a synchronized method in an object, the thread that invoked that method acquires the monitor for that object until the method returns. Other threads cannot call a synchronized method in that object until the monitor is released.

Java was not intended to be a language for parallel scientific computation. For example, it does not support multi-dimensional arrays nor are there any constructs for performing collective communication operations like reductions. However, there are efforts under way to use Java as a co-ordination language for multi-platform computational science applications [12].

4.3 Distributed-Memory MIMD Languages

One of the earliest distributed-memory MIMD languages is Communicating Sequential Processes (CSP) [9] which spurred a lot of work on the theory and practice of message-passing language constructs, such as Occam [29] and Brinch Hansen's SuperPascal [24]. More recent languages in this area have taken a message-passing library like PVM [1] or MPI [44] and grafted it onto a sequential language to obtain a distributed-memory parallel programming language. We will use FORTRAN/MPI to discuss this class of languages. In this programming model, a certain number of processes are assumed to exist, each having a unique name (usually a non-negative integer) and its own address space. Processes communicate by sending and receiving messages. A process can send data to another process by executing a SEND command, specifying the data to be transferred and the name of the recipient. The receiving process gets the data by executing a RECEIVE command, specifying the name of the sending process and the variable into which the data should be stored.

There are a number of variations on this basic SEND-RECEIVE theme. *Blocking* SEND-RECEIVE constructs require the two processes to rendezvous before the data transfer takes place, which allows data to be transferred from one process to another without buffering in the operating system. However, if one process gets to the rendezvous considerably in advance of the other one, it cannot do useful work till the other process catches up with it. This problem led to the development of *nonblocking* SEND and RECEIVE constructs. A non-blocking SEND permits the sending process to continue execution as soon as the data has been shipped out to the receiving process even if the receiving process has not executed a RECEIVE command; the non-blocking RECEIVE construct is like a probe that permits the receiving process to check for availability of data without getting stuck if the data has not yet been received.

In addition to these SEND/RECEIVE commands, MPI has a number of *collective communication calls* which are useful for doing reductions, broadcasts etc. collectively among *process groups*. These collective operations can be implemented using send's and receive's, but it is often possible to exploit the topology of the interconnection network to implement them more efficiently. MPI permits processes to belong to any number of process groups (called *communicators* in MPI terminology). All processes are members of the universal group `MPI_COMM_WORLD`.

The code shown below computes the value of π. The invocations of `MPI_COMM_SIZE` and `MPI_COMM_RANK` permit a process to determine the number of processes in the system and its own id. The broadcast of the value of `n` is performed by invoking `MPI_BCAST`. The parameters to this call are the (i) the starting address of the data to be broadcast, (ii) the number of values to be broadcast, (iii) the type of the data, (iv) the ID of process initiating the broadcast, (v) the process group to which the broadcast is performed, and (vi) an error flag. Global reductions may be performed with a similar invocation.

```
program compute_pi
include 'mpif.h'
double precision mypi,pi,w,sum,x,f,a
integer n, myid, numprocs,i,rc

f(a) =   4.d0/(1.d0 + a*a)

call MPI_INIT(ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

if (myid .eq. 0) then
   print *, 'Enter number of intervals'
   read *, n
endif
call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
w = 1.0d0/n
sum = 0.0d0
do i = myid+1,n,numprocs
   x = w*(i - 0.5d0)
   sum = sum + f(x)
enddo
```

```
mypi = w*sum

call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,MPI_COMM_WORLD,ierr)

if (myid .eq. 0) then
   print *, 'computed pi =', pi
endif
call MPI_FINALIZE(rc)
stop
end
```

5 Implicitly Parallel Programming Languages

Many of the parallel programming languages described above are in active use, but none of them is particularly *abstract* since they are all close to particular implementation models of parallel computing. It is likely that as better compiler and runtime systems technology becomes available, languages for programming parallel machines will become more abstract. This evolution would then parallel the evolution of sequential programming languages which started out being very close to the hardware on which programs ran, but have since evolved to higher levels of abstraction. For example, control flow transfer in early sequential languages like FORTRAN was done by GOTO statements which were manifestations of jump instructions in the underlying hardware, but GOTO statements have since been replaced by more abstract structured programming constructs. Similarly, variables in FORTRAN were names for fixed memory locations and existed for the duration of the program just like memory addresses in the machine model, but the data models of modern programming languages are built on abstract notions like type, scope and lifetime.

Although existing compiler and runtime systems technology is inadequate to permit efficient parallel programming in high-level abstract programming languages, many such languages have been proposed. In this section, we describe ZPL, an imperative array language which relies on parallelizing compiler technology to find opportunities for parallel execution, the functional languages Id and Haskell, and the logic programming languages Concurrent Prolog and PARLOG. An even more ambitious approach is taken by Unity [8] which attempts to derive parallel programs from high-level specifications written in a variation of temporal logic.

5.1 ZPL

In a FORTRAN or C program, statements that read and update disjoint memory locations can be executed concurrently. Therefore, it is possible in principle to use a sequential programming language like FORTRAN to program a parallel machine if one has a parallelizing compiler that can extract opportunities for parallel execution from sequential programs. The first such compiler was PARAFRASE [14] which took FORTRAN programs and attempted to find parallel DO loops through program analysis. However, automatic parallelization has proved to be difficult in general, although there has been noteworthy success in some problem domains like numerical linear algebra.

ZPL [20] is an imperative array language without explicitly parallel constructs which relies on compiler technology to identify opportunities for parallel execution. A novel feature on this language is its *region* construct which is an alternative to the *triplet* notation for describing constant-stride index sets that was described in Section 2. A disadvantage of the triplet notation is that it must be repeated for every subarray reference with this index set (as in `A[1:n] = B[1:n] + C[1:n]`.) ZPL permits a more compact expression of such statements by providing the *region* construct that permits the definition and naming of index sets. The declaration `region R = [1..n]` can be viewed as defining a template of virtual processors of the appropriate size. Regions can be used with both data declarations and blocks of statements, as shown in the code below. An integer `Intval` is allocated on each virtual processor of the region; similarly, the statements in the block are executed by each virtual processor. `Index1` is a keyword that permits each virtual processor to determine its index.

```
program Compute_pi;                -- Program to approx. pi
 config var n : integer = 100;     -- Changeable on Cmd Line
 region R = [1..n];                -- Problem space

procedure f( a : double) : double; -- Fcn for rectangle rule
    return 4 / (1 + a^2);

procedure Compute_pi();            -- Entry point
var  Intval : [R] double;          -- A vector of rect. pt.s
         pi :     double;          -- Scalar result
[R] begin
     Intval := (Index1 - 0.5) / n; -- Figure interval pts
     pi     := +<< f(Intval) / n;  -- Approximate, sum, div
     writeln("Computed pi = ", pi);-- Output to standard out
    end;
```

Regions in ZPL may also be defined by applying operations like shifts to previously defined regions. Although ZPL compilers have been written for a variety of parallel platforms, it remains to be seen if the performance of the compiled code is sufficient to persuade programmers to move away from writing explicitly parallel programs in a language like FORTRAN with MPI.

5.2 Functional Languages

One approach to addressing the difficulty of determining non-interference of statements in languages like FORTRAN or C is to use functional languages. These languages are based on the notion of mathematical functions that take values as inputs and produces values as outputs. When executing a functional language program, all functions whose inputs are available can be evaluated in parallel without fear of interference. This *data-driven* parallel execution model is the foundation of a number of functional languages like VAL [2], ID [34] and SISAL [15]. An alternative execution model called *lazy evaluation* evaluates a function only if its inputs are available and it has been determined that the result of the function is required to produce the output of the program. Lazy evaluation permits the programmer to define and use infinite data objects such as infinite arrays or infinite lists (as long as only a finite portion of these infinite objects is required to produce the output), a feature which has been recommended for promoting modularity. Miranda [25] and Haskell [16] are languages that are based on the lazy evaluation model.

Operations like I/O do not fit naturally into the functional model since they are effects and not functions. Haskell uses *monads* to integrate I/O into a purely functional setting. A monad provides the illusion of an object with updatable state on which all actions are sequenced in a well-defined manner, which is sufficient for performing I/O. Monads permit the introduction of a limited form of side-effects into functional languages in a controlled manner, but these side-effects are limited since monads cannot be used to define objects that can be updated concurrently.

Two problems have limited the impact of functional languages on the parallel programming community. The first is *aggregate update problem* which refers to the difficulty of manipulating data structures like large arrays efficiently. Data structures are treated as values in functional languages, so they cannot be updated in place. The effect of storing a value v into element i of array A must be obtained by defining a new array B that is identical to A except in the i^{th} position where it has the value v. A naive implementation that makes a copy of A will be very inefficient. A variety of compiler optimizations [7] and language constructs (like *I-structures* in Id [3]) have been proposed to address this problem but it is not clear to what extent these address the problem. A second problem is locality. In principle, an interpreter for a functional language can keep a work-list of expressions whose inputs are available, and evaluate these expressions in any order. Unless this is done carefully, it will have an adverse effect on locality, making it difficult to exploit caches and memory hierarchies. One solution is to remove caches from the implementation model and rely on *multi-threaded processors* like dataflow processors that are latency-tolerant. A complementary solution is to use compiler techniques to extract long sequential threads of computation from functional programs, and exploit locality in the execution of these threads. However, there appears to be little commercial interest in building multi-threaded processors at this time; furthermore, the problem of sequentializing functional programs does not appear to be any easier than the problem of parallelizing imperative language programs.

5.3 Logic Programming Languages

Although functional languages eliminate the notion of sequential control from the programming model, they still retain the notion of *directionality* in the sense that the inputs of a function are distinct from its output. Logic programming languages provide an even higher level of abstraction by eliminating directionality through the use of relations (predicates) instead of functions. A logic program consists of a set of clauses that describe relations either explicitly by enumerating the tuples in the relation or implicitly in terms of other relations. Clauses that describe a relation explicitly are called *facts* while those that describe relations implicitly are called *rules*. The first three facts in the program shown below specify that the `father` relation contains the tuples `<Adam,Abel>`, `<Adam, Cain>` and `<Abel,Bill>`. The `parent` relation is described by a rule: for all `X` and `Y`, the tuple `<X,Y>` is contained in the `parent` relation if it is contained in the `mother` relation (informally, `X` is the parent of `Y` if `X` is the mother of `Y`). The `grandfather` clause is defined implicitly as well: for all `X`, `Y` and `Z`, the tuple `<X,Y>` belongs to the `grandparent` relation if there exists a `Z` such that `<X,Z>` and `<Z,Y>` belong to the the `parent` relation.

```
father(Adam, Abel).
father(Adam, Cain).
father(Abel, Bill).
mother(Eve, Abel).
mother(Eve, Cain).

parent(X,Y) :- mother(X,Y).
parent(X,Y) :- father(X,Y).
grandparent(X,Y) :- parent(X,Z),parent(Z,Y).

:- grandparent(Adam,W).
```

In terms of formal logic, the symbol `:-` stands for logical implication, and the symbol ',' on the right hand side of clauses stands for conjunction. Variables like `X` and `Y` are universally quantified over the clause in which they appear. Each clause is therefore a *Horn clause* and the program is a conjunction of Horn clauses.

Given the relations, it is possible to make a variety of queries such as asking if a given tuple occurs in a relation. *Bottom-up query evaluation* starts from the facts and uses the rules repeatedly to compute the tuples in the relations of the program, terminating when enough information has been obtained to answer the query. This kind of data-driven evaluation obviously exposes a lot of parallelism but it can lead to an unbounded amount of useless computation in general. *Top-down* query evaluation generates sub-problems from the original query and solves them recursively to answer the query. The query `grandfather(Adam,W)` can be answered if we can find a `Z` and `W` such that `parent(Adam,Z)` *and* `parent(Z,W)`. The first sub-problem can be solved in two ways: either by solving `mother(Adam,Z)` *or* by solving `father(Adam,Z)`. These explorations can be described compactly by an *AND-OR tree*, shown in Figure 2.

Parallelism in top-down query evaluation comes in two flavors called *and-parallelism* and *or-parallelism*. In and-parallelism, conjunctive sub-goals such as `parent(Adam,Z)` and `parent(Z,W)` in our example are solved concurrently. The first sub-goal produces possible solutions for `Z`, the second sub-goal produces possible solutions for `Z` and `W` and the natural join of these solution sets produces the answers to the original query. Similarly in or-parallelism, disjunctive sub-goals are solved in parallel and the results are unioned together.

The idealized model of parallel logic programming described here is difficult to implement efficiently, so researchers have proposed adding constructs to give programmers some control

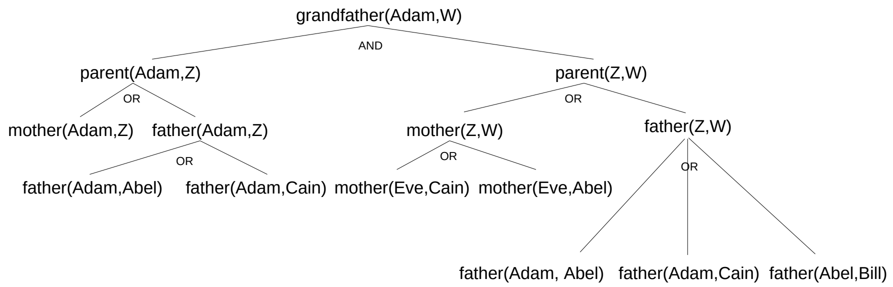

Figure 2: And-Or Tree

of parallel execution. To avoid having to compute the natural join of solutions from conjunctive sub-goals solved in parallel, *mode declarations* can be used to specify that some sub-goals will produce solutions that will be consumed by other subgoals. For example, Concurrent Prolog [38] has *read-only* annotations ('?') using which we can write the `grandfather` clause as follows:

```
grandparent(X,Y) :- parent(X,Z),parent(Z?,Y).
```

This requires the first sub-goal to produce `Z` and the second sub-goal to read it. Similarly, PARLOG [10] has *mode declarations* on variables in the left hand side of clauses. A limited form of or-parallelism called *committed choice or-parallelism* that uses Dijkstra's guards has been proposed in GHC [42] and PARLOG.

Logic programming ideas continue to be used in areas like artificial intelligence, but there is little mainstream interest at this point. The early enthusiasm for separating the logic of algorithms from their control did not last very long, and logic programming found themselves introducing extra-logical constructs like guards and modalities to improve program efficiency. In addition, a real programming language has to have arithmetic functions like addition and multiplication, but interpreted functions have always existed somewhat uneasily in the relational model.

6 Conclusion

Parallel programming today is done in languages that are very close to particular parallel implementation models. Thus, efficiency comes at the cost of portability. It is likely that parallel programming languages will become more abstract when the necessary compiler and runtime systems technology becomes available.

References

[1] A.Beguelin, G. Geist, W. Jiang, T. Manchek, K. Moore, and V. Sunderam. A user's guide to PVM: Parallel virtual machine. Technical Report TM-11826, Oak Ridge National Laboratories, 1991.

[2] W. Ackerman and J. Dennis. VAL - a value-oriented language. Technical Report LCS/TR-218, M.I.T., 1979.

[3] Arvind, R. Nikhil, and K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11, October 1989.

[4] W. Brainerd, C. Goldberg, and J. Adams. *Programmer's Guide to FORTRAN 90*. Springer, 1996.

[5] Burroughs Corp., Paoli, Penn. *Burroughs Scientific Processor Vector Fortran Specification*, 1978.

[6] D. Callahan and K. Kennedy. Compiling programs for distributed memory multiprocessors. *The Journal of Supercomputing*, 2(2), October 1988.

[7] D. Cann. *Compilation techniques for high performance applicative computation*. PhD thesis, Colorado State University, 1989.

[8] K. Chandy and J. Misra. *Parallel Program design: a foundation*. Addison-Wesley, New York, 1988.

[9] C.Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[10] K. Clark and S. Gregory. *Concurrent Prolog: collected papers*, volume 1, chapter PARLOG: parallel programming in logic. M.I.T. Press, 1987.

[11] CRAY Research Inc., Bllomington, Minn. *CRAY-1 Computer System Hardware Reference Manual*, 1978.

[12] K. Dincer and G. Fox. Using Java and JavaScript in the Virtual Programming Laboratory: A web-based parallel programming environment. Technical report, Syracuse University, 1997.

[13] E.Johnson and D. Gannon. HPC++: Experiments with the Parallel Standard Templates Library. Technical Report TR-96-51, Indiana University, 1996.

[14] D. Kuck et al. The effects of program restructuring, algorithm change and architectural choice on program performance. In *International Conference on Parallel Programming*, pages 129–138, 1984.

[15] J. McGraw et al. Sisal: Streams and iterations in a single-assignment language. Technical Report M-146, Lawrence Livermore Noational Laboratories, March 1985.

[16] J. Peterson et al. Haskell: a purely functional language. http://www.haskell.org, April 1997.

[17] D. Gelernter, N. Carriero, S. Chandran, and S. Chang. Parallel programming in Linda. In *Proceedings of the International Conference on Parallel Programming*, pages 255–263, August 1985.

[18] J. Gosling, W. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[19] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas. Towards efficiency and portability:programming with the BSP model. In *Proceedings of the 8th annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, June 1996.

[20] W. Griswold, G. Harrison, D. Notkin, and L. Snyder. Scalable abstractions for parallel programming. In *Proceedings of the 5th distributed memory computer conference*, pages 1008–1016, 1990.

[21] Mark Guzzi, David Padua, Jay Hoeflinger, and Duncan Lawrie. Cedar FORTRAN and other vector and parallel FORTRAN dialects. *Journal of Supercomputing*, 3:37–62, 1990.

[22] R. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, October 1985.

[23] P. Hansen. An Evaluation of High Performance FORTRAN. *ACM SIGPLAN Notices*, 33(3):57–64, March 1998.

[24] Per Brinch Hansen. *Studies in Computational Science.* Prentice-Hall Inc, 1995.

[25] I. Holyer. *Functional Programming with Miranda.* UCL Press, 1992.

[26] L. Kale and S. Krishnan. Charm++: a portable concurrent object-oriented system based on C++. In *Proceedings of the conference on object-oriented programming systems, languages and applications*, September 1993.

[27] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook.* MIT Press, 1994.

[28] F. Leighton. *Introduction to Parallel Algorithms and Architectures.* Morgan Kaufmann, 1992.

[29] Inmos Ltd. *Occam 2 Reference manual.* Prentice-Hall Inc, 1988.

[30] MathWorks Inc., Natick, MA 01760. *MATLAB Programmer's Manual*, 1996.

[31] R. Miller. A library for Bulk Synchronous Parallel programming. In *Proceedings for the BCS Parallel Processing Specialist Group workshop on general purpose parallel computing*, pages 100–108, December 1993.

[32] R. Millstein and C. Muntz. The Illiac IV FORTRAN compiler. *ACM Sigplan Notices*, 10(3):1–8, March 1975.

[33] R. Millstein and C. Muntz. The Illiac IV Fortran compiler. *ACM Sigplan Notices*, 10(3), 1975.

[34] R. Nikhil, K. Pingali, and Arvind. Id Nouveau. Technical Report CSG Memo 265, M.I.T. Laboratory for Computer Science, 1986.

[35] OpenMP Organization. OpenMP: A proposed industry standard API for shared memory programming. available from http://www.openmp.org.

[36] G. Paul and M. Wilson. An introduction to VECTRAN and its use in scientific computing. In *Proceedings of the 1978 LASL Workshop on Vector and Parallel Processing*, pages 176–204, 1978.

[37] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *Proceedings of the ACM Symposium on Programming Language Design and Implementation*, 1989.

[38] E. Shapiro. *Concurrent Proog: collected papers*, volume 1, chapter A subset of Concurrent Prolog and its interpreter. M.I.T. Press, 1987.

[39] David Skillicorn and Domenico Talia. *Programming Languages for Parallel Processing.* IEEE Press, 1994.

[40] Thinking Machines Corp., Cambridge, MA 02139. *Paris Reference Manual*, 1991.

[41] Thinking Machines Corporation. *Connection Machine CM-200 Technical Summary*, June 1991.

[42] K. Ueda. *Concurrent Prolog: collected papers*, volume 1, chapter Guarded Horn Clauses. M.I.T. Press, 1987.

[43] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[44] W.Gropp, E.Lusk, and A. Skjellum. *Using MPI.* M.I.T. Press, 1994.

[45] R.G. Zwakenberg. Vector extensions to LRLTRAN. *ACM Sigplan Notices*, 10(3):77–86, March 1975.