Introducing the Parallel Random Access Machine together with FORTRAN 90/95

Hans J. Schneider
Universität Erlangen-Nürnberg – Institut für Informatik
(Lehrstuhl 2)
Martensstraße 3, D-91058 Erlangen
schneider@informatik.uni-erlangen.de

Abstract

Parallelism is an interesting theme in many areas of computer science and its applications. Nevertheless, students try very hard to design parallel solutions to problems of which they know efficient sequential solutions. Studying the theoretical foundations may facilitate understanding parallelism. Nevertheless, it seems necessary to make run the algorithms in order to bridge the gap between theory and practice. By way of example, we show how to make students familiar with SIMD parallelism both by introducing the theory and by implementing the algorithms in a suitable programming language.

1 Introduction

In a previous paper [13], we have shown how to embed the theory of computable functions into an introductory course on programming. The students were simultaneously made familiar with programming in Scheme and the theory of computable functions. Now, we consider the converse situation: We had to give a course on the fundamentals of parallel algorithms, and we have made the theoretical algorithms run by implementing them in a suitable programming language. A major problem has been that students have learned to think sequentially; they use an assembler, Basic, Fortran, Pascal, C, or some other sequential programming language including the object-oriented languages such as Java. Parallelism is usually considered only on the level of tasks, e.g., in courses on operating systems. Another reason why students are trying very hard is that sequential algorithms often introduce data dependencies to reduce the number of operations to be performed, thus preventing the operations to be executed in parallel. Therefore, the

main focus of our course has not been how to parallelize well-known algorithms, but to show the maximal degree of parallelism a given problem allows. (Only in one chapter, we have mentioned the restrictions imposed by communication structures and data distribution.) The course has been based on the concept of the parallel random access machine (*PRAM*). It has closely followed the presentation of parallel algorithms for shared-memory machines given by Karp and Ramachandran [10].
Of course, the most important fields to apply parallel algorithms are science and engineering. Realistic numerical problems, however, need too much time to provide the students with the information necessary to understand the algorithms. Therefore, we mainly present the standard techniques using problems which every computer science student is familiar with, and we illustrate how to apply them to a wide range of problems, e.g., with the aid of matrix multiplication, we show that the fundamental techniques are also part of numerical algorithms.
FORTRAN is not our preferred programming language, but it is the most used language in scientific and engineering applications, and it has evolved considerably from its origins to the most recent versions. In our course, we have used a subset of FORTRAN 90/95 which is called F [4, 8, 9]. This dialect supports modular programming as well as structured control-flow, including recursive procedures. *PRAM* parallelism can be expressed by manipulating whole arrays and array sections. F provides the programmer only with the modern constructs; we do not use the historically ballast of FORTRAN IV.
The paper is organized as follows. First, we review the definition of the parallel random access machine, and we characterize the constructs of FORTRAN 95 that we need. Section 4 gives an example how to divide a problem into smaller ones if the logical and the physical structure of the data agree, whereas Section 6 considers the case that they do not. An application of Brent's principle to improve an algorithm and the bitonic sorting algorithm complete the presentation.

2 Parallel Random Access Machine (PRAM)

The aim of the course is to study the logical structure of parallel algorithms without referring to special hardware. Therefore, we need a virtual machine model that is as simple as possible and makes explicit the maximal degree of parallelism. The parallel random access machine has proved extremely useful to meet these requirements despite its disadvantages.
A *Random Access Machine (RAM)* is an idealized Von Neumann style computer [6]: It consists of a finite control, called the program, one or more accumulator registers, an instruction counter, and an infinite collection of memory cells. The instruction set contains instructions influencing flow of control (unconditional

jump, conditional jump), instructions transferring data between accumulator(s) and memory (load from memory, store into memory), and instructions performing arithmetic and logical operations. The *Parallel Random Access Machine (PRAM)* is the parallel analogue of the *RAM* [7, 12]: It consists of several independent sequential processors, each with its private memory, communicating with one another through a global memory. In one unit of time, each processsor can read one global or local memory location, execute a single *RAM* operation, or write into one global or local memory location; all processors perform the same operation, but some of them may skip the operation (do nothing).
The *PRAM* can not be considered a physically realizable model. As the number of processors or the size of the global memory scales up, it becomes impossible to provide a constant-length data path from any processor to any memory cell. Furthermore, it assumes that some processors may access the same memory cell at the same time. This leads to a hierarchy of *PRAM* models defined by the restrictions in accessing the global memory:

(1) An algorithm running on an *EREW PRAM* must ensure that different processors do not simultaneously access any memory location (exclusive-read/ exclusive-write).

(2) An algorithm running on a *CREW PRAM* must ensure that different processors do not simultaneously write into any memory location, but simultaneous read operations are allowed (concurrent-read/exclusive-write).

(3) An algorithm running on a *CRCW PRAM* may use both simultaneous read and simultaneous write operations (concurrent-read/concurrent-write).
Write conflicts can be resolved in different ways:

(3a) An algorithm running on a *COMMON PRAM* has to ensure that all processors writing into the same location write the same value.

(3b) An algorithm running on an *ARBITRARY PRAM* should work correctly regardless of which of the processors, participating in a common write operation, succeeds.

(3c) On a *PRIORITY PRAM*, the processors are linearly ordered; in a concurrent-write operation, the minimum numbered processor succeeds.

This classification introduces the *PRAM* models in increasing order of their power, i.e., each algorithm running correctly on one of the models also runs correctly on all models subsequently defined. On the other hand, any algorithm running on a *PRIORITY PRAMă* with $p(n)$ processors in time $t(n)$ can be simulated on the *EREW PRAM* in time $t(n)s(p(n))$ without increasing the number of processors, where $s(p)$ is the number of steps needed to sort p items [14].

3 The Programming Language

FORTRAN was the first problem-oriented programming language developed in the 1950s by an IBM team led by John Backus [1]. It has evolved considerably over the years. The first standard was published in 1966 (FORTRAN 66) and revised in 1978 (FORTRAN 77). During the following years, a lot of language extensions have been proposed, implemented, and tested. In 1991, a new standard (FORTRAN 90) has changed the appearance and the expressive power of the language substantially. The new features include better control structures, user-defined data types and pointers, recursion, modules, etc. FORTRAN 95 has added some minor features that we do not need in this paper. In our course, we have used the language F[1] [9, 17], that is a subset of both FORTRAN 90 and FORTRAN 95 [4, 8]. We have preferred this subset because it is restricted to the features supporting modern programming concepts.

The array processing operations of F are the main feature to implement data parallelism of the *PRAM*. If we have declared a vector `x(1:n)` of *n* components, we can access all the components of this vector simultaneously by writing `x(1:n)` (or simply `x`) on the left-hand or on the right-hand side of a statement. But we can also access special sections of the vector, e.g., the first half by `x(1:n/2)` or the even numbered components by `x(2:n:2)`, where the third parameter determines that every other element is selected. Let us consider a statement reducing the vector `x` to a vector `y` of half size by combining the components two by two:

```
y(1:n/2) = x(1:n-1:2) + x(2:n:2).
```

Another example is computing the odd numbered components of a vector `z` by adding the component of `x` with the same index to the preceding component of `z`:

```
z(1:n:2) = z(0:n-1:2) + x(1:n:2).
```

In this example, `z` is assumed to start at `0`ů. Now, we consider a vector simulating a linked list: The vector `val` associates a value with each index, and the vector `suc` defines the index where the next element can be found. At each position, the value should be changed by adding the value associated with the successor index:

```
val(1:n) = val(1:n) + val(suc(1:n)).
```

In this statement, `suc(1:n)` is used as a vector of indices, i.e., the second operand is a vector consisting of the components of `val`, but rearranged according to the values of `suc`. The statement may be abbreviated by `val = val + val(suc)`. The next example replaces the link to the successor component by a link to the component after the next:

[1] F is a trademark of The Fortran Company.

```
suc(1:n) = suc(suc(1:n)).
```

The examples with succ may run into some difficulty: We have assumed that each component defines a successor. This holds true if the links define a ring. If the list, however has a last element, i.e., a component without a successor, we must avoid an undefined operation. A possible solution is to use a special index nil indicating that there is no successor. Then, we can exclude this component from being processed by masking the statements:

```
where ( suc(1:n) /= nil )
   val(1:n) = val(1:n) + val(suc(1:n))
   suc(1:n) = suc(suc(1:n))
endwhere.
```

F requires that all declarations are given explicitly, the implicit declarations of traditional FORTRAN are not allowed. Furthermore, the intented use (in, out, or inout) of formal parameters must be specified in functions and subroutines. Overall, we can say that F implements the *CREW PRAM*.

4 Collecting and Broadcasting

Our first example is the *prefix sum algorithm* [11]. Given an array $x_1, x_2, \ldots, x_n$ of n elements from a domain D on which an associative operation + is defined. The problem is to compute $s_i = x_1 + x_2 + \ldots + x_i = \sum_{j=1}^{i} x_j$ for $i = 1, 2, \ldots, n$. The idea of the parallel algorithm is to reduce the problem size by adding each element at an odd position to its right-hand neighbor and then applying this procedure recursively. The recursion yields the final results at the even positions. In the last step, we compute the results at the odd positions. (The result at position 1 has already been set in the first line of the algorithm.)

```
recursive function prefix_sum (n, x) result (s)

   integer, intent(in) :: n
   integer, dimension (:), intent(in) :: x
   integer, dimension (1:n) :: s
   integer, dimension (1:n/2) :: y

   s(1) = x(1)
   if (n > 1) then
      y(1:n/2) = x(1:n-1:2) + x(2:n:2)
      s(2:n:2) = prefix_sum (n/2, y)
```

```
      s(3:n:2) = s(2:n-1:2) + x(3:n:2)
   end if

end function prefix_sum
```

As an immediate consequence of dividing the problem size by 2 in each step, the algorithm is efficient in the sense that it is an $O(\log(n))$-algorithm. But it is not optimal in the sense discussed below.
A closer look at the operations to be simultaneously performed shows that this is an *EREW*-program. The same holds true for the next program that copies the information stored in location `x(0)` into all the components of `x` such that processors can subsequently access them by exclusive-read operations:

```
subroutine broadcast (n, x)

   integer, intent(in) :: n
   integer, dimension (0:), intent(inout) :: x

   integer :: d, l, i
   integer, dimension (0:n-1) :: id

   id = (/ (i, i=0,n-1) /)
   d  = 1

   do l = 1, ceil_log(n)
      where ( (id + d ) < n )
         x(id+d) = x(id)
      endwhere
      d = 2*d
   enddo

end subroutine broadcast
```

The idea of the algorithm is that in each step, we double the number of processors that have received the correct value. In the first step, each processor sends its value to the "right-hand" neighbor, i.e., to the processor with the address increased by 1. After this step, the correct value is available at the processors 0 and 1, all other processors have copied irrelevant information. In the next step, the distance is doubled such that processors 2 and 3 receive the correct values from 0 and 1, respectively. The `where`-clause ensures that nothing is copied beyond processor `n-1`. (The vector `id` associates its index with each position.) Again, we

get an $O(\log(n))$-algorithm[2] since d runs through the powers of 2. From an architectural point of view, it is worth noting that this algorithm can efficiently run on a hypercube.

5 Brents's Principle

Given a problem of size n, a parallel algorithm is called *optimal* if the number of steps $t(n)$ and the number of processors $p(n)$ it needs to solve the problem satisfy:

$$\begin{aligned} t(n) &= \text{polylog}(n) \\ p(n) \cdot t(n) &= O(T(n)) \end{aligned}$$

where $T(n)$ is the time complexity of best sequential algorithm. It is called *efficient* if it satisfies

$$\begin{aligned} t(n) &= \text{polylog}(n) \\ p(n) \cdot t(n) &= O(T(n) \cdot \text{polylog}(n)) \end{aligned}$$

In this sense, the parallel prefix-sum algorithm is efficient, but not optimal. It is an instructive example how to apply *Brent's principle*: If a computation can be performed in time t with q operations and sufficiently many processors, then it can be performed in time $t + \frac{q-t}{p}$ with p processors [3]. The parallel prefix-sum algorithm is not optimal since we get $p(n)t(n) = O(n \log(n))$. Brent's principle suggests that we get an optimal algorithm if we can reduce the number of processors to $n/\log(n)$. This principle works fine if the processor allocation is not too difficult. In case of the prefix-sum problem, the trick is to combine the non-optimal parallel algorithm with the sequential one: We divide the given sequence of length n into subsequences of length $q = \log(n)$ each of which is associated with one processor. Each processor sums up the elements of its subsequence sequentially, then the non-optimal algorithm is applied to the sequence of subtotals using the reduced number of processors, and finally each processor sequentially computes the prefix sums in its subrange:

```
function opt_prefix_sum (n, x) result (s)

   integer, intent(in) :: n
   integer, dimension (:), intent(in) :: x
   integer, dimension (1:n) :: s

   integer :: p     ! No. of processors
```

[2] Function `ceil_log(n)` computes $\lceil \log_2 n \rceil$.

```
   integer :: q     ! No. of operands per processor
   integer, allocatable, dimension (:) :: y, z, zs, f
   integer :: i, j

   p = ceiling(n/dual_log(n))
   q = ceiling(real(n)/real(p))
   allocate (y(1:p), z(1:p), zs(1:p), f(1:p))
   f = (/ ((i-1)*q, i=1,p) /)
       ! first component of processor i

   ! Phase 1: Each processor sums up a subsequence
   y = 0
   do j = 1,q
      where ( f+j <= n )
         y = y + x(j:n:q)
      endwhere
   enddo

   ! Phase 2: Prefix sums of subtotals
   z = prefix_sum(p, y)

   ! Phase 3: Each vector computes prefix sums
   ! in its subrange
   z = (/ 0, z(1:p-1) /)
   do j = 1,q
      where ( f+j <= n )
         z = z + x(j:n:q)
         s(j:n:q) = z
      endwhere
   enddo

   deallocate(y, z, zs)

end function opt_prefix_sum
```

Since the number of processors depends on the parameter *n*, we must use dynamic storage allocation in the F version. The `where`-clauses ensure that the last processor does not access components with an index larger than *n*.

6 Linked Lists

The algorithms presented in the previous sections take advantage of the fact that the logical and the physical arrangement agree. If we have a linked list, the logical neighbor of an element, i.e., the successor or the predecessor, may be stored anywhere in the memory, and we have a link pointing to it. In this case, it is impossible to decide immediately on whether an element is at an even position of the list or not. We assume a linked list to be represented by two arrays. `val(i)` gives the value of the element at location `i` and `suc(i)` defines the location of the logical successor. Furthermore, we assume existence of a special address `nil` indicating that an element has no successor.[3] If we need a doubly linked list, we can construct the pointers to the predecessors in constant time:

```
where ( suc /= nil )
   pred(suc) = id
endwhere
pred(head) = nil
```

A fundamental technique to reduce the size of linked lists is *pointer jumping*. We consider the *list ranking problem* as an example. The problem is to compute the suffix sums, i.e., `rank(i)` is the sum of the values at position `i` and at all the positions that follow element `i` in the list. Wyllie's algorithm [15, 16, quoted from [10]] can be implemented as follows:

```
function list_ranking (n,val,suc) result (rank)

   integer, intent(in) :: n
   integer, dimension (:), intent(in) :: val, suc
   integer, dimension (1:n) :: rank
   integer, dimension (1:n) :: v, s
   integer :: k

   v = val  ! copy the given vectors
   s = suc
   do k = 1, ceil_log(n)
      where (s /= nil )
         v(1:n) = v(1:n) + v(s(1:n))
         s(1:n) = s(s(1:n))
      endwhere
```

[3] Without loss of generality, we may choose -1. In [10], the last element points to itself. This leads to a *CREW*-algorithm.

```
   enddo
   rank = v

end function list_ranking
```

In the first step of the loop, the value at the successor position is added to the original value, and the pointer `s` is replaced by the pointer to the element after the next. This distance is doubled in the next step, etc. Contrary to the prefix sum example, the operations are performed at each position and not only at every second.
But, how to divide a linked list into subsequences without running through the whole list? The students are very surprised when learning that it is possible to construct a $\lceil \log n \rceil$-ruling set in constant time [5]. For convenience, we link the last element to the first to avoid distinguishing special cases at the beginning and at the end of the list. Given a list L of n elements, we construct a sublist S such that no two adjacent elements of L are in S, and for each $e \in L$, there is a path of length $\leq \lceil \log n \rceil$ from e to an $e' \in S$. The following F program marks the elements belonging to S:

```
function ruling_set(n, suc) result (marked)

   integer, intent(in) :: n
   integer, dimension (0:), intent(in) :: suc
   logical, dimension (0:n-1) :: marked
   integer, dimension (0:n-1) :: id, pred, q, l
   logical, dimension (0:n-1) :: b, loc_min, loc_max, avail
   integer :: i

   id = (/ (i, i=0,n-1) /)
   pred(suc) = id
   marked = .false.

   q = diff_pos(id, id(suc))
   b = diff_bit(id, id(suc))
   loc_min = (q <= q(pred) .and. q <= q(suc))
   loc_max = (q >= q(pred) .and. q >= q(suc))

   where (loc_min)
      marked = (.not. loc_min(pred) &
                .and. .not. loc_min(suc)) .or. b
   endwhere
```

```
   avail = .not. marked .and.          &
           .not. marked(pred) .and.    &
           .not. marked(suc) .and. loc_max
   where (avail)
      marked = marked                  &
               .or. (.not. avail(pred) &
                     .and. .not. avail(suc)) .or. b
   endwhere

end function ruling_set
```

The vector `id` identifies each processor. The function `diff_pos(x,y)` determines the least-significant bit position where `x` differs from `y`, i.e., the identification of a processor differs from the identification of its (logical) successor. The position is given as a power of two[4]:

```
h = max(x,y) - min(x,y)
l = h - 1
r = ieor(h,l)
q = (r+1)/2
```

The function `diff_bit(x,y)` computes the value of the bit at this position:

```
b = (iand(x, diff_pos(x, y)) > 0 )
```

We mark the elements that are local minima if both neighbors are not.[5] If one of them is also a local minimum, we take the element which has value 1 at the distinguished bit position. (In the program, we use `.true.` and `.false.` instead of 1 and 0.) Then, we apply the analogous procedure to the local maxima, if both neighbosr are not yet marked. The distance of two marked elements can not exceed the number of bits we need to represent the number n.

7 Bitonic Sort

Although Batcher's sorting algorithm [2] has been designed for comparator networks, it fits well into the other material. Contrary to the sequential sorting algorithms the students already know, this algorithm is oblivious, i.e., the pairs to be compared do not depend on the outcome of previous comparisons.

[4] Contrary to the discussion in [10], it is not necessary to compute the number of the position itself, since we use these positions only in comparisons.

[5] The &-sign indicates that the end of the line is not the end of the statement.

The algorithm uses a property of special sequences $X = (x_0, x_1, \ldots, x_{2m-1})$ of even length. X is called *unimodal* if it has at most one local extremum[6], and it is called *bitonic* if it is a cyclic shift of a unimodal sequence. Trivially, shifting a bitonic sequence yields a bitonic sequence again, and concatenating an increasing sequence with a decreasing one (or vice versa) gives a unimodal sequence, and therefore a bitonic sequence. Batcher defines two operations on sequences that can easily be performed in parallel:

$$\begin{aligned} L(X) &:= (\min(x_0, x_m), \min(x_1, x_{m+1}), \ldots, \min(x_{m-1}, x_{2m-1})) \\ U(X) &:= (\max(x_0, x_m), \max(x_1, x_{m+1}), \ldots, \max(x_{m-1}, x_{2m-1})) \end{aligned}$$

He has shown that if X is bitonic, then $L(X)$ and $U(X)$ are bitonic, too, and $\max(L(X)) < \min(U(X))$. Furthermore, if $L(X)$ and $U(X)$ are sorted in nondecreasing order, then their concatenation is sorted in nondecreasing order. This results in a simple recursive algorithm to sort bitonic sequences: Compute $L(X)$ and $U(X)$ and store them in the lower and upper part of X, respectively. Then apply this procedure recursively to both parts.

The algorithm to sort non-bitonic sequences starts with considering the unsorted sequence as a list of $n/2$ sequences of length 2. Trivially, sequences of length 2 are bitonic, and in the first step ($l = 1$), we sort these subsequences alternately into nondecreasing and nonincreasing order by applying L and U with $m = 1$. Concatenating two of these subsequences, we get bitonic sequences of length 4, which are recursively sorted in the next step ($l = 2$). Again, we alternate nondecreasing with nonincreasing sequences, and we get bitonic sequences of length 8, etc. Figure 1 describes this process. If an arrow points from i to j, then x_i is compared to x_j. If we have $x_i > x_j$, the values are exchanged. (For simplicity, we restrict the implementation to sequences the length n of which is a power of 2.)

>From the educational point of view, it is worth mentioning that the theory of this algorithm is much easier than implementing it. The proof of Batcher's lemma is straightforward even if it is lengthy. Writing the program, however, requires some tricks to implement the "groups of arrows" and their directions. In the F version, we have implemented the compare-exchange operations by the min- and max-operations in the body of the inner loop. The directions are controlled by a Boolean vector b checking the $(l-1)$-th bit of the index. The vector i defines the $n/2$ positions at which the comparisons are to be performed:

```
subroutine batcher_sort(n,x)

   integer, intent(in) :: n
```

[6]Since we define a local minimum by $x_{i-1} > x_i \wedge x_i < x_{i+1}$ (and a maximum analogously), neither x_0 nor x_{2m-1} is considered a local extremum.

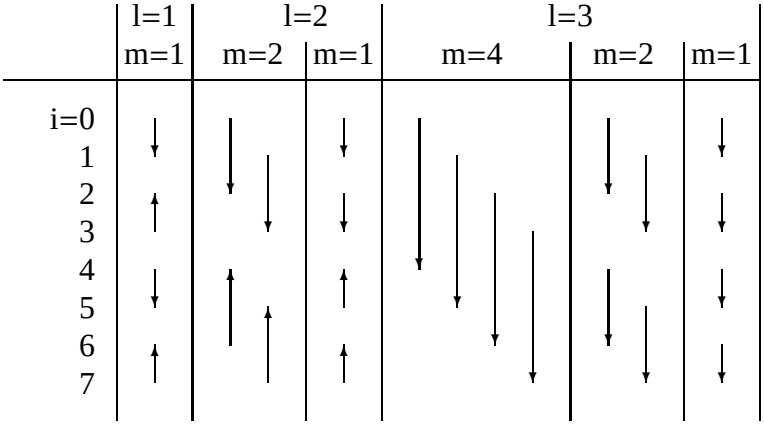

Figure 1: Bitonic sort with $n = 8$ elements

```
integer, dimension (0:), intent(inout) :: x

integer :: m, h, j, k, l, p
logical, dimension (0:n/2-1) :: b
integer, dimension (0:n/2-1) :: i
integer, dimension (0:n-1) :: aux

h = 1
do l = 1, ceil_log(n)
   m = h
   h = 2*h
   b = (/ (.not. btest(j,l-1), j = 0,n/2-1) /)
   do k = l, 1, -1
      i = (/ ((j+p, j=0,m-1), p=0,n-1,2*m) /)
      where (b)
         aux(i) = min(x(i),x(i+m))
         aux(i+m) = max(x(i),x(i+m))
      elsewhere
         aux(i) = max(x(i),x(i+m))
         aux(i+m) = min(x(i),x(i+m))
      endwhere
      x = aux
      m = m/2
   enddo
   h = 2*h
enddo
```

```
end subroutine batcher_sort
```

8 Conclusion

Of course, the concept to explain topics of theoretical computer science in a programming-oriented way is restricted to subjects with constructive proofs. In the course on parallel algorithms [18], we have started with the basic techniques some of them are presented in this paper. Then, we have considered various algorithms related to sorting, searching, and merging. Finally, we have applied these techniques to problems on trees and graphs, e.g., computing spanning trees and Euler paths. In a special chapter, we have considered the implementation on mesh-connected arrays and on cube-connected arrays (hypercubes).
The abstract model of the *PRAM* is well-suited to introduce the students into thinking in parallel on an abstract level, because this model is simpler than real parallel machines. Especially, the constant time algorithms, such as determining the predecessors, the ruling-set algorithm, or finding the maximum of n numbers, have convinced the students that studying parallel algorithms does not only mean subdividing a given algorithm into parts that can be performed in parallel, but that designing efficient parallel solutions means starting from scratch.
As we have already mentioned, FORTRAN is not our preferred programming language, and we were skeptical about using a FORTRAN dialect in a computer science course. The examples, however, show that the subset we have chosen is a good tool to implement SIMD parallelism. Nevertheless, we think that other languages should be used to implement parallelism on a higher level.
Overall we can state that explaining theoretical concepts in the usual way is much easier than implementing them. If, however, you want to understand theory better, you should implement the concepts. Conversely, thinking about the fundamentals of algorithms leads to better design.

References

[1] J.W. Backus: *The history of Fortran I, II, and III*, ACM SIGPLAN Notices 13, 8 (1978), pp. 165-180

[2] K.E. Batcher: *Sorting networks and their application*, Proc. AFIPS Spring Joint Comp. Conf. vol. 32 (1968), pp. 307-314

[3] R.P. Brent: *The parallel evaluation of general arithmetic expressions*, J. Assoc. Comput. Mach. 21, 2 (1974), pp. 201-206

[4] I. Chivers/J. Sleightholme: *Introducing FORTRAN 95*, Springer, London, 2000

[5] R. Cole/U. Vishkin: *Deterministic coin tossing with applications to optimal parallel list ranking*, Inform. and Control 70, (1986), pp. 32-53

[6] S.A. Cook/R.A. Reckhow: *Time bounded random access machines*, J. Comput. System Sci. 7 (1973), pp. 354-375

[7] S. Fortune/J. Wyllie: *Parallelism in random access machines*, Proc. 10th Ann. ACM Symp. on Theory of Computing (1978), pp. 114-118

[8] W. Gehrke: *Fortran 95 – Language Guide*, Springer, Londdon, 1996

[9] W. Gehrke: *Die Programmiersprache F*, Springer, Berlin, 1998

[10] R.M. Karp/V. Ramachandran: *Parallel algorithms for shared-memory machines*, Handbook of Theoretical Computer Science (Ed.: J. van Leeuwen), vol. A pp. 869-941, Elsevier, Amsterdam, 1990

[11] R.E. Ladner/M.J. Fischer: *Parallel prefix computation*, J. Assoc. Comput. Mach. 27 (1980), pp. 831-838

[12] W.J. Savitch/M.J. Stimson: *Time bounded random access machines with parallel processing*, J. Assoc. Comput. Mach. 26 (1979), pp. 103-118

[13] H.J. Schneider: *Computability in an introductory course on programming*, Bulletin of the European Association for Theoretical Computer Science 73 (2001), pp. 153-164

[14] U. Vishkin: *Implementation of simultaneous memory address access in models that forbid it*, J. Algorithms 4 (1983), pp. 45-50

[15] J.C. Wyllie: *The complexity of parallel computation*, TR 79-387, Dept. Comp. Sc., Cornell Univ., Ithaca, N.Y., 1979

[16] J.C. Wyllie: *The complexity of parallel computations*, Ph.D. Dissertation, Dept. Comp. Sc., Cornell Univ., Ithaca, N.Y., 1981

[17] *Welcome to the F Programming Language Homepage*, `http://www.swcp.com/~walt/`, Link: Information about F (Checked on 2003/08/07)

[18] *Foundations of Parallel Algorithms*, `http://www2.informatik.uni-erlangen.de/Lehre/WS02_03/ParAlg/` (Checked on 2004/01/05)