Synthesis of Massively Pipelined Algorithms for List Manipulation

Ali E. Abdallah

The University of Reading
Department of Computer Science
Reading, RG6 6AY, U.K.
Email: A.Abdallah@reading.ac.uk

Abstract. This paper presents new, efficient, massively pipelined algorithms for several list manipulation operations. Transformational programming is used in the development of these algorithms from clear functional specifications to networks of linearly connected communicating processes in CSP. The derivation of each algorithm is achieved by transforming the specification into an instance of a generic parallel functional form called *pipe pattern* and then refining this into CSP. The approach is demonstrated by transforming quadratic functional algorithms for sorting, removing duplicates, and calculating the difference between two lists, into pipelined versions running in linear time with a linear number of processors. The refinement from functions to CSP processes is based on a formal treatment given in earlier work by the author. Derivation and reasoning are conducted using Bird-Meertens Formalism.

1 Introduction

List manipulation operations such as those for sorting, removing duplicates, and calculating the difference between two lists are essential in the construction of numerous applications. The purpose of this paper is to demonstrate how transformational programming can be used to synthesize new massively parallel algorithms for these operations. The starting specification of each operation is formulated as a functional program and the final implementation is formulated in CSP as a static network of processes. The implementation is derived from the specification by a process of systematic algebraic manipulations.

The effectiveness of the calculational approach in deriving efficient sequential algorithms from high-level functional specifications has been demonstrated in numerous case studies [3, 4, 5, 6]. Because of the much greater complexity involved in the development of parallel algorithms, the transformational programming approach seems to be essential, not only for a deeper understanding of existing algorithms, but also, as this paper shows, for the discovery of new ones. Clearly, achieving parallelism is not the objective of the initial functional program. Typically, parallelism and communication are introduced at a later stage during the development process, for the sole purpose of capturing functionally equivalent but more efficient parallel algorithms.

The transformational derivation of each algorithm is structured into two stages. The first aims at identifying implicit *pipelined parallelism* in the functional specification. This is achieved by transforming it into an instance of a generic parallel functional form called *pipe pattern*. The transformation is conducted using Bird and Meertens calculus and is guided by new methods for identifying pipelined parallelism. The second stage aims at refining the functional instance of the *pipe pattern* into an appropriate static pipeline network of communicating processes in CSP. This is achieved whithin an algebraic framework based on earlier work by the author. It uses refinement concepts and transformation laws introduced in [1, 2, 8]. The individual processes in the network are usually instantiations of a single parameterized process in CSP. Connections between the processes are fixed throughout the execution of the network and communications take place only locally between neighbouring processes. Pipe patterns can be efficiently implemented on a variety of parallel machines which includes systolic arrays, pipelined, and MIMD machines.

The usefuness of the approach is demonstrated by sytematically synthesizing new massively pipelined algorithms for sorting, removing duplicates, and calculating the difference between two lists. The sequential functional version of each of these algorithms runs in quadratic time, but the pipelined version runs in linear time and uses a linear number of processors.

The remainder of this paper is organised as follows. Section 2 contains a brief summary of the notation and Section 3 introduces the concept of refinement from functions to sequential processes in CSP. New techniques for identifying pipelined parallelism in functional programs are introduced in Section 4. The usefulness of these techniques is demonstrated in Sections 5-7 by deriving several new, efficient, massively parallel algorithms for list manipulations. Section 8 describes related work and concludes this paper.

2 Notation

Throughout this paper, we use the functional notation and calculus developed by Bird and Meertens [3, 5] for specifying algorithmics and reasoning about them. We also use the CSP notation and its calculus developed by Hoare [8] for specifying processes and reasoning about them. We give a brief summary of the notation and conventions used in this paper.

Lists are finite sequences of values of the same type. The list concatenation operator is denoted by $+\!\!+$ and the list construction operator is denoted by : (pronounced "cons"). Functional application is denoted by a space and functional composition is denoted by $\circ$. The operator $*$ (pronounced "map") takes a function on the left and a list on the right and maps the function to each element of the list. Informally, we have:

$$f * [a_1, a_2, \ldots, a_2] = [f(a_1), f(a_2), \ldots, f(a_n)]$$

The operator / (pronounced "reduce") takes an associative binary operator on

the left and a list on the right and is informally described as follows

$$(\oplus)/[a_1, a_2, \ldots, a_n] = a_1 \oplus a_2 \oplus \cdots \oplus a_n$$

In order to concisely describe structured networks of processes we find it very convenient to use, in addition to the CSP notation, functions which return processes and functional operators such as map $(*)$ and reduce $(/)$. For example, if F is a function which returns processes, $\oplus$ is an associative CSP operator and $[a_1, a_2, \ldots, a_n]$ is a list of values, we have

$$\begin{aligned} F * [a_1, a_2, \ldots, a_2] &= [F(a_1), F(a_2), \ldots, F(a_n)] \\ (\oplus)/F * [a_1, a_2, \ldots, a_n] &= F(a_1) \oplus F(a_2) \oplus \cdots \oplus F(a_n) \end{aligned}$$

In CSP, the notation $P \lhd bool \rhd Q$ is just an infix form for the traditional selection construct **if** *bool* **then** P **else** Q. In general, we use identifiers with lower case letters to name functional values and with upper case letters to name processes or types. Occasionally, we will underline a symbol, such as $\underline{F}$, in order to emphasize the fact that it is a function which returns processes. We will also use the notation $F(x)$ instead of $(\underline{F}\ x)$ to denote the process obtained by applying the function $\underline{F}$ to the value x. When parsing expressions, we assume that functional application has the highest precedence and associates to the left, but all other functional operators have equal precedence and associate to the right. For example, the expression $F * s +\!\!+ t$ means $F * (s +\!\!+ t)$ and not $(F * s) +\!\!+ t$.

3 Refinement from Functions to Processes

The refinement from functions to CSP processes is based on the formal treatment given in [2, 1]. In general, there could be many semantically different sequential processes which refine (or correctly implement) a given function. Some of these processes are more suitable than others in the context of parallel computations. The most useful functions for designing algorithms as networks of communicating processes are those which manipulate lists. A function $f :: [A] \to [B]$ can be viewed as a specification of a pipe process which consumes a stream of values (argument) on the input channel and produces a stream of values (result) on the output channel. By convention, the end of each stream is denoted by a special symbol *eot* indicating "end of transmission". In CSP, a pipe process Q is said to refine a function $f :: [A] \to [B]$ iff for all lists s drawn from the domain of f, the output stream of Q is $f(s) +\!\!+ [eot]$ whenever the input stream is $s +\!\!+ [eot]$. This concept is illustrated in Figure 1.

Formally, a pipe process Q is a refinement of a function $f :: [A] \to [B]$, written as $(f \prec Q)$, iff the following condition holds:

$$\forall s \in \mathrm{dom}\ f \bullet Prd(s) \rhd Q = Prd(f\ s)$$

Fig. 1. A process Q refining a function f.

The operator $\triangleright$, see [2, 1] for full details, is similar to the CSP piping operator $\gg$ except that the left operand of $\triangleright$ is a producer (a process which can only output). For any list s, the producer process $Prd(s)$ is defined by the equations

$$\begin{aligned} EOT &= !eot \rightarrow SKIP \\ Prd\ [] &= EOT \\ Prd\ (x : s) &= !x \rightarrow Prd(s) \end{aligned}$$

The proof that a process Q refines a function f , that is $(f \prec Q)$, is usually established by a simple inductive argument.

3.1 Examples

We consider several useful functions and present a typical refinement of each of these functions as a sequential process in CSP. The resulting processes will be the basic building blocks for the construction of several parallel algorithms which will be encountered later. Correctness proofs of the refinement and techniques for systematically synthesizing sequential processes in CSP from functional definitions can be found in [1].

- **The identity function** $id_{[A]}$: The identity function over lists of values $id_{[A]} :: [A] \rightarrow [A]$ can be refined by any bounded buffer process and, in particular, by the process:

$$COPY = \mu X \bullet ?x \rightarrow !x \rightarrow (SKIP \ \triangleleft x = eot \triangleright \ X)$$

- **The function** *map*: For any function $f :: A \rightarrow B$, the map function $f* :: [A] \rightarrow [B]$ can be refined by the process:

$$MAP(f) = \mu X \bullet ?x \rightarrow (EOT \ \triangleleft x = eot \triangleright \ !(f\ x) \rightarrow X)$$

- **The function** *filter*: For any predicate $p :: A \rightarrow bool$, the filter function $(filter\ p) :: [A] \rightarrow [A]$ can be refined by the process:

$$FILTER(p) = \mu X \bullet ?x \rightarrow (EOT \ \triangleleft x = eot \triangleright \ (!x \rightarrow X \ \triangleleft p\ x \triangleright \ X))$$

4 Decomposition Strategies for Pipelined Parallelism

Pipelined parallelism is a very effective means for achieving efficiency in numerous algorithms. It is generally much harder to detect than data parallelism. The *function decomposition* strategy aims at exhibiting pipelined parallelism in functional programs. The fundamental objective of this strategy is to transform a given algorithmic expression into a new form in which the dominant term is a composition of several functions. To fully appreciate the usefulness of this transformation, we will appeal to a basic result, shown in [1], that the composition of functions is naturally refined in CSP by the piping operator as follows

$$\begin{array}{l} f :: [A] \to [B];\ g :: [B] \to [C] \\ g \circ f \\ \hline\hline \\ F \gg G \end{array} \quad \{\ f \prec F \wedge g \prec G$$

By an inductive argument, using the associativity of $\gg$, this result can be generalised so that the composition of any finite list of functions, say $[f_1,\ f_2, ..,\ f_{n-1},\ f_n]$, is refined by piping the list of processes $[F_n,\ F_{n-1}, ..,\ F_2,\ F_1]$ where for each index i, $1 \leq i \leq n$, the process F_i is a refinement of the function f_i.

4.1 Pipe Patterns

There are several general recursive functional forms, called *pipe patterns* [1], which can be systematically transformed into networks of linearly connected processes in CSP. These patterns encapsulate algorithmic definitions which are frequently encountered in functional specifications. They are generally suitable for massively parallel implementations. A particular *pipe pattern* which will be used to derive all the parallel algorithms presented in this paper has the form:

$$\begin{array}{l} spec :: [A] \to [B];\ f :: A \to ([B] \to [B]);\ e :: [B] \\ spec\ [] \quad\ = e \\ spec\ (a{:}\ s) = f\ a\ (spec\ s) \end{array}$$

An alternative formulation of this pattern can be captured by the higher order function *foldr* as $(spec\ s = foldr\ f\ e\ s)$. This pattern has a high degree of implicit parallelism. The parallelism can be clearly identified by using the function decomposition strategy. All we need is to transform $(spec\ s)$ into an expression in which the dominant term is of the form $(\circ)/fs$, for some list of functions fs. This is achieved by transforming the recursive definition of *spec s* into a new form defined using the composition of a list of functions, namely $((\circ)/f * s)\ e$. The informal justification for this transformation is as follows:

$$\begin{array}{rl} spec\ s & = spec\ [a_1, a_2, \ldots, a_n] \\ & = f\ a_1\ (spec\ [a_2, a_3, \ldots, a_n]) \\ & = f\ a_1\ (f\ a_2\ (spec\ [a_3, \ldots, a_n])) \\ & = (f\ a_1 \circ f\ a_2)\ (spec\ [a_3, \ldots, a_n]) \\ & = (f\ a_1 \circ f\ a_2 \circ \cdots \circ f\ a_n)\ (spec\ []) \\ & = ((\circ)/f * [a_1, a_2, \ldots, a_n])\ e \\ & = ((\circ)/f * s)\ e \end{array}$$

A formal proof of this recursion unrolling rule is straightforward by induction. Now provided that for all values a in A, the function $(f\ a)$ is refined by a process $F(a)$ in CSP, then $spec(s)$ can be refined into the following network:

$$SPEC(s) = Prd(e) \triangleright (\gg)/\, F * (reverse\ s)$$

The proof of this result directly follows from the refinement of function composition and the refinement of function application [2, 1]. If the list s contains n values, that is $s = [a_1, a_2, \ldots, a_n]$, then *spec* s can be implemented as a pipe of $(n+1)$ processes. Processes in the pipe are mainly instances of a single process F. The network $SPEC([a_1, a_2, \ldots, a_n])$, can be pictured as follows:

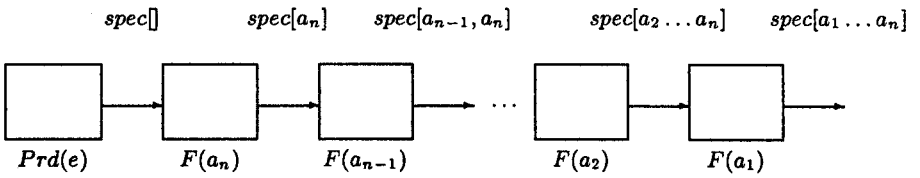

Fig. 2. $SPEC([a_1, a_2, \ldots, a_n])$

5 Parallel Insert Sort

The functional specification of the insertion sort algorithm *isort* can be recursively captured as follows:

$$\begin{array}{lll} isort :: [\alpha] \to [\alpha];\ insert :: \alpha \to [\alpha] \to [\alpha] & & \\ isort\ [] & = [] & \\ isort\ (a{:}\ s) & = insert\ a\ (isort\ s) & \\ & & \\ insert\ a\ [] & = [a] & \\ insert\ a\ (x{:}\ s) & = x{:}\ insert\ a\ s, & \text{if } x < a \\ & = a{:}\ x{:}\ s, & \text{otherwise} \end{array}$$

This algorithm is usually regarded as inherently sequential and cannot be effectively parallelized. We will show that by using new techniques for exploiting pipelined parallelism, this algorithm can be transformed into an efficient massively pipelined network of communicating processes in CSP. To achieve this, observe that the definition of *isort* is expressed as a *pipe pattern*. In this case the starting value e is [] and the step function f is *insert*. Therefore, it follows from the previous section that for all lists s, $isort(s)$ can be implemented as the following network:

$$ISORT(s) = EOT \triangleright ((\gg)/\, \underline{INSERT} * (reverse\ s))$$

where for all values a, the sequential process $INSERT(a)$ is a refinement of the function $(insert\ a)$. Here is a possible refinement of $(insert\ a)$:

$$INSERT(a) = \mu X \bullet ?x \rightarrow \begin{array}{ll} (\ !a \rightarrow !eot \rightarrow SKIP & \{\!| x = eot |\!\} \\ \ \ !x \rightarrow X & \{\!| x < a |\!\} \\ \ \ !a \rightarrow !x \rightarrow COPY\ ) & \end{array}$$

The diagram in Figure 3 depicts how the network $ISORT([5,4,8,9,3,5,8])$ may evolve with time by illustrating the timed behaviour of the individual processes in the network. Note that the input stream for each process in the network is displayed on the horizontal line below it and its output stream is displayed on the line above it. Communications can only take place between neighbouring processes in a synchronized fashion, that is the output of each process is simultanously input to the process above it.

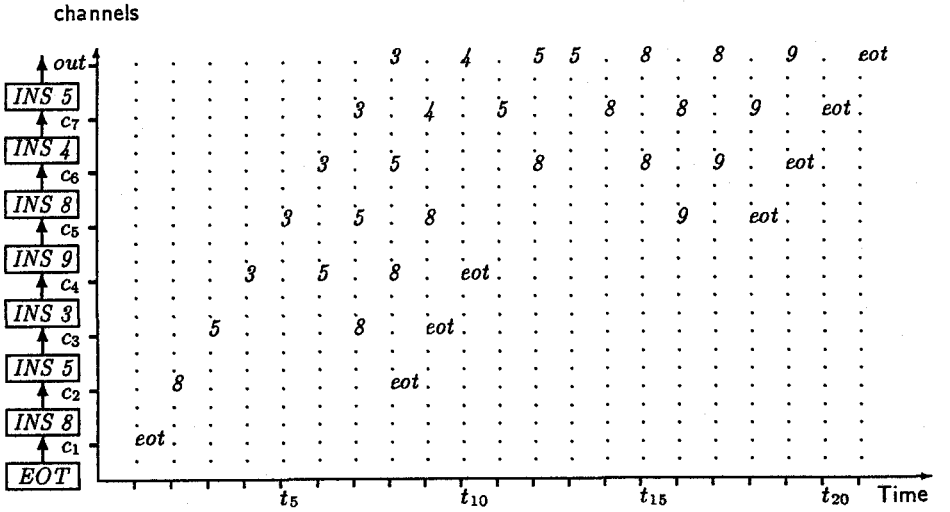

Fig. 3. Time diagram for the computation of $ISORT([5,4,8,9,3,5,8])$

To analyse the time complexity of the network $ISORT(s)$ for a list of length n, say $T_{ISORT}(n)$, observe that the first element of the result is output on the external channel after n computational steps, after which the remaining elements of the sorted list will successively appear on the output channel after two steps each (one communication and one comparison). Hence, $T_{ISORT}(n) = O(n)$. Therefore, using n processes, the parallel implementation of $isort(s)$ shows an ideal $O(n)$ speed up over the sequential implementation.

6 Parallel Removal of Duplicates

The function $rmdup :: [\alpha] \to [\alpha]$ that removes all but the first occurrence of each element in a list is defined as:

$$\begin{aligned} rmdup\ [] &= [] \\ rmdup\ (x{:}\ xs) &= x : filter\ (\neq x)\ (rmdup\ xs) \end{aligned}$$

This function is automatically executed in some relational database implementations in order to normalize the result of some relational operators such as *select*, *join*, *union*, and *project*. Clearly, this function matches the *pipe pattern* form. In this case, $e = []$ and the step function f is

$$f\ x\ ys = x : filter\ (\neq x)\ ys$$

Hence, for all lists s, a parallel implementation of $(rmdup\ s)$ is synthesized as:

$$\begin{aligned} RMDUP(s) &= EOT \triangleright ((\gg)/\ (\underline{F} * (reverse\ s))) \\ F(x) &= !x \to FILTER(\neq x) \end{aligned}$$

The diagram in Figure 4 depicts how the network $RMDUP([4,3,4,5,3,5,2,3])$ may evolve with time. Observe that, the sequential implementation of *rmdup* has a quadratic time complexity but the pipelined version runs in linear time.

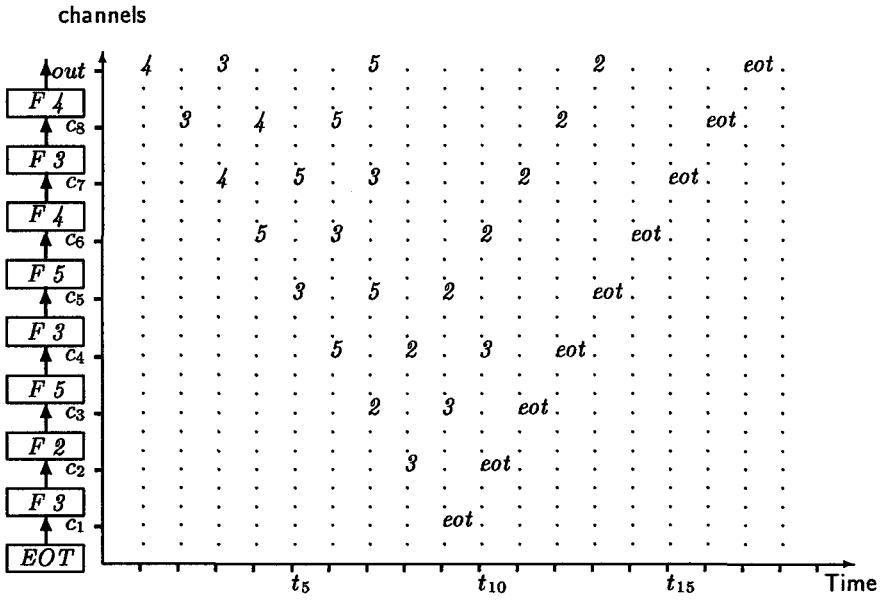

Fig. 4. Time diagram for the network $RMDUP[4,3,4,5,3,5,2,3]$

7 Parallel List difference

The list difference operator *listdiff*, also written as $(-)$, is recursively defined as:

$$\begin{aligned} &\mathit{listdiff}\ xs\ [] &&= xs \\ &\mathit{listdiff}\ xs\ (y\colon ys) &&= \mathit{remove1}\ y\ (\mathit{listdiff}\ xs\ ys) \end{aligned}$$

where the function $\mathit{remove1}\ y$ takes a list s and removes the first occurrence of y, if any, from s. For example, we have: *listdiff* *"parallel"* *"table"* = *"prall"*. It is easy to see that if we take $\mathit{spec} = \mathit{listdiff}\ xs$ then the above definition will immediatly match the *pipe pattern* form. In this case, we have $e = xs$ and $f = \mathit{remove1}$. Hence, by unrolling recursion we get:

$$\mathit{listdiff}\ xs\ ys = ((\circ)/\mathit{remove1} * ys)\ (\mathit{listdiff}\ xs\ [])$$

and now provided that the function *remove*1 is refined into a sequential process, say RM, then for all lists xs and ys, *listdiff* xs ys is refined,into the network:

$$LISTDIFF(xs, ys) = Prd(xs) \rhd ((\gg)/(\underline{RM} * (\mathit{reverse}\ ys)))$$

$$\begin{aligned} RM(z) = \mu X \bullet ?x \rightarrow (\ & EOT && \not< x = eot \not> \\ & !x \rightarrow X && \not< x \neq z \not> \\ & COPY) \end{aligned}$$

The timing diagram in Figure 5 depicts how the computation in the network $LISTDIFF(\text{"}parallel\text{"}, \text{"}table\text{"})$ may evolve with time. Again with pipelined parallelism, we have synthesized a linear time network from a quadratic time recursive algorithm.

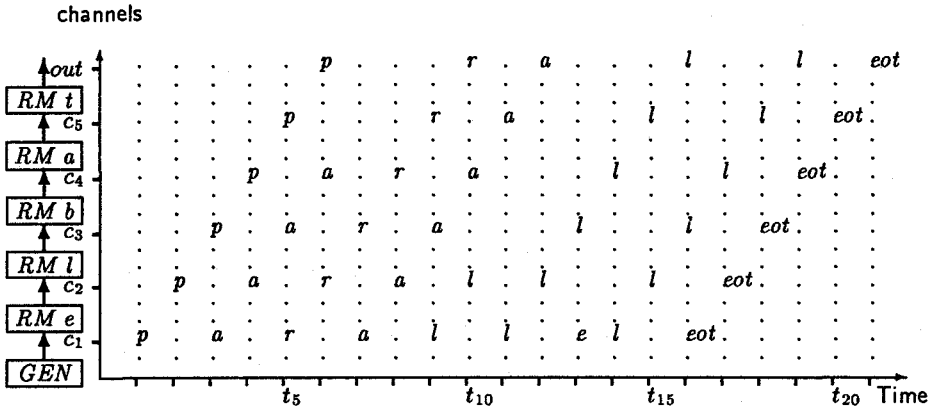

Fig. 5. Animation of the network $LISTDIFF(\text{"}parallel\text{"}, \text{"}table\text{"})$

8 Related Work and Conclusion

Many researchers have proposed the use of functional notations and formalisms for programming parallel machines. The focus has mainly been on exploiting data parallelism rather than pipelined parallelism. Darlington, Field, Harrison, Kelly [7], Cole, Mou, Huddak, Misra, Runciman, Wakeling[10], Partsch, Pepper [9] and Skillicorn [11], to name bu a few, have all used functional notations and algebraic laws to develop parallel functional programs. Our work goes a step further in refining the final functional version into networks of CSP processes. This allows us to address, reason about, and study the effects of implementational issues such as, locality of communications, granularity of computations and scalability.

In summary, we have presented several efficient massively pipelined algorithms for list manipulations. Transformational programming has been used in the development of these algorithms from clear functional specifications to massively pipelined networks of communicating processes in CSP. We have developed new techniques for identifying *pipelined parallelism* in functional specifications and showed how this parallelism can be efficiently realized in CSP. By applying these techniques, the time complexity of several algorithms has been reduced from quadratic to linear time.

Acknowledgments I would like to thank Richard Bird, Jeff Sanders, Philip Wadler, Mark Josephs and three anonymous referees.

References

1. A. E. Abdallah, Derivation of Parallel Algorithms from Functional Specifications to *CSP* Processes, in: Bernhard Möller, ed., *Mathematics of Program Construction*, LNCS **947**, (Springer Verlag, 1995) 67-96.
2. A. E. Abdallah, An Algebraic Approach for the Refinement of Functional Specifications to *CSP* Processes, Submitted to *Formal Aspects of Computing*, 1996.
3. R. S. Bird, An Introduction to the Theory of Lists, in M. Broy, ed., *Logic of Programming and Calculi of Discreet Design*, (Springer, Berlin, 1987) 3-42.
4. R. S. Bird, J. Gibbons, and G. Jones, Formal derivation of a pattern matching algorithm, *Science of Computer Programming* **12** (1989), 93-104.
5. R. S. Bird, and P. Wadler, *Introduction to Functional Programming*, (Prentice-Hall, 1988).
6. CIP language group, *The Munich project CIP*, LNCS **1**, (Springer-Verlag, 1984).
7. J. Darlington, A. Field, P. Harrison, P. Kelly, D. Sharp and Q. Wu, Parallel Programming Using Skeleton Functions. In *Parallel Architectures and Languages Europe, PARLE93.* LNCS **947**, (Springer Verlag, 1993) 146-160.
8. C. A. R. Hoare, *Communicating Sequential Processes.* (Prentice-Hall, 1985).
9. R. Paigee, J, Reif, snd R. Wachter (eds), *Parallel Algorithm Derivation and Program Transformation*, (Kluwer,1993).
10. C. Runciman and D. Wakeling (eds), *Applications of Functional Programming*, (UCL press, 1995).
11. D. B. Skillicorn, Models for Practical Parallel Computation, *International Journal of Parallel Programming* **20** (2) (1991) 133-158.