Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Systems
October 16-19, 1996 Chicago, Illinois, USA

Synthesis of Massively Pipelined Algorithms from Recursive Functional Programs

Ali E. Abdallah

Department of Computer Science
The University of Reading
Reading, RG6 6AY, UK
email: A.Abdallah@reading.ac.uk

Theoharis Theoharis

Department of Informatics
The University of Athens
Panepistimioupolis, 15771 Athens, Greece
email: theotheo@di.uoa.gr

Abstract

The purpose of this paper is to show how a class of recursively defined functional algorithms can be efficiently implemented as massively parallel networks of Communicating Sequential Processes (CSP). The method aims at achieving efficiency by exploiting pipelining parallelism which is inherent in functional programs. The backbone of the method is a collection of powerful transformation rules for unrolling recursion. Each of these rules transforms a generic recursive functional definition into a composition of a fixed number of simpler functions. Parallelism is explicitly realized in CSP by implementing the composition of functions as piping of appropriate processes.

Keywords: Functional programming; Program transformation; Pipelined parallelism.

1 Introduction

Recursion plays an essential role in the design and formulation of numerous algorithms. The prominent role of recursion is most evident in the functional style of programming where most non-trivial functions are defined recursively. Often, recursive definitions are interpreted in an inherently sequential manner which does not allow substantial opportunities for parallelism. However, a deeper analysis based on new techniques introduced in [1, 2] may reveal just the opposite.

For example, the *insertion sort* algorithm which is recursively defined as:

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

is usually regarded as inherently sequential and cannot be effectively parallelized. Although, a data parallel execution strategy for this algorithm may not result in substantial gain in efficiency, a parallel execution stategy aimed at exploiting pipeline parallelism in this algorithm is shown to be much more effective. It results in transforming this quadratic time sequential algorithm into a linear time parallel algorithm with a linear number of concurrent processes. The algorithm is scalable because communications between the processing elements are local.

The purpose of this paper is to present a method for systematically transforming certain recursive functional programs into massively parallel networks of processes. The method is based on several powerful recursion unrolling transformation rules. Each of these rules expresses a generic recursive functional definition in an iterative form as a composition of a fixed number of simpler functions (a pipe pattern). Function composition can be subsequently refined into process piping and, therefore, the whole recursive functional definition is transformed into a massively pipelined network of processes.

Starting from an initial recursive functional definition of the problem, the synthesis is done in two stages. The first stage aims at identifying pipelined parallelism in the functional definition. This is achieved using one of the recursion unrolling rules which transform the specification into a *pipe pattern* (a composition of a fixed number of functions). The second stage aims at refining the functional instance of the *pipe patten* into an appropriate static pipeline network of communicating CSP processes. This is achieved whithin an algebraic framework based on earlier work and using refinement concepts and transformation laws introduced in [1, 7].

2 Notation

Throughout this paper, we use the functional notation and calculus developed by Bird and Meertens [4, 5] for specifying algorithmics and reasoning about them. We also use the CSP notation and its calculus developed by Hoare [7] 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:

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

The operator / (pronounced "reduce") takes an associative binary operator on the left and a list on the right and is informally described as follows

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

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

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

In CSP, the notation $P \not< bool \not> Q$ 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.

3 Refinement from Functions to Processes

The refinement from functions to CSP processes is based on the formal treatment given in [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 $f :: [A] \rightarrow [B]$ 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 $f :: [A] \rightarrow [B]$ 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.

Figure 1: A process Q refining a function f.

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

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

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

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

For example 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:

$$COPY \quad = \mu X \bullet ?x \rightarrow (\ EOT \ \not< x = eot \not> !x \rightarrow X\ )$$

4 Decomposition Strategies for Pipelined Parallelism

Pipeline 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 pipeline 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

$$\frac{f :: [A] \to [B];\ g :: [B] \to [C] \quad g \circ f}{F \gg G} \quad f \prec F \wedge g \prec G$$

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.

5 Parallelizing Tail Recursion

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. The first pattern which will consider encapsulates the general form of *tail recursion*. It can be described as:

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

where e and f are parameters. 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 exhibited 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 using the following recursion unrolling rule:

(Recursion Unrolling RU1)

$$\frac{\begin{array}{l} spec :: [A] \to [B];\ f :: A \to ([B] \to [B]);\ e :: [B] \\ spec\ [] = e \\ spec\ (a : s) = f\ a\ (spec\ s) \end{array}}{spec\ s = ((\circ)/\,f * s)\ e}$$

The informal justification for this transformation is as follows:

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

A formal proof of this recursion unrolling rule is straightforward by induction. Now provided that $f \prec \underline{F}$, $spec(s)$ can be refined into the following network of communicating processes

$$SPEC(s) \quad = \quad Prd(e) \rhd (\gg)/\,\underline{F} * (reverse\ s)$$

The proof of this result directly follows from the refinement of function composition and the refinement of function application. 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 instantiations of a single process $\underline{F}$. The network $SPEC([a_1, a_2, \ldots, a_n])$, can be pictured as depicted in Figure 2.

In order to ensure efficiency of the resulting parallel implementation $SPEC([a_1, a_2, \ldots, a_n])$, the function f must have an on-line implementation.

5.1 Example: Parallel Insert Sort

The recursive functional definition of the insertion sort algorithm *isort* introduced in Section 1 is just a special case of the tail recursion form. In this case we have 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:

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

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

$$\begin{array}{lll} \mu X \bullet ?x \to (& !a \to !eot \to SKIP & \nleq x = eot \nrightarrow \\ & !x \to X & \nleq x < a \nrightarrow \\ & !a \to !x \to COPY\ ) & \end{array}$$

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. This animation is based on [3]. 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.

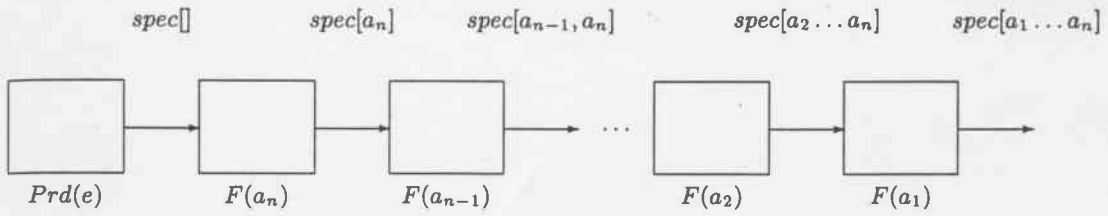

Figure 2: $SPEC([a_1, a_2, \ldots, a_n])$

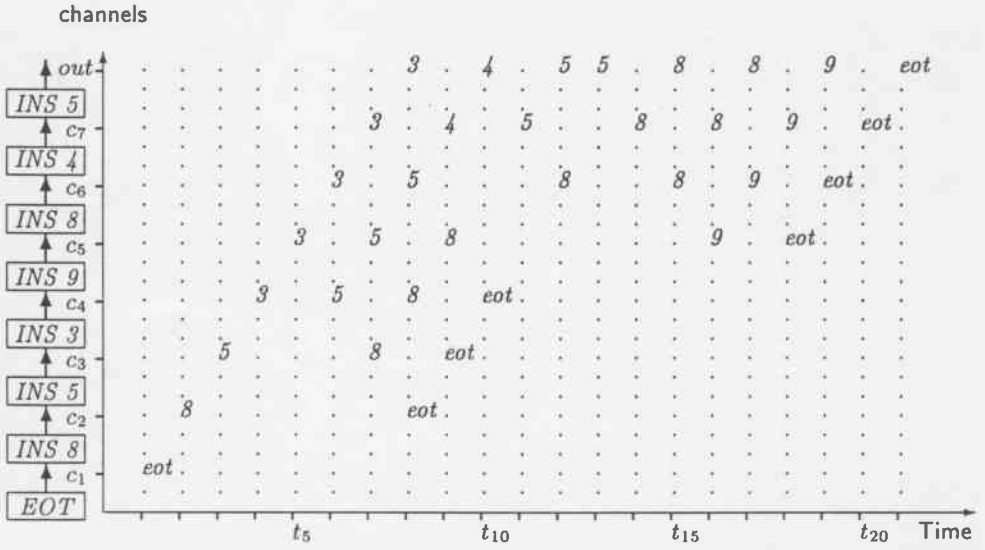

Figure 3: Time diagram depicting the pipelined computation of $ISORT([5, 4, 8, 9, 3, 5, 8])$

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 Parallelizing Recursion with Parameter Accumulation

Parameter accumulation is one of the basic techniques for achieving efficiency, in the serial sense, in functional programs [6]. The definition of a function is usually modified so that it takes an extra parameter for accumulating the result after each recursive call. Typically, the computation of a list of values usually starts from the first element and proceeds towards the last. This is in contrast with tail recursion where computation starts from the last element of the list and proceeds towards the first. The general functional pattern which is used for this purpose has the following form:

$$
\begin{array}{lcl}
spec :: [B] \to [A] \to [B];\ add :: A \to [B] \to [B] & & \\
spec\ t\ [] & = & t \\
spec\ t\ (a{:}\ s) & = & spec\ (add\ a\ t)\ s
\end{array}
$$

An alternative formulation of this pattern can be captured using the higher order function $foldl$ as $spec\ t\ s = foldl\ add\ t\ s$, for details see Bird and Wadler's book [5]. It is a well known fact that the computation of the above pattern can be efficiently achieved using a stack machine [6]. At first glance, there seems to be no serious opportunities for exhibiting parallel computation in this pattern. However, a deeper examination reveals just the opposite. Our objective is to show how the above pattern can be transformed into a highly parallel network of communicating processes. We aim to achieve this by transforming *spec* into a pipe pattern. To do so, we define a new function h as follows

$$h\ t\ s \quad = \quad ((\circ)/\ add * (reverse\ s))\ t$$

Informally, we have

$$h\ t\ [a_1, a_2, \cdots, a_k] \quad = \quad ((add\ a_k) \circ \cdots \circ (add\ a_1))\ t$$

The function h is clearly expressed as a pipe pattern. The surprising thing is that the functions *spec* and h are identical. The proof of this claim is done by induction.

The corresponding CSP implementation of *spec* t s is as follows

$$SPEC(t, s) \quad = \quad Prd(t) \rhd ((\gg)/\ ADD * s)$$

where the process $ADD(a)$ refines the function $add(a)$ for all values of the parameter a. This parallel implementation of *spec* can be substantially more efficient than the sequential one provided that, for any given a, the function $(add\ a)$ is incrementally computable. Finally, we can encapsulate the whole derivation in the following transformation rule:

(Recursion Unrolling RU2)

$spec :: [B] \to [A] \to [B];\ add :: A \to [B] \to [B]$

$$\begin{array}{lcl} spec\ t\ [] & = & t \\ spec\ t\ (a : s) & = & spec\ (add\ a\ t)\ s \end{array}$$

$$\Updownarrow$$

$$spec\ t\ s \quad = \quad ((\circ)/\ add * (reverse\ s))\ t$$

6.1 Example: Converting lists to bags

The function *mkbag* converts a list into a bag. For any list xs, $(mkbag\ xs)$ is a list of pairs. Each pair has the form (x, i) where x is an elememt of xs and i is a positive number indicating the count of occurrences of x in the list xs. For example, we have:

$$mkbag\ \text{"}parallel\text{"} = [(p, 1), (a, 2), (r, 1), (l, 3), (e, 1)]$$

The function *mkbag* can be defined recursively as follows:

$$\begin{array}{lcl} mkbag :: [\alpha] \to [(\alpha, num)] & & \\ mkbag\ s & = & mkbag'\ []\ s \\ mkbag'\ t\ [] & = & t \\ mkbag'\ t\ (x : s) & = & mkbag'\ (add\ a\ t)\ s \end{array}$$

$$\begin{array}{lcll} add :: \alpha \to [(\alpha, num)] \to [(\alpha, num)] & & & \\ add\ x\ [] & = & [(x, 1)] & \\ add\ x\ ((y, i) : ys) & = & (y, i) : add\ x\ ys & \text{if } x \neq y \\ & = & (y, i + 1) : ys & \text{otherwise} \end{array}$$

The function *mkbag'* matches the recursive definition of *spec* with parameter accumulation. Therefore, provided that the process $ADD(x)$ correctly implements the function $add\ x$, $mkbag\ s$ can be implemented as the following pipelined network of processes:

$$MKBAG(s) \quad = \quad Prd([]) \rhd (\gg)/\ ADD * s$$

For all x, the process $ADD(x)$ can be defined as follows:

$$\begin{array}{lll} \mu X \bullet ?z \to (& !(x, 1) \to EOT & \nleftarrow z = eot \nrightarrow \\ & !(x, i + 1) \to COPY & \nleftarrow\ x = y\ \nrightarrow \\ & !z \to X) & \\ & \text{where } z = (y, i) & \end{array}$$

Assuming that the length of the list s is n, the sequential implementation of $mkbag(s)$ requires $O(n^2)$ steps but the parallel implementation is linear.

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, Synthesis of Massively Pipelined Algorithms for List Manipulation, Proceedings of the European Conference on Parallel Processing, EuroPar'96, LNCS **1024**, (Springer Verlag, 1996), pp 911-920.

[3] A. E. Abdallah, Visualisation and Animation of Networks of Communicating Processes, Proceedings of the Eighth IASTED International Conference on Parallel and Distributed Computing and Systems, Chicago, USA, October 1996.

[4] 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.

[5] R. S. Bird, and P. Wadler, *Introduction to Functional Programming*, (Prentice-Hall, 1988).

[6] P. Henderson, *Functional Programming: Application and Implementation.* Prentice-Hall, 1980.

[7] C. A. R. Hoare, *Communicating Sequential Processes.* (Prentice-Hall, 1985).