Parallel Algorithms for Fast Fourier Transformation Using *PowerList*, *ParList* and *PList* Theories

Virginia Niculescu

Department of Computer Science
Babeş-Bolyai University
Cluj-Napoca, Romania
gina@cs.ubbcluj.ro

Abstract. *PowerList*, *ParList* and *PList* data structures are efficient tools for functional descriptions of parallel programs that are *divide & conquer* in nature.
The goal of this work is to develop three parallel variants for Fast Fourier Transformation using these theories. The variants are implied by the degree of the polynomial, which can be a power of two, a prime number, or a product of prime factors. The last variant includes the first two, and represents a general and efficient parallel algorithm for Fast Fourier Transformation. This general algorithm has a very good time complexity, and can be mapped on a recursive interconnection network.

1 Introduction

PowerList, *ParList* and *PList* are data structures that can be successfully used for simple functional descriptions of parallel programs that are *divide&conquer* in nature. To assure methods for verification of the parallel programs correctness, algebras and induction principles are defined on these data structures[1].

A *PowerList* is a linear data structure whose elements are all of the same type. The length of a *PowerList* data structure is a power of two. A *PowerList* with 2^n elements of type X is specified by $PowerList.X.n$. Two similar *PowerList*s can be combined into a *PowerList* data structure with double length in two different ways: using *tie* operator $(p \mid q)$ when the result contains elements from p, followed by elements from q, and using *zip* operator $(p \natural q)$ when the result contains elements from p and q, alternatively taken.

Example 1. (*Polynomial Value*)

A polynomial with coefficients $(a_i, 0 \leq i < 2^n)$, where $n \geq 0$, may be represented by a $PowerList$ – p, whose ith element is a_i. The following function vp evaluates a polynomial p; vp accepts an arbitrary $PowerList$, which contains the points, as its second argument.

$$\begin{array}{l} vp : PowerList.X.n \times PowerList.X.m \rightarrow PowerList.X.m \\ vp.[a].[z] \quad = [a] \\ vp.p.(u|v) \;\; = vp.p.u \mid vp.p.v \\ vp.(p\natural q).w = vp.p.w^2 + (w \cdot (vp.q.w^2)) \end{array} \tag{1}$$

B. Monien and R. Feldmann (Eds.): Euro-Par 2002, LNCS 2400, pp. 400–404.

The length of a *ParList* data structure is not always a power of two. A *ParList* with n elements of type X is specified by $ParList.X.n$. It is necessary to use two other operators: $cons(\triangleright)$ and $snoc(\triangleleft)$; they allow the adding of an element to a *ParList*, at the beginning or at the end of the *ParList*.

Example 2. (*Polynomial Value*)

$$\begin{array}{l} vp : ParList.X.n \times ParList.X.m \to ParList.X.m \\ \begin{array}{l|l} vp.[a].[z] = a & \\ vp.(p \natural q).w = vp.p.w^2 + w \cdot vp.q.w^2 & vp.p.(u \mid v) = vp.p.u \mid vp.p.v \\ vp.(a \triangleright p).w = a + w \cdot vp.p.w & vp.p.(z \triangleright w) = vp.p.[z] \ \triangleright \ vp.p.w \end{array} \end{array} \tag{2}$$

*PList*s are constructed with the n-way $\mid$ and $\natural$ operators; for a positive integer n, the n-way $\mid$ takes n similar *PList*s and returns their concatenation, and the n-way $\natural$ returns their interleaving. Functions over *PList* are defined using two arguments. The first argument is a list of arities of type *PosList* (*PosList* is the type of linear lists with positive integers), and the second is the *PList* argument. Functions over *PList* are only defined for certain pairs of these input values.

Example 3. (*Polynomial Value*)

We extend the strategy used for *PowerList* case by using different radices, and we define the function vp on *PList*s (we use the notation $\overline{n} = \{0, \ldots, n-1\}$):

$$\begin{array}{l} vp : PosList \times PList.X.n \times PosList \times PList.X.m \to PList.X.m \\ defined.vp.lx.p.ly.w \equiv prod.lx = length.p \wedge prod.ly = length.w \\ vp.[].[a].[].[z] = a \\ vp.lx.p.(y \triangleright ly).[|i : i \in \overline{y} : w.i] = [|i : i \in \overline{y} : vp.lx.p.ly.(w.i)] \\ vp.(x \triangleright lx).[\natural i : i \in \overline{x} : p.i].ly.w = (+i : i \in \overline{x} : w^i \cdot vp.lx.(p.i).ly.w^x) \end{array} \tag{3}$$

2 Fast Fourier Transformation

We consider a polynomial pf with coefficients $(a_i, 0 \leq i \leq n)$. A scalar function will be used in all the cases: function $root : Nat \to Com$ applied to n returns the principal nth order unity root (Com is the type of complex numbers). Three functions named $powers : Com \times PXxxList.Com.n \to PXxxList.Com.n$ will be used; they each return a $PXxxList$ of the same length as the input list(p) containing the powers of the first argument from 0 up to the length of p (Xxx could be $Power, Par$, or P).

2.1 The Case $n = 2^k$

In this case, for the parallel program specification, *PowerList* data structures can be used. The function $fft : PowerList.Com.n \to PowerList.Com.n$ can be defined as:

$$\begin{array}{l} fft.[a] = [a] \\ fft.(p \natural q) = (r + u \cdot s) \mid (r - u \cdot s) \\ \text{where} \\ \begin{array}{l|l} r = fft.p & u = powers.z.p \\ s = fft.q & z = root.(length.(p \natural q)) \end{array} \end{array} \tag{4}$$

The values r, s, and u are independent and can be computed in parallel. So, the time complexity is $O(\log_2 n)$, where n is the length of p.

2.2 The Case n Prime

In this case, it is necessary to compute directly the polynomial values:

$$\begin{array}{l} fft : ParList.Com.n \to ParList.Com.n \\ fft.p = vp.p.(powers.z.p) \end{array} \tag{5}$$

The maximum complexity for the computation of a polynomial value at one point is obtained when $n = 2^k - 1$ ($n = length.p$), when there are $2k-2$ parallel steps, and the minimum complexity is achieved when $n = 2^k$. If we can compute the n polynomial values in parallel, the time complexity is still $O(\log_2 n)$, even if the constant is larger.

2.3 The Case $n = r_1 \cdot \ldots \cdot r_k$

If n is not a power of two, but a product of two numbers r_1 and r_2, the formula for computing the polynomial value can be generalized in the following way:

$$pf.w_j = \sum_{k=0}^{r_1-1} \{ \sum_{t=0}^{r_2-1} a_{tr_1+k} e^{\frac{2\pi ijt}{r_2}} \} e^{\frac{2\pi ijk}{n}}, \quad 0 \le j < n \tag{6}$$

Theorem 1. *The best factorization $n = r_1 \cdot r_2$ for* FFT *(from the complexity point of view) is to choose r_1 from the prime factors of n [4].*

Therefore, for the specification of the parallel algorithm, we consider the decomposition in prime factors $n = r_1 \cdot \ldots \cdot r_k$. The *PList* data structures will be used. The *PosList* is formed by the prime factors of $n : [r_1, r_2, \cdots, r_p]$.

We start the derivation from the classic definition: $fft.l.p = vp.l.p.l.w$, where $w = powers.z.l.p$. Function vp is the one defined on *PLists* (*Example 3*).

We use the notation $W.z.l = powers.z.l.p$. The following properties are true, due to the properties of the unity roots:

$$\begin{array}{ll} W.z.(x \triangleright l) & = [|i : i \in \overline{x} :< z^{\frac{n}{x} i} \cdot > .(W.z.l)], \text{where } n = x \cdot prod.l \\ (W.z.(x \triangleright l))^x & = [|i : i \in \overline{x} : W.z^x.l] \\ (W.z.l)^i & = W.(z^i).l \end{array} \tag{7}$$

We derive a new expression for fft, based on the induction principle:
Base case:

$$\begin{array}{l} \quad fft.[x].[\natural i : i \in \overline{x} : [a.i]] \\ = \{ \text{ definition of } fft, \text{ calculus} \} \\ \quad [|j : j \in \overline{x} : (+i : i \in \overline{x} : a.i \cdot z^{(i \cdot j)})] \end{array} \tag{8}$$

Inductive Step:

$$\begin{array}{l} \quad fft.(x \triangleright l).[\natural i : i \in \overline{x} : p.i] \\ = \quad \{\text{definitions of } fft \text{ and } vp\text{, properties of function } W.z.l\text{, calculus}\} \\ \ldots \\ \quad (+i : i \in \overline{x} : [|j : j \in \overline{x} :< z^{\frac{n}{x}ij} \cdot > .(W.(z^i).l) \cdot fft.l.(p.i)] \end{array} \tag{9}$$

So, the definition of fft is now:

$$\begin{array}{l} fft : PosList \times PList.Com.n \to PList.Com.n \\ defined.fft.l.p \equiv (prod.l = length.p) \\ fft.[x][\natural i : i \in \overline{x} : [a.i]] \quad = [|j : j \in \overline{x} : (+i : i \in \overline{x} : a.i \cdot z^{(i \cdot j)})] \\ \text{where} \quad z = root.x \\ fft.(x \triangleright l).[\natural i : i \in \overline{x} : p.i] = [|j : j \in \overline{x} : (+i : i \in \overline{x} : r.i \cdot u.i.j)] \\ \text{where} \\ r.i = fft.l.(p.i) \quad \Big| \; z = root.n \\ u.i.j =< z^{(ij \cdot \frac{n}{x})} \cdot > .W.(z^i).l \; \Big| \; n = length.[\natural i : i \in \overline{x} : p.i] \end{array} \tag{10}$$

(A special case of the high order function *map* was used: $< z \cdot > .p = map.(z \cdot).p.$)

For the base case, the algorithm presented when n is prime can be used, which is more efficient. If the list of arities contains just values equal to 2, the algorithm becomes the one specified in the first case.

The algorithm for Fast Fourier Transformation can be done simultaneously with the decomposition of n in prime factors. If the prime factors become too large, then we can stop and apply the algorithm used when n is prime.

The time complexity of the algorithm depends on the prime factors of n and on their number m. If we consider that all the prime factors are less than a number M, then the time complexity is $O(m)$, with a constant that depends on M. If, for example, $n = 3^k$, then the time complexity is $O(\log_3 n)$.

We can implement this algorithm using a *recursive* interconnection network[2], which has the same arity list of the nodes like the arity list used for the calculation of fft. The implementation has two stages: a descendent stage and an ascendent stage.

3 Conclusions

The last algorithm for Fast Fourier Transformation is a general parallel algorithm that does not depend on the degree of the polynomial. It was formally derived, and so its correctness was proved.

The time complexity that can be obtained with this algorithm is better than the complexity of the other two (and so better than the classic one); and also the algorithm can be mapped on a classic interconnection network.

References

1. Kornerup, J.: Data Structures for Parallel Recursion. PhD thesis, University of Texas at Austin (1997)

2. Kornerup, J.: PLists: Taking PowerLists Beyond Base Two. In: Gorlatch, S. (ed.): First International Workshop on Constructive Methods for Parallel Programming, MIP-9805 (1998) 102-116
3. Misra, J. : PowerList: A structure for parallel recursion. ACM Transactions on Programming Languages and Systems, Vol. 16 No.6 (1994) 1737-1767
4. Wilf, H.S.: Algorithms and Complexity. Mason & Prentice Hall (1985)