Parallel Dynamic Programming using Data-Parallel Skeletons *

Kazuhiko Kakehi, Kiminori Matsuzaki, Akimasa Morihata, Kento Emoto and Zhenjiang Hu

Department of Mathematical Informatics, University of Tokyo

{kaz,kmatsu,fooofo,emoto,hu}@ipl.t.u-tokyo.ac.jp

Skeletal parallel programming serves us as a successful method for parallelization. It provides a fixed set of program components called skeletons, each of which has its efficient parallel implementation. Their combination enables users to develop parallel programs without concerning themselves with communication deadlocks.

For demonstrating the powerfulness of skeletal programming we have developed a parallelization technique toward one class of optimization problems called maximum marking problems, which can be recognized as one class of dynamic programming. This paper, as in the course of an ongoing research, focuses on problems of dynamic programming in general, and examines how the skeletal approach successfully parallelizes problems of dynamic programming.

1 Introduction

Dynamic programming, which has been first developed by R. Bellman in 1950s, describes a class of sequential computation (or decision) processes, where computation in one stage, or one problem, depends on the results of its previous stages, or its smaller subproblems. This description naturally calls for memoization of computed results, which brings about benefits of avoiding the recomputation of subproblems. This class is often applied to optimization problems. In the viewpoint of optimization problems, the ingredients of dynamic programming are rephrased as *optimal substructure* and *overlapping subproblems* [7].

Indeed dynamic programming realizes both of optimality and efficiency of the computed results, but the data structures required for the computation, which is often a matrix, become huge when the concerned problems become large. This fact often makes users refrain from employing this technique. One approach toward remedying this situation is parallelization. The involvement of matrices and the real world demands of DNA sequencing make it a good topic of parallel computation, and indeed there have been quite a few researches tackling this topic [17, 8, 10, 11, 5, 16, 1, 2].

In the field of parallel programming, *skeletal parallelism* has been proposed as one methodology to develop parallel programs using *skeletons*, that is "*some useful patterns of parallel computation and interaction . . . packaged up as 'framework/second order/template' constructs.*" [6] Skeletons realizes great conciseness in parallel programming by enclosing troublesome communications; data-parallel skeletons directly link to BMF (Bird-Meertens Formalism) [3, 18], providing us a tool of systematic transformation. There are a lot of implementations of skeleton-based parallel programming environments, which run efficiently under not only shared, but also distributed memory environments which are much more obtainable as PC clusters. We are also implementing a skeleton library called *SkeTo* for various data structures [15], which is implemented toward distributed memory environments and has self optimization mechanism [14].

Despite the need of parallel treatments of dynamic programming, skeletal approaches toward this problem are rare, except for one work based on control-parallel skeletons [16]. One of our current

*This research is partially supported by the Ministry of Education, Culture, Sports, Science and Technology, Japan, Grant-in-Aid for Young Scientist (B), No. 17700026, 2005. This research is partially supported by the *Comprehensive Development of e-Society Foundation Software* project of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

research topics is to investigate solutions for dynamic programming using data-parallel skeletons. As a part of ongoing research, this short paper gives a uniform framework called *DAG matrix formalization*, and show how computations of such matrices are effectively realized using list skeletons. To demonstrate the generality of our framework, we examine how well-known classes of dynamic programming are reformatted into our framework. Theoretical complexity of the obtained solutions is enough competing with the solution under CGM/BSP [19, 9], one model of distributed memory environments.

This paper is organized as follows. After this Introduction, we briefly explain parallel skeletons over lists (one dimensional arrays) as well as some other notational conventions in Section 2. Section 3 analyzes the requirements to implement dynamic programming using skeletons, and presents our formalization as well as its skeletal implementation. Section 4 applies our formalization to well-known classes of dynamic programming. Section 5 concludes this paper with some remarks.

2 Preliminaries

2.1 Parallel skeletons over lists

Four higher order functions are often called *list skeletons*, that is map, zipw, reduce and scan.

Map, denoted as an infix $*$, is to apply a function to every element in a list.

$$k * [a_1, a_2, \ldots, a_n] = [k\, a_1, k\, a_2, \ldots, k\, a_n]$$

Zipw is the skeleton that takes two lists and returns a new list through applying a specified operation $\oplus$ to every pair of corresponding elements from the two lists of the same length; therefore

$$\begin{aligned}&[a_1, a_2, \ldots, a_n]\, \Upsilon_\oplus\, [b_1, b_2, \ldots, b_n]\\ &\quad = \ [a_1 \oplus b_1, a_2 \oplus b_2, \ldots, a_n \oplus b_n]\ .\end{aligned}$$

When the operator $\oplus$ is omitted it is assumed to return a list of pairs. Reduce, written as an infix /, is the skeleton which collapses a list into a single value by repeated application of some associative binary operator $\otimes$ with an initial value c.

$$\otimes\, /_c\, [a_1, a_2, \ldots, a_n] = c \otimes a_1 \otimes a_2 \otimes \cdots \otimes a_n$$

Scan accumulates all intermediate results of reduce up above. Informally we have

$$\begin{aligned}&\otimes /\!\!/_c\, [a_1, a_2, \ldots, a_n]\\ &= \ [c, c \otimes a_1, c \otimes a_1 \otimes a_2, \ldots, c \otimes a_1 \otimes a_2 \otimes \cdots \otimes a_n]\ .\end{aligned}$$

This left-to-right scanning is called scanl while the symmetric right-to-left is called scanr. When this accumulative computation refers to every n-th preceding element other than 1, we associate this gap to the operator like $/\!\!/^n$.

In addition to the common data-parallel four skeletons, we use shifting operations in this paper. Shift moves values in a certain number of positions, which is informally specified as follows.

$$\begin{aligned}&\overset{i}{\rightarrow}_d\, [a_1, a_2, \ldots, a_n]\\ &= \begin{cases} [d, \ldots, d, a_1, a_2, \ldots, a_{n-i}] & (i \geq 0)\\ [a_{1-i}, a_{2-i}, \ldots, a_n, d, \ldots, d] & (i < 0)\end{cases}\end{aligned}$$

These five skeletons have nice massively parallel implementations on many architectures [4, 6, 18]. If k, $\oplus$ and $\otimes$ need O(1), then both map $k*$ and zipw $\Upsilon_\oplus$ can be implemented using O(1) parallel time, and both reduce $\otimes\ /_c$ and scan $\otimes /\!\!/_c$ can be implemented using $O(\log n)$ parallel time over an input list of length with abundance of processors. For example, $\otimes\ /_c$ is implemented over a tree-like structure with the combining operator $\oplus$ applied in the nodes. Shift can be realized using O(1) parallel time.

2.2 Notational conventions

Dynamic programming is often defined over matrices. To represent some sequence of values in a matrix, we prepare some shorthands. Assume we are given a matrix $D_{[i,j]}$ of size $\mathrm{N} \times \mathrm{M}$.

$$\begin{aligned}D_{[\_,j]} &= [D_{[0,j]}, \ldots, D_{[\mathrm{N}-1,j]}]\\ D_{[a \leq \cdot \leq b,j]} &= [D_{[a,j]}, \ldots, D_{[b,j]}]\\ D_{[\cdot \leq b,j]} &= [D_{[0,j]}, \ldots, D_{[b,j]}]\\ D_{[a \leq \cdot,j]} &= [D_{[a,j]}, \ldots, D_{[\mathrm{N}-1,j]}]\end{aligned}$$

Shorthands in terms of j are similarly given.

To compute a value in the concerned matrix, auxiliary functions like $w(i, j)$ are often involved whose parameters are identical to indexes of the matrix. We prepare similar shorthands for the sequences of

(0) Given $D[p,0]$ $(0 \le p \le \mathrm{M})$ and w, compute

$$D[p,j] = \min_{0<q\le \mathrm{N}} \{D[q,j-1] + w(j,p,q)\} \quad \text{for } 0 < j < \mathrm{M}$$

(1) Given $D[0]$ and w, compute

$$D[i] = \min_{0\le j<i} \{D[j] + w(i,j)\} \quad \text{for } 0 < i < \mathrm{N}$$

(2) Given $D[0,0]$, $n_{i,j}$, $w_{i,j}$, and $nw_{i,j}$, compute

$$D[i,j] = \min \begin{cases} D[i-1,j] + w_{i,j} \\ D[i-1,j-1] + nw_{i,j} \\ D[i,j-1] + n_{i,j} \end{cases} \text{for } \begin{array}{l} 0 \le i \le \mathrm{N} \\ 0 \le j \le \mathrm{M} \end{array}$$

(3) Given $D[i,i+1]$ (for $0 \le i < \mathrm{N}-1$) and w, compute

$$D[i,j] = \min_{i<r<j} \{D[i,r] + D[r,j] + w(i,r,j)\} \quad \text{for } 0 < i < j < \mathrm{N}$$

Figure 1: Some prominent classifications of dynamic programming

values these functions return: e.g., $w_{(\_,j)}$ denotes $[w(0,j),\ldots,w(\mathrm{N}-1,j)]$.

For the convenience of representation we use variables to hold a list. To be easily noticed, variables for lists have a single overline like $\overline{c}$.

3 Dynamic Programming using List Skeletons

Apart from general specifications of dynamic programming, various application often classify it into several forms. Figure 1 shows some examples of them (some of which have been given by [10]).

3.1 Strategy of skeletal dynamic programming

Skeletons themselves do not provide solutions for dynamic programming directly, in which involved value dependencies reduce sequential computational cost while complicating parallelizability. In skeletal programming, it is a common approach to implement a desired algorithm as combinations of skeletons. This convention naturally raises the following strategy consisting of two steps.

Step 1. *Generation analysis: dynamic programming into recurrence equations.* To resolve the dependencies we first find their *decreasing chain*. By this we split matrix elements into groups, namely *generations*. An element in one generation can depend on elements of previous generations or the other elements in the same generation. Additionally, we also figure out suitable *pedigrees* or *ancestries*. Assuming there are abundant processors, one processor basically takes care of one pedigree. Finding good pedigrees eliminates unnecessary transactions between processors.

Step 2. *Inter- and intra-generation analysis: recurrence equations into combinations / sequences of skeletons.* After disentangling dependencies we specify how one generation is computed using skeletons. The computation here requires two kinds of value transactions: inter-generational and intra-generational. Inter-generational dependencies appear when an element of one generation requires some other elements of other pedigrees

which have been obtained as in the previous generations; the shift skeleton serves this purpose. Intra-generational dependencies are about the relations of elements in the same generation. No circularity is naturally assumed under DAG, and such computation is often efficiently realized by the scan skeleton.

By the progress of the *computational front line* from one generation to another, a fixed number of iteration, depending on the size of the concerned matrix, produces the resulting matrix distributively kept in processors. Each computational front line is hoped to be executed in parallel. The multiplication of execution cost of one generation and the number of generations, namely iterations, decides the overall computation complexity.

3.2 DAG matrix formalization

It is true the Step 1 in the previous subsection is a nontrivial as well as challenging problem. Yet the Step 2 would be rather mechanically resolved. To help mechanization of skeletal implementations we define a uniform framework of dynamic programming called *DAG matrix formalization*.

Definition 1 With assuming the first index i denoting computational pedigrees and the second j stating the number of steps, namely generations. The size of matrix is assumed to be $\mathrm{N} \times \mathrm{M}$, namely there are M generations of N pedigrees. *DAG matrix formalization* $E_{[i,j]}$is defined as follows.

$$E_{[i,j]} = (E_{[i-h,j]} \odot w_{(i,j)}) \oplus F_{[i,j]} \quad \text{where } h \in \mathbb{N}_+ \qquad \text{— (b)}$$

$$F_{[i,j]} = \bigotimes_{l \in \{1,\ldots,l'\}} \{ \; f_l(E_{[i+g_l\ k,\ j-k]}, i, j, k) \mid k \in \{1, 2, \ldots, t_l\} \text{ where } t_l \leq j, \; g_l \in \mathbb{Q} \; \} \qquad \text{— (a)}$$

Note that f_l which takes nonexistential matrix elements (partly because of involvement of $\mathbb{Q}$ in indexes) in its first argument are assumed to return $\iota_\otimes$. □

It is quite common to employ dynamic programming for optimization problems, and computations appearing in Figure 1 are just min and sum; they can naturally relaxed to other sets of computations. This definition up above consists of two parts: (a) the computation of an auxiliary matrix F denoting inter-generational dependencies using f and $\oplus$, (b) the computation denoting intra-generational dependencies using $\odot$ and $\oplus$. Each of them is explained as follows.

The computation of the part (a) for inter-generational dependencies first needs resolving. This computation produces some intermediate results by referring to the preceding generations, which are possibly of different pedigrees. The gaps of the differences in pedigree numbers are restricted to be linear to the generation gap (using some constant g_l). With this restriction the processor i at the j-th generation can safely receive the required list of values of length t_l from the processor $i + g_l$, which should have received its required list of values from the processor $i+2g_l$ at the $j-1$-th generation.

The part (b) works afterwards, which states intra-generational dependencies, from other pedigrees of the same generation. Yet the distances of skips are again liner (using a constant h), and the scan computation works fine to compute the required values once the ring property holds between $\odot$ and $\oplus$. When we do not need any intra-generational dependencies, defining the computation $(\odot w(i,j))$ to return $\iota_\oplus$ serves this purpose.

3.3 Skeletal implementation of DAG matrix formalization

Now that we have given a framework of dynamic programming, we give its implementation using skeletons.

Theorem 1 Assume the ring property holds between $\odot$ and $\oplus$. $E_{[\_,j]}$ and its auxiliary matrix $F_{[\_,j]}$ defined in Definition 1 can be implemented as follows:

$$E_{[\_,j]} \;=\; \textbf{let } F_{[\_,j]} = \overline{c_{1,1}}\, \Upsilon_\otimes \, (\, \ldots \, \Upsilon_\otimes \, \overline{c_{t_{l'},l'}} \,) \;\; \textbf{in } \oslash /\!\!/^h_{(\iota_\oplus,\iota_\odot)} (F_{[\_,j]} \, \Upsilon \, w_{(\_,j)}) \;,$$

where

$$
\begin{aligned}
idx(G_{[a\le\cdot\le b,j]}) &= [a,\ldots,b]\,\Upsilon\, G_{[a\le\cdot\le b,j]} \\
\overline{c_{l,k}} &= (\ \lambda\ (i,x)\ .\ f_l(x,i,j,k)\ ) \\
&\quad *\ idx(\overset{g_l\ k}{\rightarrow}_{\iota\otimes}\ E_{[\_,j-k]}) \\
(c_l,w_l)\oslash(c_r,w_r) &= (c_l\odot w_l\oplus c_r,\ w_l\odot w_r)\ .
\end{aligned}
$$

□

It is worth mentioning that some skeletal computation can be changed, or simplified, into other constructs when some conditions suffice. To be concrete, there are cases that the computation of $F_{[\_,j]}$ is realized using a single reduce, or that the computation of $E_{[\_,j]}$ is implemented using the map instead of the scan. We will see this in the next Section 4.

Once we have the implementation using skeletons, the analysis of complexity is straightforward.

Theorem 2 We write $\sum_{l\in\{1,\ldots,l'\}} t_l$ as C_F. When the parameterized functions and operators take constant time, computation of matrices specified using DAG matrix formalization takes, using N processors, $\mathrm{O}(C_F\cdot \mathrm{M}\cdot\log \mathrm{N})$ parallel time in case intra-generational dependencies using $\odot$ and $\oplus$ in the part (b) is involved, and $\mathrm{O}(C_F\cdot \mathrm{M})$ parallel time without the dependencies. □

Note that the cost of $\mathrm{O}(C_F)$ becomes $\mathrm{O}(\log \mathrm{M})$ in case the computation of $F_{[\_,j]}$ is realized by the reduce skeleton, and that the cost of $\mathrm{O}(\log \mathrm{N})$ becomes $\mathrm{O}(1)$ when the intra-generational dependencies are described using the map.

4 Application of the Framework

Now that the previous section has clarified the uniform formalization for dynamic programming and its parallelizable implementations using skeletons, we show how the classes shown in Figure 1 are reformatted into the formalization in the following subsections. We also evaluate the overall parallel complexity as its natural consequence.

Note that for the case (0) in Figure 1, we have already analyzed its parallelization, which realizes parallel execution in $\mathrm{O}(\mathrm{N}^3\log \mathrm{M})$ time [12, 13]. We resolve the dependency by transposing the concerned matrix and regard the matrix consists of one single generation in which one pedigree holds N elements. The whole computation in this case is simple enough to be implemented using a single scan for the table of optimal values associated with states.

4.1 Case (2)

This form of dependencies appears in the problems of longest common subsequence or minimum edit distance. Using a matrix of size $\mathrm{N}\times\mathrm{M}$, the sequential complexity is $\mathrm{O}(\mathrm{N}\,\mathrm{M})$.

There are two ways to find the decreasing chain. One is to slice the matrix horizontally, creating M generations; $E_{[i,j]}$ is therefore equal to $D[i,j]$. In this case, $F_{[\_,j]}$ is obtained from two functions f_1 and f_2 whose associated parameters are as follows.

- f_1: $g_1=0$, $t_1=1$, $f_1(x,i,j,\_)=x+n_{i,j}$
- f_2: $g_2=-1$, $t_2=1$, $f_2(x,i,j,\_)=x+nw_{i,j}$

Their results are zipped with $\otimes_{=\downarrow}$. This slicing leaves intra-generational dependencies, and they are resolved by two operators $\odot = +$ and $\oplus_{=\downarrow}$ which form a ring. With an auxiliary function w and the constant $h=1$, scan skeleton finishes the computation of one generation. The parallel complexity is $\mathrm{O}(2\cdot\mathrm{M}\cdot\log\mathrm{N})=\mathrm{O}(\mathrm{M}\ \log\mathrm{N})$. This cost is comparable to the previous work under distributed memory environments [2].

Another is to slice crossly, deriving $E_{[i,j]}=D[j-i,i]$. This uses three functions f_1', f_2' and f_3':

- f_1': $g_1=0$, $t_1=1$, $f_1'(x,i,j,\_)=x+n_{i,j}$,
- f_2': $g_2=-1$, $t_2=1$, $f_2'(x,i,j,\_)=x+w_{i,j}$,
- f_3': $g_3=-1/2$, $t_3=2$, $f_3'(x,i,j,\_)=x+nw_{i,j}$.

With generations of size $\mathrm{N}+\mathrm{M}-2$ this approach eliminates intra-generational dependencies completely. The parallel complexity in this approach is therefore $\mathrm{O}(4\cdot(\mathrm{N}+\mathrm{M}-2))=\mathrm{O}(\mathrm{M}+\mathrm{N})$.

4.2 Case (3)

This form of dependencies appears in the problems of matrix chain product problem, optimal binary search tree or optimal polygon triangulation.

Using a matrix of size $\mathrm{N} \times \mathrm{N}$, the sequential complexity is $\mathrm{O}(\mathrm{N}^3)$.

Since the initial value is given askew $D[i, i+1]$, we slice the matrix crossly to find the decreasing chain. $E_{[i,j]}$ therefore is to imply $D[i, i+j+1]$; the operator $\otimes$ is again $\downarrow$. The question is how we define f_l.

When computation of j-th generation is over, the processor i is to have $E_{[i,\cdot\leq j]}$. Since we do not need any shift operations to refer to $E_{[i,\cdot\leq j]}$, we transform the definition as follows.

$$\begin{aligned} E_{[i,j]} &= \min_{i<r<j+i+1} \\ &\quad \{E_{[i,r-i-1]} + E_{[r,j-r-1]} + w(i,r,j)\} \\ &= \min_{i<r<j+i+1} \\ &\quad \{E_{[r,j-r-1]} + (\ E_{[i,r-i-1]} + w(i,r,j)\ )\} \end{aligned}$$

With assuming nonexistent $E_{[i,j]}$ and $w(i,r,j)$ as $\iota_{\downarrow} = +\infty$, we notice that a single function f_1 is enough.

- f_1: $g_1 = 1$, $t_1 = j$,
  $f_1(x,i,j,r) = x + (\ E_{[i,r-i-1]} + w(i,r,j)\ )$.

Since this computation requires both shifting and computation to be $\mathrm{O}(j)$ for the j-th generation, the overall parallel complexity becomes $\mathrm{O}(\mathrm{N}^2)$.

Note that one processor executes just one pair of data transactions, namely sending and receiving, of size j takes place at the j-th step. This is because the whole information the processor i requires is kept in the neighboring processor $i+1$ after finishing the $j-1$-th step.

4.3 Case (1)

As the last example we consider the case (1) This form of dependencies appears in the problems of least weight subsequence, optimum paragraph formatting, or finding a minimum height B-tree. Using an array of size N, the sequential complexity is $\mathrm{O}(\mathrm{N}^2)$.

Due to simplicity of this problem we can come up with optimization ideas, which makes the problem unfit to the DAG matrix formalization. Still, we consider two approaches along with the formalization using a matrix E of size $\mathrm{N} \times \mathrm{N}$, and in both cases we assume $E_{[i,i]} = D[i]$.

The first approach is to use the part (a), and omit the part (b). We define f_1, with $t_l = j$ and $g_1 = -1$, as $f_1(x,i,j,k) = x + w(i, i-k)$ only when $i = j$ holds, "don't care" otherwise. As is seen in the case (3), this specification requires computation of cost j in the j-th generation, resulting in $\mathrm{O}(\mathrm{N}^2)$. However, the specification given here "doesn't care" $E_{[i,j]}$ of $i \neq j$, and the processor i' just takes care of a single value $E_{[i',i']}$. When we assume $w(i,j)$ is computed in a constant time by the processor j and with the help of associativity in $\otimes = \downarrow$, the following skeletal solution also works.

$$E[i,i] = \downarrow /idx([E_{[0,0]}, \ldots, E_{[i-1,i-1]}])\ \Upsilon_+\ w_{(i,\cdot\leq i-1)}$$

The cost of the reduce in one generation is $\mathrm{O}(\log \mathrm{N})$, and the overall parallel complexity is $\mathrm{O}(\mathrm{N} \log \mathrm{N})$.

The second approach depends on the part (b) and defines the matrix E as follows, using transposition.

$$\begin{aligned} E_{[i,j]} &= \begin{cases} F_{[i,j]} (= D[i,i]) & (i = j) \\ (\ E_{[i,i]} + w(i,j)\ ) \downarrow F_{[i,j]} & (i > j) \end{cases} \\ F_{[i,j]} &= E_{[i,j-1]} \end{aligned}$$

Different from the previous, this approach tries to use $E_{[i,j]}$ of $i \neq j$ more actively. This definition almost follows the part (b), with $\downarrow$ as $\oplus$ and $+$ as $\odot$, except for the left occurrence of $\odot$: it uses $E_{[i,i]}$ of the fixed index i. This simple dependency eliminates the need of scan, and mapping of $E_{[i,i]}$ to each processor $j > i$ works enough. Since C_F is equal to 1 and we do not need the $\mathrm{O}(\log \mathrm{N})$-costing scan, the overall parallel cost is $\mathrm{O}(\mathrm{N})$.

5 Concluding Remarks

In this paper we have analyzed the skeletal implementations of dynamic programming through giving a form called DAG matrix formalization. This form is general enough to cover well-known classes of dynamic programming, and once defined under this form their skeletal implementations as well as their parallel complexity are automatically derived.

This paper also leaves several future works. The first to be mentioned is that the process of generation analysis has been performed by human insight. Mechanization is required for our framework to be widely used. In the theoretical point of view, the

obtained parallel complexity can be enough competing, as the case (2) in Section 4 showed. Yet our current approach falls behind the previous work under CGM/BSP [2] in terms of the number of *supersteps*, which, between *local computations* by each processor independently, perform data transactions among processors and correspond to generations in our formalization. Our approach requires generations linear to the size of the indexes M, while their approach requires supersteps of $O(\log p)$ where p is the number of processors used. Since data transactions are often more expensive than the local computation, it would be preferable to reduce the number of generations. Some more transformation or program derivation need also investigating in this respect.

There is a room for more parallelization in the case (3). When associativity holds in the operator $\otimes$ of inter-generational dependencies, each reduction computation can be executed in parallel. This requires the underlying skeletal parallel libraries to support nested occurrences of skeletons, namely skeletons having a skeleton as their parameter. Realization of this mechanism, aside from enabling users to write much more efficient codes, may affects the approaches to derive parallel programs.

References

[1] C.E.R. Alves, E. Cáceres, and F. Dehne, Parallel dynamic programming for solving the string editing problem on a CGM/BSP. In *Proceedings of ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 275–281, 2002.

[2] C.E.R. Alves, E. Cáceres, and S.W. Song. A BSP/CGM Algorithm for the All-Substrings Longest Common Subsequence Problem. In *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[3] R. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 5–42. Springer-Verlag, 1987.

[4] G.E. Blelloch. Scans as primitive operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.

[5] P.G. Bradford, G.J.E. Rawlins, and G.E. Shannon. Efficient Matrix Chain Ordering in Polylog Time. *SIAM Journal of Computing*, 27(2): 466–490, 1998.

[6] M. Cole. Skeletal Parallelism home page. `http://homepages.inf.ed.ac.uk/mic/Skeletons/`.

[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*, Second edition. MIT Press, 2001.

[8] A. Czumaj. Parallel Algorithm for the Matrix Chain Product and the Optimal Triangulation Problems (Extended Abstract). In *Proceedings of Symposium on Theoretical Aspects of Computer Science (STACS)*, 294–305, 1993.

[9] F. Dehne, ed. Coarse grained parallel algorithms. *Special Issue of Algorithmica*, 24(3/4): 173–176, 1999.

[10] Z. Galil and K. Park. Parallel Algorithms for Dynamic Programming Recurrences with More Than O(1) Dependency. *Journal of Parallel and Distributed Computing*, 21(2): 213–222, 1994

[11] S.-H.S. Huang, H. Liu, and V. Viswanathan. Parallel Dynamic Programming. *IEEE Transactions on Parallel and Distributed Systems*, 5(3): 326–328.

[12] K. Kakehi, Z. Hu, and M. Takeichi. MMPP: Maximum Marking Problems in Parallel. In *Proceedings of the 20th JSSST Conference*, 2003.

[13] K. Kakehi, Z. Hu, and M. Takeichi. List homomorphism with accumulation. In *Proceedings of Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 250–259, Oct. 2003.

[14] K. Matsuzaki, K. Kakehi, H. Iwasaki, Z. Hu, and Y. Akashi. A Fusion-Embedded Skeleton Library. In *Proceedings of European Conference on Parallel Processing (Euro-Par)*, 644–653, 2004.

[15] K. Matsuzaki, Y. Akashi, K. Emoto, H. Iwasaki, and Z. Hu. SkeTo: A Library for Parallel Programming with Constructive Skeletons. In *Proceedings of the 22nd JSSST Conference*, 2005.

[16] D.G. Morales, F. Almeida, F. Garcia, J. Gonzalez, J.L. Roda, and C Rodríguez. A Skeleton for Parallel Dynamic Programming. In *Proceedings of European Conference on Parallel Processing (Euro-Par)*, 877–887, 1999.

[17] W. Rytter. On Efficient Parallel Computations for some Dynamic Programming Problems. *Theoretical Computer Science*, 59: 297–307, 1988.

[18] D.B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12): 38–51, December 1990.

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