Parallelism in Sequential Divide-and-conquer

Zhijing G. Mou, Steve Anderson, and Paul Hudak

Research Report YALEU/DCS/RR683
Febuary 1989

This research was supported by DOE grant DOE FG02-86ER25012.

Abstract

A programming paradigm called sequential divide-and-conquer (SDC) is identified. The structure of SDC algorithms is revealed by showing how they are composed from a small set of parallel constituent functions. SDC's sequentiality and parallelism are studied. It is shown that despite the inherent sequentiality SDC algorithms often can be computed in parallel with significant speedup and nearly optimal efficiency. Three practical problems, Scan, Gaussian Elimination, and Triangle Linear System are used as running examples in the discussion. A parallel programming language called DIVACON is informally introduced so that the structure of SDC algorithms can be naturally reflected in their programs.

Introduction

Divide-and-conquer, as generally described in literature [1],[5], [9], computes a function by dividing the input data structure into two or more substructures, recursively applying the function to each of substructures, combining the results for the substructures to obtain the final result.

Although the recursive application of the computed function over one substructure is independent of the applications over others, the substructures in general need to communicate with each other and perform an internal transformation at certain stages of the computation. The inter-structure reference is intrinsic to all non-trivial divide-and-conquer algorithms, and has crucial impact on the parallel implementation of the algorithms. The conventional description of divide-and-conquer is inadequate as it fails to address this important issue.

The problem was first recognized in [6]. In the formal model for divide-and-conquer proposed there, the *adjust functions*, which perform cross-structure reference, were identified together with divide/combine operations as intrinsic constituents of divide-and-conquer. Communication implied in divide-and-conquer algorithms, as a component of adjust functions, therefore received the separate treatment it deserves, which in turn facilitated the mapping of divide-and-conquer algorithms onto parallel machines and the parallel complexity analysis. As shown in [6], the model yields for a broad class of problems natural (to people) and efficient (on machine) algorithms.

Unfortunately, we have encountered some problems which resist our model in the course of expanding its applications. We realized that those problems share one characteristic: the computation on one substructure depends on the result of the computation of another substructure to such a degree that the computation on the dependent substructure really should wait for the complete computation on the other substructure. The model in [6] cannot deal with these problems well because it insists on applying the divide-and-conquer functions recursively over the substructures *in parallel.*

In this paper, we introduce a model for divide-and-conquer particularly for the problems with the above characteristic. To make a distinction, the model in [6] will from now on be referred to as the *parallel divide-and-conquer* (PDC), and the model presented here will be referred to as *sequential divide-and-conquer* (SDC).

It is generally believed that the parallelism in divide-and-conquer comes mainly from the fact that a function can be recursively applied to more than one substructures in parallel. It is therefore not obvious that SDCs can be computed on parallel machines with any significant speed-up. In fact, we will show that SDCs indeed are totally sequential in some well-defined sense. Nevertheless, we will show why SDC algorithms can be computed on parallel machines with substantial speedup and nearly optimal efficiency. This seems paradoxical and calls for in-depth understanding of sequentiality and parallelism. We therefore have made an attempt at formalizing the concept of parallelism, which has enabled us to explain naturally both the sequential and parallel aspect of SDC algorithms.

Both the SDC and PDC models reveal that divide-and-conquer algorithms can be specified in terms of their constituent functions and regular structures over the constituents. These constituents and the structures are well supported in a language called DIVACON [7] which is being developed on the Connection Machine at Yale. We will give an informal introduction to the language and use it to specify the SDC algorithms.

In section 1, we will informally introduce a subset of DIVACON and review briefly the PDC model for divide-and-coquer. The SDC model is given in section 2 with examples. In section 3, we establish that SDC algorithms are totally sequential with respect to its constituents. In contrast with section 3, we will show in section 4 that SDCs can often be computed in parallel with substantial speedup. Section 5 discusses the role of balancing in SDC algorithms. In the last section, we study the efficiency of SDC algorithms computed on multiprocessors.

1 Preliminary

Notations used in the paper are taken from the language DIVACON [7], which is a functional style parallel language designed to support programming under DC models. In this section, we shall introduce informally the core of the language, which consists of

1. Two compound data types: tuples and arrays.
2. Three highly parallel primitives: divide/combine, local, and communication.
3. One functional form [2], namely, partially ordered function composition (POC); and some important special cases of POC.

and review the PDC model proposed in [6].

Tuples: A n-tuples has the form $\vec{x} = (x_0, \ldots, x_{n-1})$. Two operations are defined over tuples: projection and set-projection defined respectively as:

$$\begin{array}{lll} \text{(projection:)} & .i & (x_0, \ldots, x_i, \ldots, x_{n-1}) = x_i \\ \text{(set-projection:)} & . = (i, y) & (x_0, \ldots, x_i, \ldots, x_{n-1}) = (x_0, \ldots, y, \ldots, x_{n-1}) \end{array}$$

As syntactic sugar, we sometimes use $\vec{x}.i$ to stand for $.1\ \vec{x}$, and $\vec{x}. = (i, c)$ to stand for $. = (i, y)\ \vec{x}$. Note that a binary operator $\oplus$ can be considered naturally as a unary operator over two-tuple or pairs:

$$\oplus(x, y) = x \oplus y$$

For convenience, an integer one-tuple of the form (i) is often identified with i, particulary in the case (i) is used as the a vector index.

Arrays: An array A is a function $A : I \rightarrow V$, where I is its index set, V is its value set. The index set I of a k-dimensional array is a set of k-tuples with the expected continuity and monotonic property. Array indexing is treated as function application, denoted simply by $A\ \vec{i}$. However, indexing is *relative* by default. Therefore if $\vec{i_0}$ is the index of the first entry, the relative and absolute indices of an entry are $\vec{i}$ and $\vec{i'}$ respectively, then $\vec{i} = \vec{i'} - \vec{i_0}$, where '-' here is tuple's element-wise subtraction. *Backward relative indexing* is also allowed and denoted by $A\ -(\vec{i})$. For example, if $A = [1\ 2\ 3\ 4]$ then $A(-0) = 4$.

Like all other functions, an array $A : I \rightarrow V$ can be considered as a set of argument-value paris called its *entries* of the form $(\vec{i}, v)$, where $\vec{i} \in I$, and $v \in V$. We encourage this 'set of pairs' view of arrays since arrays are always distributed in our computation model. It is important to remember an entry $(\vec{i}, v)$ has both its *index value* $\vec{i}$ and *value*

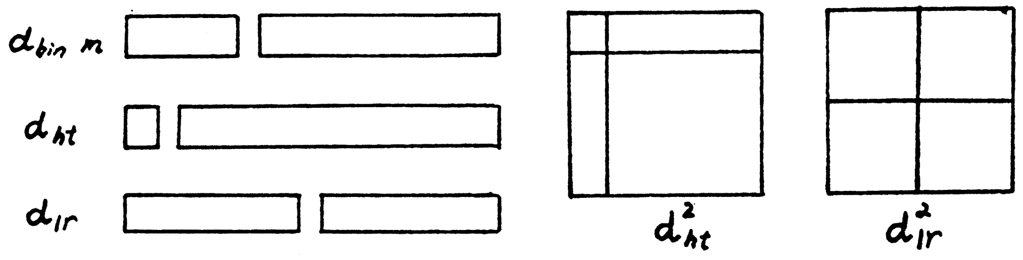


Figure 1: Common Divide Functions

value although sometimes value value is referred to as value. A predicate p over array entries by default is to be applied to the value values, but by adding a prefix '%' this default can be changed. For example if %(> 2) is applied in parallel to all entries of [1 2 3 4], we get [false, false, false, true].

The shape of a k dimensional array is a k-tuple defining the arrays size along each dimension. The special function $ can be used to obtain the shape (any entry of) an array. For example, $.0 and $.1 will return the number of rows and columns of a matrix. A special predicate *atom?* is defined to return *true* for arrays with size one along all dimensions.

Divide and Combine Functions: The binary divide function d_{bin} m divides a vector (one dimensional array) into two subvectors v_0 and v_1, where v_0 consists of the first m entries of v, and v_1 consists of the rest. Two important cases are $m = 1$ and m =v/2(*recall* returns the size of a vector), which we give names of d_{ht} and d_{lr} respectively. The subscripts *ht* and *lr* stand for 'head-tail' and 'left-right' respectively.

A k dimensional array can be naturally regarded as a one dimensional array where each entry is a $(k-1)$ dimensional array. The function d_{bin} therefore can be used to divide higher dimensional arrays if we specify a particular dimension. This particular dimension can be specified by supplying d_{bin} m an additional argument. For example, d_{lr} 0 divides a matrix along the first (row) dimension, and d_{lr} 1 divides a matrix along the second (column) dimension (Figure 1).

Operations over divide functions can be used to define new divide functions from more basic divide functions. One such operation is the intersection operation denoted by $\wedge$ [1]. Particularly, $(d_1\ i) \wedge (d_2\ j)$ partitions an array along the i-th dimension by d_1 and the j-th dimension by d_2 at the same time. This allows us to define

$$d_{ht}^2 = (d_{lr}\ 0) \wedge (d_{lr}\ 1)$$
$$d_{lr}^2 = (d_{ht}\ 0) \wedge (d_{ht}\ 1)$$

which when applied to a matrix will divide it into four submatrices, in a respectively unbalanced and balanced fashion (Figure 1).

Combine functions in this paper are treated simply as the left-inverse [3] of the divide functions. For a divide function d, d^{-1} is used to denote the corresponding combine function. Vector catenation operation is a left-inverse for both d_{ht} and d_{lr}, and we give it the name *cat* since it is used many times.

[1]It is so named because it is essentially the intersection over partitions in modern algebra [3].

Local Functions: Let $f : X \rightarrow Y$ be a function, A an array of type $I \rightarrow X$, then $!f$ (read as "bang f") is a function that maps A to an array B of type $I \rightarrow Y$ by applying in parallel f to the contents of each entry. For example $!(+1)[1\ 2\ 3] = [2\ 3\ 4]$.

When the operator ! is given an integer argument in the form of $!.i$, then a function is defined over the 'super-entries' along the i-th dimension. For example, if (*reduce* +) is a function mapping a vector to a single value by summing up the entries, then $!.0$ (*reduce* +) will map a matrix into a vector. We will also allow ! to be applied to tuples; in particular, if f is a function defined over arrays, (A_0, A_1) are arrays, then $!f\ (A_0, A_1) = (f\ A_0, f\ A_1)$.

Note that in the context of divide-and-conquer, the term 'local' refers to functions that contain no inter-structure communication[6]. It is *strongly local* if it contains no inter-entry communication, otherwise *weakly local.* Thus $!f$ applied to an array is always strongly local, while $(!.i\ f)$ for array, and $!f$ for tuples of arrays are generally not.

Communication Functions: Communication from a set A to a set B can be generally modeled by a binary relation R from A to B such that for $a \in A$ and $b \in B$, $(a, b) \in R$ iff a receives a message from b. We will however restrict ourselves to the communications whose relations happen to be functions. Therefore, a communication from A to B can be specified by a function $f : A \rightarrow B$.

Similarly, given two arrays $A : I \rightarrow X$, and $B : J \rightarrow Y$, a function $f : I \rightarrow J$ fully specifies a communication from A to B. That is, each entry $(\vec{i}, x)$ of A will receive the value of an entry in B with index $(f\ \vec{i})$. For practical reasons, we would also hope that the value x of the entry would not be lost after the communication. Therefore we define for f, A, B:

$$!!f(B\ A) = A' \text{ where } A'\ \vec{i} = (A\ \vec{i},\ B : f\ \vec{i})$$

Observe that each entry is mapped to a pair where its old value is saved as the first element, and the value acquired from the communication is the second element.

For example, if corr = id is the identity function, then

$$!!\text{corr } ([1\ 2],\ [3\ 4]) = [(3,\ 1)\ (4,\ 2)]$$

Communications defined above is always one-directional. For bidirectional communication we define:

$$!!(f_a, f_b)(A, B) = (!!f_a(A, B),\ !!f_b(B, A))$$

For example,

$$!!(\text{corr},\text{corr})\ ([1\ 2],\ [3\ 4]) = ([(1,3)(2,4)],\ [(3,1)(2,4)])$$

The same method can be used to specify communication from an array to itself. When !!f takes only one array as argument, we define

$$!!\text{f A} = !!(\text{A},\text{A})$$

For example, if f (i, j) = (i, 0), !!f applied to a matrix M will broadcast M(i, 0) to all entries in the ith row for all rows.

Functions like f, corr above will be referred to as *communication generators.* Surprisingly, a very broad class of functions over arrays seem to share a very small set of communication generators in their DC algorithms. Many of the generators are also so simple that they are not dimension sensitive, and applicable to arrays of different dimensions:

corr $\vec{i} = \vec{i}$	@ correspondent communication
mirr $\vec{i} = -\vec{i}$	@ mirror-image communication
br $\vec{c}\,\vec{i} = \vec{c}$	@ broadcast from one entry to all
last $\vec{c}\,\vec{i} = -\vec{c}$	@ broadcast from a entry with backwards index
vm-row-br $\vec{i} = \vec{i}.0$	@ from vector to matrix br row-wise
vm-col-br $\vec{i} = \vec{i}.1$	@ from vector to matrix br col-wise
row-br $c\,\vec{i} = (.=\ 0\ c)\vec{i}$	@ matrix br row-wise
col-br $c\,\vec{i} = (.=\ 1\ c)\vec{i}$	@ matrix br col-wise

Note that !!corr(A,B) will make each entry in A communicate with an entry with the same index in B, therefore achieves the ***correspondence*** communication. !!mirr(A, B) will make each entry in A communicate with an entry with the same but backward index in B, thereby achieving the ***mirr-image*** communication. Both corr and mirr are ***one-one generators***, and any entry needs to send at most once during communications they define. All other functions are ***broadcast generators*** as some entry (entries) need(s) to send value(s) for unbounded number of times.

In DC algorithms, a communication function in general is followed by a local function. Communication always maps arrays to arrays of pairs, local function are often defined to map arrays of pairs to arrays. Some combinations of communication functions and local functions occur so frequently it is worth giving them special names. In particular, from a binary operator $\oplus$ over two types, we define $[\oplus]$ to be the binary operator over two arrays of the type by:

$$[\oplus] = !\oplus : !!\text{corr}$$

which performs 'entry-wise' operation over two arrays of any dimension. For example,

$$[+]([1\ 2],[3\ 4]) = !(+)[(3,1)(4,2)] = [4\ 6]$$

Functional Forms: [2]: Functional forms are used to combine several functions together to form a new function, therefore allowing complex function to be hierarchically constructed from basic functions. Function composition denoted usually by '$\circ$' is one of the best understood function combining forms. Its application to programming specification is however limited for the imposed linear ordering over the component functions.

Partially ordered composition (POC) is simply a generalization of conventional function composition. Relations over component functions are allowed to be any partial order relation [3] in POC rather than only total order relation [3]. It is well-known that a partial order relation corresponds to a directed acyclic graph (DAG) and vice versa. We therefore can define a POC by defining its DAG. Formally, a POC G always has the form:

$$\begin{aligned}
&G\ (v_0, ..., v_{n-1}) :: (x_0, ..., x_{p-1}) = (y_0, ..., y_{q-1}) \\
&\quad \text{where } (y_{0,0}, ..., y_{0,q}) = v_0(x_{0,0}, ..., x_{0,p}) \\
&\qquad \dots \\
&\qquad (y_{(n-1),0}, ..., y_{(n-1),q}) = v_{(n-1)}(x_{(n-1),0}, ..., x_{(n-1),p})
\end{aligned}$$

[2]The term 'functional form' is from [2].

where $v_0, ..., v_{n-1}$ are poc's n nodes, $x_0, ..., x_{p-1}$ are poc's p in edges, $y_0, ..., y_{q-1}$ are poc's q out edges. The n equations in the where clause specify for each node its adjacent list, with the in edges on the right-hand-side, and out edges on the left-hand-side.

A POC G like above can be instantiated by supplying it with n functions. The call $f = G(f_0, ..., f_{n-1})$ therefore will match f_i to the node v_i, f_i's jth input to v_i's jth in edge, f_i's kth output to v_i's kth out edge. The result f is a function that takes p inputs and returns q outputs. Functions $f_0, ..., f_{n-1}$ are called f's *components.* Notice that there is no reason why f cannot act as a component in a POC, particularly, we can define a function recursively by using itself as a component.

There are some special POC forms worth being identified. One of them of course is the conventional linear function composition, which we will denote by ':'. Another is the *function tuple form,* defined as $!!(f_0 ... f_k)(x_0, x_k) = (f_0\ x_0, ... f_1\ x_k >$.

Conditionals are given by '$p \rightarrow f; g$' with the usual meaning, $p \rightarrow f$ is treated as $p \rightarrow f; id$.

Filters denoted by '$\Rightarrow$' work with functions that map an array a to an isomorphic array a'. If p is a predicate over the index sets, then $!p \Rightarrow f\ a = a''$, where $a''\ \vec{i} = a'\ \vec{i}$ if $p\ \vec{i}$, else $a''\ \vec{i} = a'\ \vec{i}$. The essential difference between conditional and filter should be observed. For example,

$$!(<2? \rightarrow +1)[4\ 3\ 2\ 1] = [4\ 3\ 3\ 2]$$
$$<2? \Rightarrow !(+1)[4\ 3\ 2\ 1] = [5\ 4\ 2\ 1]$$

Parallel Divide-and-conquer: Parallel divide-and-conquer algorithms in [6] all have the following general form:

$$f = p \rightarrow f_b; (c : h : !f : g : d)$$

where f's *constituents* d, c, g, h, p, f_b are respectively divide, combine, preadjust, postadjust, base case predicate, base function. [3]

A higher order functional DC was given in [6] to specify a parallel divide-and-conquer algorithm directly from its constituents. Observe that the 'else' part in the following definition is a POC with the graph given in Figure 2.

$$DC(d, c, g, h, p, f_b) = p \rightarrow f_b; (c : h :\ (!) f : g : d)$$

The well-known scan function over vectors ($scanv = v',\ where\ v'(i) = \sum_{k \leq i} v(k)$) was given as PDC in [6]:

$$\begin{aligned} &\text{scan} + \ = \text{DC}\ (d_{lr}, \text{cat, id, h, atom?, id}) \\ &\quad \text{where}\ \ h + \ = \ (!\text{id}, !(+)) : !!(\text{nil}, (\textit{last}\ 0)) \end{aligned}$$

Example (Scan PDC): Consider (scan +) [1 2 3 4]. The vector is divided by d_{lr} into ([1 2], [3 4]); Recursive parallel application of scan will give us ([1 3], [3 7]); The communication function !!(nil, (*last 0*)) maps them to ([1 3], [(3,3) (7,3)]. The left vector is not changed since its communication is !!nil. By applying !id and !(+) respectively to the left and right, we get ([1 3],[6 10]); The final result is given by their catenation [1 3 6 10].

[3] In [6], p was built into d, and here we have made it an explicit constituent.

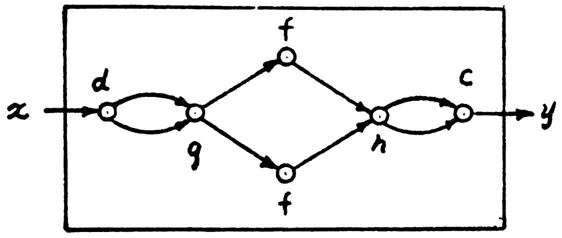
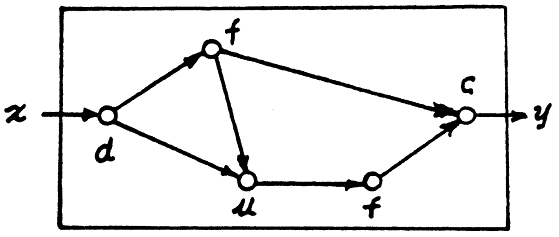


Figure 2: POC used in PDCs



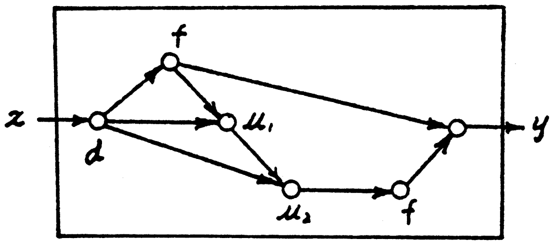


Figure 3: POCs used in Sequential Divide-and-conquer

2 Sequential Divide-and-conquer Algorithms

A function $f : X \rightarrow Y$ is a SDC, if there exist the following *constituent* functions:

Divide :	$d : X \rightarrow X \times X$ is a divide function in X;
Combine:	$c : Y \times Y \rightarrow Y$ is a combine function in Y;
Adjust :	$\mu : X \times Y \rightarrow X$ is the (cross-)adjust function;
Base Predicate:	$p : X \rightarrow$ *Boolean* is the base predicate;
Base Function:	$f_b : X' \rightarrow Y$, where $X'(\subset X) = \{x \mid p\ x, x \in X\}$

such that

$$f = \text{SDC1 } (d, c, \mu, p, f_b)$$

where the functional SDC1 and a POC G_1 used in the functional are:

```
SDC1 $(d, c, \mu, p, f_b) = f_{rec}$
    where $f_{rec} = p \rightarrow f_b; G1(d, c, \mu, f_{rec}, f_{rec})$
G1 $(d, c, \mu, f, f) :: x = y$
    where $y = c\ (y_0, y_1)$
          $y_0 = f\ x_0$
          $y_1 = f\ x'_1$
          $x'_1 = \mu\ (y_0, x_1)$
          $(x_0, x_1) = d\ x$
```

The POC G_1 used in above SDC is drawn on the left of Figure 3. By definition SDC can be computed by the following general procedure:

Algorithm 2.0: General SDC Algorithm
Input: Constituents d, c, μ, p, f_b of SDC $f : X \rightarrow Y,\ x \in X$
Output: $y = f\ x,\ (y \in Y)$

```
0. base: if $p\ x$ then return $f_b\ x$; else:
1. divide: $d\ x = x_0, x_1$;
2. recursion 1: $f\ x_0 = y_0$;
3. adjust: $\mu(x_0, x_1) = x_1'$
4. recursion 2: $f x_1' = y_1$
5. combine: return $c(y_0, y_1)$   □
```

We can immediately see some of the important differences between SDCs and PDCs:

(1) They use different POCs (Compare Figure 2 with Figure 3).

(2) Recursive application of f to the two sub-structures can be done in parallel in PDC, and not in SDC.

(3) In PDC an adjust function maps two structures from X (preadjust) or Y (postadjust) to two structures in the same domain, while in SDC, an adjust function maps two structures, one from Y, one from X, and map them to a structure in X.

Nevertheless, SDC and PDC share many features. Most obvious of all:

(1) They are both computed by dividing the argument, computing the function recursively over the (adjusted) sub-argument, and combining the subresults to get the final result.

(2) They share the same set of constituent functions, namely the divide, combine, adjust, base predicate and base functions. All but the adjust functions have exactly the same functionality in both PDC and SDC.

Scan, for example, can be computed not only by PDC but also SDC. In the following SDC algorithms, a vector is equally divided into the left and right subvector, scan over the left is first computed, the adjust function then adjusts the value of the right according to the result of the left, scan then is computed over the adjusted right. Final result is the vector catenation of the two subresults.

Algorithm 2.1: Scan (over a vector).
Input: A vector v (size of power of two).
Output: $v' =$ scan v, where $v'(i) = \sum_{k \leq i} v(k)$
Method: (Balanced) SDC.
Program:

```
scan + = SDC1 $(d_{lr}, cat, \mu_{scan}, atom?, id)$
   where $\mu_{scan}$ + = $(=0?) \Rightarrow (!(+) :!!(last\ 0))$   □
```

Note that adjust function μ_{scan} takes two vectors, and maps the second one to a new one by adding to each entry the last entry of the first vector. However, because of the filter '=0?', this 'fetch and add' operation take effect only on the first entry of the second vector.

Example(Scan SDC): Consider (Scan +) [1 2 3 4]. The vector is divided into [1 2], [3 4]; scan [1 2] = [1 3]; μ ([1 3], [3 4]) gives [(3+3) 4] = [6 4], and scan [6 4] = [6 10]. The final result is thus

[1 3 6 10].

Another example of an SDC algorithm is Gaussian Elimination.

Algorithm 2.2: Gaussian Elimination.
Input: A matrix m of size $n * (n+1)$ representing a linear system with n unknowns.
Output: An upper triangle system m' represented the same way.
Method: By (unbalanced) SDC.
Program:

```
    $ge = SDC1((d_{ht}0), (d_{ht}0)^{-}1, \mu_{ge}, p, id)$
        where $\mu_{ge}$ = F ⇒ ($!loc_{ge}$ : !!(row-br k) : !!(col-br 0))
            $loc_{ge}((x,y),(u,v)) = x * v/u - y$
            $p = (= 1? : \$.0)$          @ base predicate: one row matrix?
            $F = (> (k, \%.1))$          @ filter: column index greater than k?
            $k = (\$.1 - \$.0)$          @ k: the column being eliminated. □
```

Recall that (d_{ht} 0) is the unbalanced division over array along the first dimension. It would divide a matrix M into two sub-matrices: (M_h, M_t) where M_h consists of the first row of M, M_t the rest. Since M_h satisfies the base predicate $p = (= 1? : \$.0)$, which tests if the matrix has only one row (recall \$.0 returns the size along dimension 0 for any array), the base function is applied. Since the base function is identity function, the Gaussian Elimination of M_h is simply itself.

The adjust function is then applied to (M_h, M_t). It is the linear composition of three functions: (1) the inter-array communication !!(col-br 0) will broadcast M_h's jth entry to M_t's jth column; (2)the intra-array communication !!(row-br k) broadcasts each entry in M_t's jth column to the row to which the entry belongs, where k is the difference between M_t's column and row dimensions; (3) the local function loc$_{ge}$ is then applied, which is defined over a pair of pairs since each communication preceded produced a pair. It should be verified that M_t's kth column will be eliminated after the adjust function is applied. Filter is used in the adjust function to make the already eliminated entries inactive (recall %.1 returns the entries column index for matrices).

Example (Gaussian Elimination Unbalanced SDC): Let the system be represented by matrix M:

$$\begin{bmatrix} 3 & 2 & 1 & 1 & 7 \\ 6 & 1 & 2 & 2 & 11 \\ 3 & 4 & 3 & 2 & 12 \\ 1 & 3 & 5 & 4 & 13 \end{bmatrix}$$

The matrix M is divided into two submatrices: the first row M_0 and the rest M_1. M_0 satisfies the base predicate, and is mapped to itself by the base function id. The inter-array communication !!(col-br 0) is applied to M_0 and M_1, the result is M_1':

$$\begin{bmatrix} (6,3) & (1,2) & (2,1) & (2,1) & (11,7) \\ (3,3) & (4,2) & (3,1) & (2,1) & (12,7) \\ (1,3) & (3,2) & (5,1) & (4,1) & (13,7) \end{bmatrix}$$

The intra-array communication !!(row-br 0) is then applied to M_1', we get:

$$\begin{bmatrix} ((6,3),(6,3)) & ((1,2),(6,3)) & ((2,1),(6,3)) & ((2,1),(6,3)) & ((11,7),(6,3)) \\ ((3,3),(3,3)) & ((4,2),((3,3)) & ((3,1),(3,3)) & ((2,1),(3,3)) & ((12,7),(3,3)) \\ ((1,3),(1,3)) & ((3,2),(1,3)) & ((5,1),(1,3)) & ((4,1),(1,3)) & ((13,7),(1,3)) \end{bmatrix}$$

Local function loc_{ge} is then applied:

$$\begin{bmatrix} 6*3/6-3 & 1*3/6-2 & 2*3/6-1 & 2*3/6-1 & 11*3/6-7 \\ 3*3/3-3 & 4*3/3-2 & 3*3/3-1 & 2*3/3-1 & 12*3/3-7 \\ 1*3/1-3 & 3*3/1-2 & 5*3/1-1 & 4*3/1-1 & 13*3/1-7 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & -3/2 & 0 & 0 & -3/2 \\ 0 & 2 & 2 & 1 & 5 \\ 0 & 7 & 14 & 11 & 32 \end{bmatrix}$$

Notice that the first column is eliminated. This matrix is to be recurisively eliminated, the result after the recursive elimination will be combined with M_0 to form the final result.

When a structure $x \in X$ is divided into several substructures, it is possible that some of the substructures are no longer elements of X. For instance, a right triangle can be partitioned into two right triangles and a rectangle, but the rectangle is not in the domain of right triangles. We will call sub-structures of this type *odd sub-structures*, and divide functions that yield odd sub-structures will be called *odd divide functions.*

To deal with DC algorithms with odd division, we define a variant of SDC. A function $f : X \rightarrow Y$ is an (odd) SDC if:

$f = \text{SDC2 } (d, c, \mu_1, \mu_2, p, f_b)$	
where $d : X \rightarrow X \times Z \times X$	Divide function
$\mu_1 : Y \times Z \rightarrow Z$	Adjust function one
$\mu_2 : Z \times X \rightarrow X$	Adjust function two
$c, p, f_b :$	same as in SDC1

$\text{SDC2 } (d, c, \mu_1, \mu_2, p, f_b) = f_{sdc}$
  where $f_{sdc} = p \rightarrow f_b; G_2(d, c, \mu_1, \mu_2, f_{sdc}, f_{sdc})$
$\text{G2 } (d, c, \mu_1, \mu_2, f, f) :: x = y$
  where $y = c\ (y_0, y_1)$
    $y_0 = f\ x_0$
    $y_1 = f\ x'_1$
    $z' = \mu_1\ (y_0, z)$
    $x'_1 = \mu_2\ (z', x_1)$
    $y_1 = f\ x'_1$
    $(x_0, x_1) = d\ x$

Observe that the function f is not to be applied recursively to an odd structure since it is not in f's domain. However, an odd structure is not to be simply thrown out in DC algorithms. As dictated by the POC G_2 (see Figure 3 right), the second (real) sub-structure is adjusted through the odd structure by the two adjust functions successively after the first sub-structure has been computed. A general algorithm computing odd SDCs can be derived by modifying Algorithm 2.0. An example odd SDC problem is the linear triangle system.

Algorithm 2.3: Lower Triangle System (TR).
Input: The pair (M, V), where M is a $n * n$ lower triangle matrix, V is a vector of size n.
Output: A vector C such that $M \times C = V$.
Method: (Unbalanced odd) SDC.

Program:

$$\begin{aligned}
&\text{tr} = \text{SDC2}\ (\ d_{tr}, cat, \mu_1, \mu_2, size\_one?, tr_b)\\
&d_{tr} = \text{dtrpoc}\ (d_{ht}^2, d_{ht})\\
&\mu_1 = !(*) : !!(\text{last}\ (0,0))\\
&\mu_2\ (Z, (M, V)) = (M, [-]\ (V, Z))\\
&size\_one?\ (m,v) = \text{atom?}\ v\\
&tr_b = !(/) : !!\text{corr} \quad \square
\end{aligned}$$

Observe that the divide function is defined in turn by POC dtrpoc, which is defined below. The same POC will be used in Section 5 to define the divide function in balanced SDC algorithm for triangle system.

$$\begin{aligned}
&\text{dtrpoc}\ (d_m, d_v) :: (M,V) = (M_0,V_0), Z, (M_1,V_1)\\
&\quad \text{where } (M_0, nothing, Z, M_1) = d_m\ M\\
&\qquad\quad (V_0,V_1) = d_v\ V
\end{aligned}$$

The divide function divides a triangle system of size n into two smaller triangle systems of size one and (n-1) respectively and an odd structure Z, which is the first column of the triangle matrix. The base function is trivially defined over systems of size one. Adjust function μ_1 broadcast the solution to the odd column, a local multiplication is performed to produce a new column. This new column is subtracted from the vector of the second subsystem by μ_2. The SDC is then recursively applied to the second subsystem.

Example (Triangle System Unbalanced SDC): Let the system (M, V) be:

$$\begin{bmatrix} 3 & & & \\ 2 & 1 & & \\ 4 & 2 & 1 & \\ 1 & 2 & 3 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \\ 11 \\ 18 \end{bmatrix}$$

After the division, it is divided into a system of size one (M_0, V_0), a matrix corresponds to the first column of M without the first element Z, and a system of size three (M_1, V_1):

$$(M_0, V_0) = [\ 3\ ][\ 3\ ], \quad Z = \begin{bmatrix} 2 \\ 4 \\ 1 \end{bmatrix}, \quad (M_1, V_1) = \begin{bmatrix} 1 & & \\ 2 & 1 & \\ 2 & 3 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ 11 \\ 18 \end{bmatrix}$$

The system (M_0, V_1) satisties the base predicate, and by applying the base function, we get a solution vector S = [1]. The adjust function μ_1 is applied to (S, Z) in two steps:

$$(S, Z) \Rightarrow_{!!(\text{last}(0,0))} \begin{bmatrix} (2,1) \\ (4,1) \\ (1,1) \end{bmatrix} \Rightarrow_{!(*)} \begin{bmatrix} 2 \\ 4 \\ 1 \end{bmatrix}$$

Let the result of μ_1 be Z', the adjust function μ_2 is applied to $(Z', (M_1, V_1))$, which simply subtracts Z' from V_1. The adjusted sub-system therefore becomes:

$$\begin{bmatrix} 1 & & \\ 2 & 1 & \\ 2 & 3 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 7 \\ 17 \end{bmatrix}$$

which can be rescursively solved to get the solution of [2 3 4], the final solution is therefore [1 2 3 4].

5 Balancing in SDCs

Let us illustrate the role of balancing in SDC algorithms through examples.

The SDC algorithm for the Triangle System given previously used unbalanced division. The balanced version for the same problems is given below. A system of size n is now divided into two systems of size $n/2$, and a size $n/2$ square matrix. The algorithm proceeds by solving the first subsystem, multiplying the result vector by the square matrix, subtracting the result of the multiplication (a vector) from the column vector of the second subsystem, and then solving the (adjusted) second subsystem. It is noticeable that the combine function *cat*, adjust function μ_2, base function f_b, and the POC used to construct the divide function are all shared by both the balanced and unbalanced versions.

Algorithm 5.1 Balanced SDC Algorithms for Triangle System.
Input: A lower triangular matrix M, a vector v of size n.
Output: a vector x such that $M * x = v$.
Method: Balanced odd SDC.
Program:

tr = SDC2 ((dtrpoc (d_{lr}^2, d_{lr}), cat, μ_1, μ_2,size_one?, tr_b)
μ_1 = (!.0 (reduce +)) : !(*) : !!vm-col-br
μ_2 , size_one?, tr_b, dtrpoc: same as in Algorithm 2.3 □

Recall that (!.0 *reduce* +)) reduces a matrix to a column by applying (*reduce* +) [6] to all rows of the matrix.

Example (Triangle System Balanced SDC): Let the system (M, V) be the same as in the example for Algorithm 3.2. After the division, we get two triangle sub-systems of equal sizes (M_0, V_0) and (M_1, V_1), and a square matix Z:

$$(M_0, V_0) = \begin{bmatrix} 3 & \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 4 \end{bmatrix}, \quad Z = \begin{bmatrix} 4 & 2 \\ 1 & 2 \end{bmatrix}, \quad (M_1, V_1) = \begin{bmatrix} 1 & \\ 3 & 1 \end{bmatrix} \begin{bmatrix} 11 \\ 18 \end{bmatrix}$$

The system (M_0, V_1) can be recursively solved to obtain the solution vector $S = [1\ 2]$. The adjust function μ_1 is applied to (S, Z) in three steps:

$$(S, Z) \Rightarrow_{\text{!!vm-col-br}} \begin{bmatrix} (4,1) & (2,2) \\ (1,1) & (2,2) \end{bmatrix} \Rightarrow_{\text{!(*)}} \begin{bmatrix} 4 & 4 \\ 1 & 4 \end{bmatrix} \Rightarrow_{\text{!.0(reduce+)}} \begin{bmatrix} 8 \\ 5 \end{bmatrix}$$

The function μ_2 then adjusts the second sub-system (M_1, V_1) by subtracting the above vector from V_1. The system (M_1, V_1) after adjustment becomes:

$$\begin{bmatrix} 1 & \\ 3 & 1 \end{bmatrix} \begin{bmatrix} 3 \\ 13 \end{bmatrix}$$

which has the solution of [3 4]. The final result is therefore [1 2 3 4].

Now let us check if we have gained anything from balancing. Recall that time used by balanced (binary) SDC algorithm is given by:

$$T_f(n) = 2 * T_f(n/2) + T_d(n) + T_c(n) + T_\mu(n)$$

Since the divide and combine functions in Algorithm 5.1 are valid, they take $O(1)$ time. It is easy to verify that both adjust functions μ_1 and μ_2 involve only strong local functions, correspondent and broadcasting (in reduce) communications, and therefore take $O(log\ n)$ time. Therefore, time used by Algorithms 5.1 is given by:

$$T_{tr}(n) = 2 * T_{tr}(n/2) + O(log\ n) = O(n)$$

Compared with the $O(n * log\ n)$ time complexity for the unbalanced algorithms, the balancing has given us an additional $O(log\ n)$ speedup!

In the unbalanced SDC algorithm for Gauss Elimination each time one column is eliminated when the adjust function is applied. The algorithm can be modified to use balanced division, and an adjust function that must also modified to eliminate more than one column. However, one will find that the overall time complexity of the balanced version is not better than the unbalanced one. The following proposition helps to explain why balancing will contribute to speedup in some but not all situations.

Proposition 5.1 *Let f_b, f_u be respectively balanced and unbalanced SDC algorithms, $core_b$ and $core_u$ be respectively the time complexity of their core constituents. Then $T(f_b) = o(T(f_u))$ if $T(\text{core}_b) = O(T(core_u))$.*

The complexity for core constituents for Triangle System algorithms are the same in both balanced and unbalance version. While for Gaussian Elimination, the adjust function in the balanced version is of higher order complexity than that in the unbalanced version. That is why balancing contributed to speedup for the former and not for the latter.

6 Consumed Resources and Efficiency

Efficiency of a parallel algorithm is as important as its speedup if not more so, and is typically defined as the ratio between speedup and number of processors used [10]. We would, however, prefer to study the efficiency through the concept of ***processor-time resource consumption*** (PT). By which we mean the discrete integration of number of processors over the time used to implement a parallel algorithms (Figure 7). If the number of processors does not change over time, the resource consumption reduces to the conventional concept of ***time-processor product*** [10]. Given a parallel algorithm, the ratio of PT consumptions in the cases of using one processor and n processors will, as can be verified, give us the efficiency.

Due to the regularity of DC algorithms, the PT value of DC algorithms can actually be given by recurrence equations. The general form of such recurences for balanced and unbalanced SDCs can be shown as:

$$\begin{aligned} PT_b(n) &= 2 * PT_b(n/2) + PT_{dcm} \\ PT_u(n) &= PT_u(n-m) + PT_u(m) + PT_{dcm} \end{aligned}$$

where the PT_{dcm} stands for the PT resource consumed by divide, combine, and adjust functions.