Compress-and-Conquer for Optimal Multicore Computing

Z. George Mou
TIGMA, LLC
gmou5813@gmail.com

Hai Liu Paul Hudak
Yale University
{hai.liu,paul.hudak}@yale.edu

Abstract

We propose a programming paradigm called *compress-and-conquer* (CC) that leads to optimal performance on multicore platforms. Given a multicore system of p cores and a problem of size n, the problem is first reduced to p smaller problems, each of which can be solved independently of the others (the *compression* phase). From the solutions to the p problems, a compressed version of the same problem of size $O(p)$ is deduced and solved (the *global* phase). The solution to the original problem is then derived from the solution to the compressed problem together with the solutions of the smaller problems (the *expansion* phase).

The CC paradigm reduces the complexity of multicore programming by allowing the best-known sequential algorithm for a problem to be used in each of the three phases. In this paper we apply the CC paradigm to a range of problems including scan, nested scan, difference equations, banded linear systems, and linear tridiagonal systems. The performance of CC programs is analyzed, and their optimality and linear speedup are proven. Characteristics of the problem space subject to CC are formally examined, and we show that its computational power subsumes that of scan, nested scan, and mapReduce.

The CC paradigm has been implemented in Haskell as a modular, higher-order function, whose constituent functions can be shared by seemingly unrelated problems. This function is compiled into low-level Haskell threads that run on a multicore machine, and performance benchmarks confirm the theoretical analysis.

Categories and Subject Descriptors D.1.3 [*Parallel Programming*]

General Terms Algorithms, Languages, Theory.

Keywords Multicore Programming, Parallel Computing, Programming Paradigm, Functional Programming, Scan, Divide and Conquer, Compress and Conquer

1. Introduction

Parallel programs often introduce certain overheads, such as interprocessor communication, synchronization, and so on. Sometimes these overheads even occur at the algorithmic level. In particular, the total number of operations performed by a parallel algorithm is often greater than that for the best sequential algorithm. We believe that, unlike massively parallel computers, the number of processing units on a multicore system is best considered as a constant, independent of the problem size. It follows that the amount of computation on each core should be of the same order as the complexity of the original problem. Therefore, efficient sequential computation within each core is as crucial as the parallel execution of the program by all cores, in terms of overall performance.

The question we ask, then, is whether we can take advantage of the best-known sequential algorithms in multicore computing. A positive answer to the question might not only lead to efficient multicore computation, but also to a reduction in the complexity of multicore programming, in that a new algorithm does not need to be found if a multicore algorithm can be easily derived from the sequential one.

In this paper, we introduce a new programming paradigm that we call *compress-and-conquer* (CC). Given a multicore system of p cores and a problem of size n, the problem is first reduced to p smaller problems, each of which can be solved independently of the others (the *compression* phase). From the solutions to the p problems, a compressed version of the same problem of size $O(p)$ is deduced and solved (the *global* phase). The solution to the original problem is then derived from the solution to the compressed problem together with the solutions of the smaller problems (the *expansion* phase).

Although this idea sounds simple enough, we have found it fruitful to formalize, analyze, carefully implement, and finally apply the method to a number of non-trivial applications. In particular, our contributions include:

- A description of CC as a high-level algorithmic abstraction (or skeleton) for multicore computing, that demonstrates how a multicore algorithm can be derived from a sequential one.
- A proof of the optimality and linear speedup of CC programs.
- A Haskell library that captures CC as a modular higher-order function. This allows a multicore algorithm to be specified in terms of a small set of constituent functions, many of which can be shared amongst different programs, thus enhancing modularity and promoting code resue.
- An algorithm for mapping a CC abstraction to a multicore platform, as well as a monadic implementation of this algorithm in Haskell, including the use of mutable arrays.
- Identification of the class of problems subject to the CC abstraction, and an understanding of its limitations, along with a proof that CC subsumes multicore programming models based on scan or mapReduce.
- Application of the CC paradigm to several problems including scan, nested scan, second order difference equations, banded linear systems, tridiagonal systems, and mapReduce,
- Benchmarks of CC programs for some of the above problems that validate well the theoretical performance results.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAMP'10, January 19, 2010, Madrid, Spain.
Copyright © 2010 ACM 978-1-60558-859-9/10/01…$10.00

- A critical comparison of CC to divide-and-conquer, and observations for future work, including the use of a nested form of CC that can be mapped onto hierarchical multicore systems.

The paper is organized as follows. We introduce the notion of CC in Section **??**. CC algorithms expressed in Haskell are derived in Section **??** for problems including scan, nested scan, second-order linear difference equations, Fibonacci sequence, banded linear systems, tridiagonal linear systems, and mapReduce. In Section **??** we show how CC programs can be compiled for execution on multicore systems; in particular, how logical data dependencies are mapped to inter-core communications. In Section **??** we give an analysis and proof for the optimality of CC in terms of operation count, communication, and scalability. The benchmarks of some CC programs on multicore systems are also presented. In Section **??** we identify the class of problems subject to the paradigm, and its relation to the computational complexity hierarchy. Some variants of CC are given in Section **??**. The relation of CC to divide-and-conquer and related work are discussed in Sections **??** and **??**, respectively.

2. The Paradigm

We represent a collection over values of type *a* as an abstract data type *S a*, which can be anything like an array, a list, a tree, a set, etc. Given a function f_s :: *S a* → *S b*, we define the compress-and-conquer (CC) of function f as a higher order function as follows:

DEFINITION 2.1. *The algorithm of compress-and-conquer (CC)*

```
cc :: (∀ a . S a →[S a]) →        – divide
      (∀ a . [S a] →S a) →        – combine
      (S b →S c) →                – compress
      ((S d, S a) →S a) →         – expand
      (S c →S a) →                – pre-communication
      (S b →S d) →                – post-communication
      (S a →S b) →                – sequential function
      S a →S b
cc d c co xp com_g com_h f_s s =
  let seg  = d s
      pre  = map (co . f_s) seg
      core = (d . com_h . f_s . com_g . c) pre
      post = map (f_s . xp) (zip core seg)
  in  c post
```

The computation defined by the CC function can be broken clearly into into three-phases, which we will refer to as *compression*, *global*, and *expansion* phases respectively.

1. Compression phase *map (co* . f_s*) . d* : The input is first divided by *d* into a number of segments, and function f_s is applied in parallel to each segment, with no inter-dependencies. The results are then compressed by function *co* at each segment. Note that in Def. **??** we name the divided segments as *seg*, which is preserved and later retrieved in the expansion phase.
2. Global phase *d* . com_h . f_s . com_g . *c* : The compressed segments from the compression phase are first combined by *c* to become a single collection before passed to the pre-communication function com_g. This is followed by an application of the function f_s, and then a post-communication of com_h. The result is again divided into segments, ready to be distributed back.
3. Expansion phase *c* . *map (*f_s . *xp) . zip* : The results from the global phase are first zipped with the original input segments, and then expanded by function *xp*. Function f_s is applied again to each segment with no inter-dependencies, and the results are finally combined into one collection.

A schematic illustration of the CC paradigm is given in Fig. **??**. We will refer to the divide, combine, compress, expand, pre- and post-communication, and the sequential function f_s as the *constituents* of compress-and-conquer, They are further explained below:

1. Function *d* :: ∀ *a* . *S a* → [*S a*] divides the given collection into a number of disjoint segments, and the combine function *c* :: ∀ *a* . [*S a*] → *S a* is its inverse with the property *c* . *d* = *id*. They both are given a polymorphic rank-2 type because we want the division to be independent of the actual values in the collection. For example, list concatenation is polymorphic, the merge in merge-sort and the division in quick-sort are non-polymorphic.
2. Function *co* :: *S b* → *S c* compresses the result after f_s is applied to the input segments before passing them to the global phase. We say that a compress function *co* is *bounded* if there exists a constant k, such that for any s, $|s|/|co\ s| \leq k$, where $|s|$ is the size of collection s. A compress function that is not bounded is *unbounded*. For example, a function that maps any set to a singleton set is an unbounded compress function, which compresses a set of any size to one of size one. In contrast, the compression of a vector that returns all the entries with even indices is bounded, and has a compression ratio of two.
3. The expand function *xp* :: (*S d, S a*) → *S a* takes the results of type *S d* from the global phase, expand them by modifying the segments from the original input of type *S a*, before passing to the f_s function in the final phase.
4. In the global phase, before f_s works on the compressed data, they are pre-processed by function com_g :: *S c* → *S a*; then the output from f_s are post-processed by function com_h :: *S b* → *S d*. They are called the pre- and post-communication functions because they represent data dependency between the segments.

Further more we give the following definitions to the properties of a CC algorithm:

- The *arity* of a CC function is the arity of its divide and combine constituents.
- The *compression ratio* of a CC function is the compression ratio of its compression constituent.
- A CC function has an *unbounded compression ratio* if its compression constituent is unbounded.
- A CC function is *self-similar* if the CC of f_s defines the same function, i.e. *cc d c co xp* com_g com_h f_s = f_s, for some *co, xp,* com_g, com_h, and for any *d* and *c*.

As shall be seen in the later sections, functions defined with the above *cc* forms can be mapped to multicore systems and often lead to algorithms with optimal speedups. The CC higher order form provides a way to specify a multicore algorithm with often very simple constituent functions.

3. Case Studies

A broad range of problems can be solved by compress-and-conquer. In this section, we will examine its application to a number of common problems. Because these problems all deal with ordered sequences, without loss of generality, we'll use the list type as a concrete representation for *S a*:

type S a = [*a*]

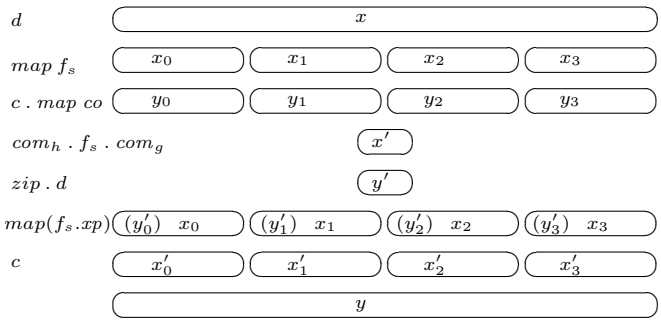

Figure 1. A schematic illustration of a compress-and-conquer algorithm to compute $f\ x = y$ where $f = cc\ d\ c\ co\ xp\ com_g\ com_h\ f_s$ with division arity 4 . The first level oval box represents the input data x, and the last level one the output y. The constituent functions to be applied to each level is labeled on the left-side of each level.

It is important to note that programs written using the list representation are not meant to be efficient implementations, but rather specifications with sufficient detail to guide real implementations over multi-cores that will be discussed in Sec **??**.

We also define as few commonly used constituent functions as follows:

```
d :: Int → S a → [S a]
d p l | p == 1      = [l]
      | otherwise = let (m, n) = splitAt (length l `div` p) l
                    in m : d (p - 1) n
c :: Int → [S a] → S a
c p = concat

first, last, last2, bothend :: S a → S a
first     l = take 1 l
first2    l = take 2 l
last      l = drop (length l - 1) l
last2     l = drop (length l - 2) l
bothend   l = first l ++ last l

sr :: a → S a → S a
sr i l = i : take (length l - 1) l
```

Function *d* divides the given sequence into *p* equal-size segments, and *c* is its inverse. Functions *first,first2,last,last2,bothend* are simple constituent functions that extract the first, frst two, last, last two, or both first and last elements from a sequence. Function *sr* shifts the given sequence one position to the right, and fills in the first element with its argument.

3.1 Scan

Scan or prefix has been considered a powerful parallel and multi-core programming constructs. Here is a formal definition:

DEFINITION 3.1. *A scan or prefix operation is defined to be a function that maps an input sequence $x_0, x_1, \ldots, x_{n-1}$ with respect to an associative binary operator $\oplus$ to an output of:*

$$x_0, x_0 \oplus x_1, \ldots, x_0 \oplus x_1 \oplus \cdots \oplus x_{n-1}$$

In Haskell, a function called *scanl1* from the *P*relude already does exactly this computation. So we'll just define our sequential scan as:

```
scan = scanl1
```

We next show a CC algorithm for scan by providing its simple constituents.

ALGORITHM 3.1. *Scan with respect to an associative binary operator $\oplus$ by Compress-and-Conquer:*

```
ccScan ⊕ = cc (d p) (c p) last addfirst id (sr 0) (scan ⊕)
  where addfirst ([v], (x : xs)) = v ⊕ x : xs
```

Informally, the *cc* higher order function takes seven of its constituents, and returns a function that computes the scan with respect to the binary associative operator $\oplus$. It does so by first dividing the input sequence into p segments, and applying the scan over each segment, all segments in parallel. The last elements of the segmented scan are then used to derive a compressed sequence of the size p. Scan is then performed over the compressed sequence of size p. The post communication shifts the global result to right by one position so that the ith result is distributed back to the $(i+1)$th segments, and added to the first element in the original segment by the expand function *addfirst*. A scan is then performed again in parallel to all the segments. All segments are then concatenated to form the final solution (See Figure **??**).

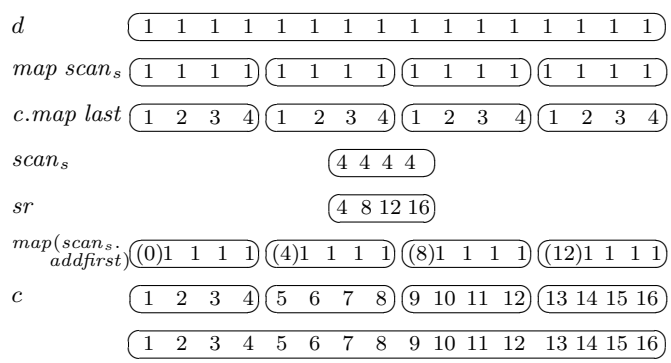

Figure 2. Scan by CC with a sequence of 16 1's, and p = 4.

3.2 Nested Scan

A nested scan is to apply scan to a list of sequences. More formally, we have:

DEFINITION 3.2. *A nested scan with respect to an associate binary operator $\oplus$ is defined in terms of scan (see Def.* **??***):*

```
nestedScan ⊕ = map (scan ⊕)
```

The solution to nested scan is usually a mapping to the flat scan without involving nested parallelism. We first convert the list of sequences to a flat sequence of pairs with the following function:

```
flat :: [S a] →S (a, Bool)
flat l = zip (concat l) [n == length v | v ← l, n ← [1..length v]]
```

Intuitively, the second component of the pairs marks if the original element was the last element in the nested sequence. For example:

$$flat\ [[1,2,3],[4,5],[6]] = [(1,\circ),(2,\circ),(3,\bullet),(4,\circ),(5,\bullet),(6,\bullet)]$$

where $\bullet$ = *True*, and $\circ$ = *False*. We also define the inverse of *flat* and a lifting function as follows:

```
unflat :: S (a,Bool) → [S a]
unflat l | null l      = []
         | otherwise = let (m, (v : n)) = break snd l
                       in map fst (m ++ [v]) : unflat n

lift :: (a → a → a) → ((a, Bool) → (a, Bool) → (a, Bool))
lift f (x, u) (y, v) = (if u then y else f x y, v)
```

It can be easily verified that if $\oplus$ is associative over type *a*, then *lift* $\oplus$ is associative over type (*a, Bool*).

ALGORITHM 3.2. *Nested scan with respect to an associative binary operator $\oplus$ can be reduced to a flat scan over pairs by*

ccNestedScan $\oplus$ = *unflat* . *ccScan* (*lift* $\oplus$) . *flat*

3.3 Second Order Linear Difference Equations

In this section, we consider the system of second order linear difference equations of the following form:

DEFINITION 3.3. *System of Second Order Linear Difference Equations:*

$$\begin{aligned} y_0 &= c_0 \\ y_1 &= c_1 \\ y_2 &= a_2 y_0 + b_2 y_1 + c_2 \\ &\vdots \\ y_{n-1} &= a_{n-1} y_{n-3} + b_{n-1} y_{n-2} + c_{n-1} \end{aligned} \tag{1}$$

Now let us consider a section of (**??**) corresponding to the variables indexed from p to q, $p < q < n$, denoted by $L[p, q]$:

$$\begin{aligned} y_p &= a_p \quad y_{p-2} + b_p \quad y_{p-1} + c_p \\ y_{p+1} &= a_{p+1} y_{p-1} + b_{p+1} y_p \quad + c_{p+1} \\ y_{p+2} &= a_{p+2} y_p \quad + b_{p+2} y_{p+1} + c_{p+2} \\ &\vdots \\ y_q &= a_q \quad y_{q-2} + b_q \quad y_{q-1} + c_q \end{aligned} \tag{2}$$

In a system of difference equations, we say a variable y_i *depends* on another variable y_j if y_j appears as a term on the right hand side of its equation; and two variables are *aligned* if they depend on the same variables. We next will align all the variables from y_p to y_q, so that they all depend on the external variables y_{p-2} and y_{p-1}. This can be achieved with the following sequential algorithm:

ALGORITHM 3.3. *(diff) A Sequential Internal Solver for a Section of Second Order Difference Equation Given a section $L[p, q]$, where only the first two variables may have external references, and all the other variables refer to variables internal to the section. Let $X = (y_{p-2}, y_{p-1}, 1)$, we define a new sequence of vectors u_i such that $y_i = u_i * X$, where $*$ stands for a point-wise multiplication for vectors.*

$$\begin{aligned} y_p &= u_p * X \\ &= a_p \quad y_{p-2} + b_p \quad y_{p-1} + c_p \\ &= (a_p, b_p, c_p) * X \\ y_{p+1} &= u_{p+1} * X \\ &= a_{p+1} y_{p-1} + b_{p+1} y_p \quad + c_{p+1} \\ &= (a_p b_{p+1}, a_{p+1} + b_p b_{p+1}, c_{p+1} + c_p b_{p+1}) * X \\ &\vdots \\ y_q &= u_q * X \\ &= \left(\begin{bmatrix} u_{q-2} \\ u_{q-1} \\ 0\ 0\ 1 \end{bmatrix} (a_q, b_q, c_q) \right) * X \end{aligned}$$

In Haskell, we write the internal solver as a function mapping from the sequence of (a_i, b_i, c_i) to the sequence of u_i as follows:

```
diff ((a0, b0, c0) : (a1, b1, c1) : xs) = u
  where u0 = (a0, b0, c0)
        u1 = (a0 * b1, a1 + b0 * b1, c1 + c0 * b1)
        u  = u0 : u1 : zipWith3 f u (tail u) xs
        f x y z = (x, y, (0, 0, 1)) ⋉ z
```

where $\ltimes$ is defined to be the operation of multiplying a 3x3 matrix with a vector of size 3.

The above gives a definition of vector sequence u_i, for $p \le i \le q$, and we have successfully aligned all variables from y_p to y_q to the external variables represented by $X = (y_{p-2}, y_{p-1}, 1)$.

Note that *diff* can also be used to solve a complete system of 2nd order linear difference equations where $a_0 = b_0 = a_1 = b_1 = 0$. It doesn't matter how we initialize the two variables in X, *diff* will always return a sequence of $u_i = (0, 0, y_i)$. In this sense, Algo. **??** is an algorithm for a generalized form of second order linear difference equations.

Now consider a system L of n second order difference equations partitioned into p sections. By applying Algo. **??** to each section, we can make all the internal variables of each section align to the last two variables of the previous section. Let L' be a system of equations formed by taking the last two equations from each section, then it is not hard to see, with a little adjustment, what we get is in turn a closed second order difference equation, with a smaller size of $2p$. We call L' a compressed version of L.

The adjustment needed here is to make the last variable from each section, except the first section, instead of aligning with the last two variables from the previous section, align with the last from the previous, and second last from its own section. This is achieved with the following function:

```
adjustdiff (x : x' : xs) = x : x' : aux xs
  where aux [] = []
        aux ((a, b, c) : (a', b', c') : xs) =
            (a, b, c) : (a'', b'', c'') : aux xs
            where a'' = b' - b'' * b
                  b'' = if a == 0 then 0 else a' / a
                  c'' = c' - b'' * c
```

Furthermore, solving L' means we have solved the last two variables of each section, therefore the first two variables of the next section can in turn be solved. We'll design an expansion function to properly re-initialize the first two variables in each section, so that they becomes individually solvable by Algo. **??**.

```
initfirst2 ([(_, _, x), (_, _, x')], (u0 : u1 : xs)) =
  (0, 0, y0) : (0, 0, y1) : xs
  where y0 = (x, x', 1) * u0
        y1 = (x', y0, 1) * u1
```

This lead to the following compress-and-conquer algorithm:

ALGORITHM 3.4. (*ccDiff*) Compress-and-Conquer for Second Order Linear Difference Equations

```
ccDiff = cc (d p) (c p) last2 initfirst2 adjustdiff (sr2 (0, 0, 0)) diff
  where sr2 v = sr v . sr v
```

Observe that Fibonacci sequence is no more than a homogeneous second order difference equations with $c_i = 0$ for $0 \le i \le n$ in Eq. **??**, therefore can be solved by CC.

ALGORITHM 3.5. *(ccFib) Since Fibonacci Sequence is no more than a special case of second order linear difference equations, Algo.* **??** *applies.*

3.4 Banded Lower Triangular Linear Systems

DEFINITION 3.4. *A banded lower triangular linear system with bandwidth of two is:*

$$\begin{bmatrix} \dot{a}_0 & & & & \\ \dot{a}_1 & \dot{b}_1 & & & \\ \dot{a}_2 & \dot{b}_2 & \dot{c}_2 & & \\ & \dot{a}_3 & \dot{b}_3 & \dot{c}_3 & \\ & & \ddots & \ddots & \ddots \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ \vdots \end{bmatrix}$$

By multiplying out the matrix and the vector of unknowns, and some simple algebraic transformation, the above banded linear system becomes a second order difference equation in the form of (**??**), where

$$\begin{aligned} y_0 &= d_0/\dot{a} \\ y_1 &= (d_1 - d_0)/\dot{b}_1 \\ y_2 &= -(\dot{b}_2/\dot{c}_2)y_1 - (\dot{a}_2/\dot{c}_2)y_0 + d_2 \\ &\vdots \end{aligned} \tag{3}$$

In other words, a banded linear system is equivalent to a difference equation where the bandwidth equal of the banded system is equal to the order of the difference equations. Algo. **??** therefore is also a compress-and-conquer algorithm for banded linear systems of bandwidth two.

ALGORITHM 3.6. *Banded Triangular Linear Systems with Bandwidth of Two*

*Convert the system to a second order difference equations by (**??**), and then apply Algo.* **??**.

In fact, Algo. **??** can be easily generalized to linear difference equations of kth order, for arbitrary k, and therefore Algo. **??** can also be generalized to solved triangular linear systems with arbitrary bandwidth of k. We choose however to omit the details of the generalization from this paper.

3.5 Tridiagonal Linear Systems

In all the previous case studies, the inter-dependencies between variables are one directional in that if we lay the variables from left to right by their indices, then the dependencies are all from right to left. Tridiagonal linear systems are examples of applications where the dependencies are bi-directional.

The following is a general form for tridiagonal linear system L with n unknowns:

DEFINITION 3.5. *Tridiagonal Linear Systems*

$$\begin{bmatrix} b_0 & c_0 & & & & \\ a_1 & b_1 & c_0 & & & \\ & a_2 & b_2 & c_2 & & \\ & & a_3 & b_3 & c_3 & \\ & & & \ddots & \ddots & \ddots \\ & & & & a_{n-1} & b_{n-1} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-1} \end{bmatrix}$$

Note that for a given variable y_i the coefficients a_i and c_i represent its dependency on y_{i-1}, and y_{i+1} respectively in the above standard form. The coefficient a_i and c_i are referred to as the forward and backward dependency coefficients, respectively.

Now let us consider a section $L[p, q]$ of the tridiagonal system consists of the rows corresponds to variables y_p to y_q, where $0 \leq p < q \leq n$.

$$\begin{bmatrix} a_p & b_p & c_p & & \\ & a_{p+1} & b_{p+1} & c_{p+1} & \\ & & \ddots & \ddots & \ddots \\ & & a_q & b_q & c_q \end{bmatrix} \begin{bmatrix} y_p \\ y_{p+1} \\ \vdots \\ y_q \end{bmatrix} = \begin{bmatrix} d_p \\ d_{p+1} \\ \vdots \\ d_q \end{bmatrix}$$

A variable y_i is said to be *forward (backward) aligned* with y_j if they are forward (backward) dependent on the same variables. They are said to be *aligned* if they are both forward and backward aligned. Hence, no two variables are aligned in the above diagram.

Variable can be aligned by Gaussian elimination. For example, The variable y_{p+1} can be forward aligned with y_p by multiply the row for y_p by $-a_{p+1}/b_p$, and then add it to the row for y_{p+1}. We repeat this process for every row except the row for y_p in $L[p, q]$, which leads to the following diagram:

$$\begin{bmatrix} a'_p & b'_p & c'_p & & \\ a'_{p+1} & & b'_{p+1} & c'_{p+1} & \\ \vdots & & & \ddots & \ddots \\ a'_q & & & b'_q & c'_q \end{bmatrix} \begin{bmatrix} y_p \\ y_{p+1} \\ \vdots \\ y_q \end{bmatrix} = \begin{bmatrix} d'_p \\ d'_{p+1} \\ \vdots \\ d'_q \end{bmatrix}$$

Now variables y_{p+1} to y_q are forward aligned with y_p, which means that the coefficients of $a'_p, a'_{p+1}, \ldots, a'_p$ are in the same column in the matrix. We can write the forward alignment as a function that takes a sequence of (a_i, b_i, c_i, d_i) and returns the modified co-efficients (a'_i, b'_i, c'_i, d'_i) as follows:

```
forward [] = []
forward (x : xs) = u
  where u = norm x : zipWith f xs u
        f (a, b, c, d) (a', b', c', d') =
          norm (-a' * a, b - c' * a, c, d - d' * a)
norm (a, b, c, d) = (a / b, 1, c / b, d / b)
```

Note that in the process we also normalize every row so that the co-efficients on the diagonal of the matrix (all the b_i) become 1.

With the forward alignment in place, we can use a similar process to backward align variable y_{q-1} with y_q, and so on, which leads to the following diagram:

$$\begin{bmatrix} a''_p & b''_p & & & c''_p \\ a''_{p+1} & & b''_{p+1} & & c''_{p+1} \\ \vdots & & & \ddots & \vdots \\ a''_{q-1} & & & b''_{q-1} & c''_{q-1} \\ a''_q & & & b''_q & c''_q \end{bmatrix} \begin{bmatrix} y_p \\ y_{p+1} \\ \vdots \\ y_{q-1} \\ y_q \end{bmatrix} = \begin{bmatrix} d''_p \\ d''_{p+1} \\ \vdots \\ d''_{q-1} \\ d''_q \end{bmatrix}$$

Note that all variables y_p to y_q are now both forward and backward aligned. We can write the backward alignment function in a similar manner as follows:

```
backward [] = []
backward u = reverse v
  where (x : xs) = reverse u
        v = x : zipWith f xs v
        f (a, b, c, d) (a', b', c', d') =
          (a - a' * c, b, - c' * c, d - d' * c)
```

We consider a tridiagonal system is solved if only diagonal coefficients are left in the matrix. Obviously, if $a_p = c_q = 0$, the system $L[p, q]$ is completely solved after forward and backward alignments. When a_p or c_q are not zeros, however, we shall only align the inner block $L[p+1, q+1]$, and adjust the boundary rows for y_p and y_q to align inward like this:

```
adjust l = let [(a0, b0, c0, d0), (a1, b1, c1, d1)] = first2 l
               [(a2, b2, c2, d2), (a3, b3, c3, d3)] = last2 l
           in [(a0, b0 - a1 * c0, -c1 * c0, d0 - d1 * c0)] ++
              middle l ++
              [(-a2 * a3, b3 - c2 * a3, c3, d3 - d2 * a3)]
```

As as result from this adjustment, we effectively obtain a diagram of the following shape for $L[p, q]$ when $a_p \neq 0$ or $c_q \neq 0$:

$$\begin{bmatrix} a''_p & b''_p & & & c''_p & \\ & a''_{p+1} & b''_{p+1} & & c''_{p+1} & \\ & \vdots & & \ddots & \vdots & \\ & a''_{q-1} & & b''_{q-1} & c''_{q-1} & \\ & a''_q & & & b''_q & c''_q \end{bmatrix} \begin{bmatrix} y_p \\ y_{p+1} \\ \vdots \\ y_{q-1} \\ y_q \end{bmatrix} = \begin{bmatrix} d''_p \\ d''_{p+1} \\ \vdots \\ d''_{q-1} \\ d''_q \end{bmatrix}$$

ALGORITHM 3.7. *(trid) A Sequential Internal Solver for a section of tridiagonal linear systems is a composition of the forward and backward alignment, and the adjustment function:*

```
trid [] = []
trid l = case (a, c) of
  (0, 0) → backward (forward l)
  _      → adjust ([x] ++ backward (forward (middle l)) ++ [y])
  where [x@(a,_,_,_),y@(_,_,c,_)] = bothend l
```

Now if we divide a tridiagonal system into p sections, and apply Algo. **??** to each section, they are then all internally solved. In Figure **??**, we show the non-zero coefficients in the matrix after internally solving all sections for an example case where $n = 16$ and $p = 4$.

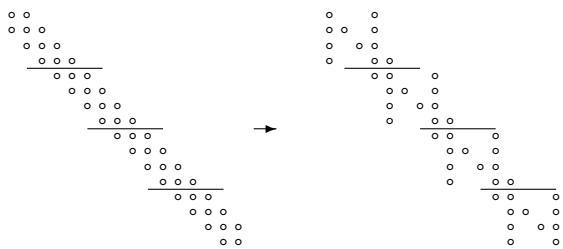

Figure 3. A tridiagonal linear system of size n (=16) divided into p (=4) sections, each section internally solved.

By focusing on the first and last variables in all sections after the internal solver *trid* is applied, one realizes that they in turn form a compressed tridiagonal system of size $2p$. This compressed system can in turn be solved by *trid*. The solution of the compressed tridiagonal system can be plugged back to each section, and each section can then be completely solved independently. This leads to the following compress-and-conquer algorithm for tridiagonal linear systems:

ALGORITHM 3.8. (*ccTrid*) Compress-and-Conquer Algorithm for Tridiagonal Linear Systems

```
ccTrid = cc (d p) (c p) bothend replace id id trid
  where replace ([x, y], l) = [x] ++ middle l ++ [y]
```

3.6 MapReduce

DEFINITION 3.6. *MapReduce is the functional composition of the map and reduce:*

$$mapReduce\ f \oplus = reduce \oplus\ .\ map\ f$$

where reduce with respect to an associative binary operator $\oplus$ *is a function that maps a non-empty sequence* $x_0, x_1, \ldots, x_{n-1}$ *to a single value of* $x_0 \oplus x_1 \oplus \cdots \oplus x_{n-1}$.

J. Dean and S. Ghemawat introduced *mapReduce* in [**?**] as a separate programming construct, gave distributed implementations, and showed it applies to many search engine problems.

The problem of *mapReduce* can be said to be an inherently simpler problem than any of the problems we have considered so far and can be computed by a compress-and-conquer where the post-phase is not needed. We therefore introduce a new and simpler version of compress-and-conquer, which we call *pre-CC*, for it contains only the pre-phase of the more general CC form as in Def. **??**:

$$cc_{pre}\ d\ c\ co\ f = co\ .\ c\ .\ map\ (co\ .\ f)\ .\ d$$

We then have the following simple algorithm for the parallel version of mapReduce:

ALGORITHM 3.9. *MapReduce with respect to an associative binary operator* $\oplus$ *and a function* f *is defined in terms of pre-CC:*

$$ccMapReduce\ f \oplus = cc_{pre}\ (d\ p)\ (c\ p)\ (reduce \oplus)\ (map\ f)$$

4. Implementation

4.1 Operational Mapping

Implementation of compress-and-conquer algorithms on multicore systems is fairly straightforward. The work done in compression and expansion phases can be easily mapped on to p threads or processors in parallel. We can certainly use a more compact representation than lists, but more fundamentally, the specification of CC as given in Def. **??** is inefficient on today's dominant CPU architectures due to the immutability implied by referential transparency, which prevents destructive updates. Also, the divide and combine functions should just share the original input data instead of making new copies of them, and the order and the arity of the divide function needs to be consistent with combine.

For the above reasons, we move over to a monadic form of compress-and-conquer defined below:

DEFINITION 4.1. *The implementation of monadic compress-and-conquer* (cc_m)*:*

```
cc_m :: Monad m ⇒
    (∀a. ([S a] → m())
        → S a → m(S a)) →          – divide then combine
    (∀a. (S a → m())
        → [S a] → m[S a]) →        – combine then divide
    (S a → m(S b)) →               – compress
    ((S c, S a) → m(S a)) →        – expand
    (S b → m(S a)) →               – pre-core
    (S a → m(S c)) →               – post-core
    (S a → m(S a)) →               – sequential
    S a → m(S a)
cc_m dc cd co xp g h f_s = dc aux
  where aux seg = do
        pre  ← parmap (co · f_s · dup) seg
        core ← cd (h · f_s · g) pre
        parmap (f_s · xp) (zip core seg)
      (f · g) x = g x >>= f
```

In this program, we intentionally define a composition operator $(\cdot)$ to be the monadic counterpart of function composition so that our implementation of cc_m closely match the specification of CC in Def. **??**. Further explanations are given below:

1. In order to do destructive updates, we must now make the sequential function f_s return the same collection type as its input. This affects the overall types of cc_m and its constituent functions.
2. We pair up the divide and combine functions as either a single divide-then-combine or combine-then-divide operation. Both are now higher order functions that take as argument a function that can update the original data in place, but can not change the structure of them.
3. Because the original CC algorithm requires the input collection to remain unchanged until the expansion phase, we must use $dup :: S\ a \rightarrow m(S\ a)$ to create a local copy of the segment during the compression phase.
4. The original *map* function is changed to a monadic *parmap* $:: (a \rightarrow m\ b) \rightarrow [a] \rightarrow m[b]$ that spawns off a system thread for each segment, and only returns when all threads are done.

In our actual implementation, we choose to define the concrete collection type as an unboxed mutable array as follows in order to minimize computation overhead:

data S a = Arr (IOUArray Int a)	– shared array
Int	– lower bound
Int	– upper bound

This definition leads to straightforward implementation of both *cd* and *dc* by sharing the original array without creating duplicate copies. All the constituent functions used in the specification of our algorithms must also be modified to operate on arrays, with direct indices and destructive updates. We omit such details here.

For all the example applications of CC, their sequential algorithms must now change to a monadic version that works on the above *S a* type, while the CC versions of these algorithms require little change except moving from cc to cc_m.

Among the different parallelism facilities GHC provides, the lightweight thread library becomes a natural choice because the IO monad implements destructive update. So in other words, the monad m in Def. **??** is just *IO*, and *parmap* is implemented using *forkIO*. We choose the division parameter p to match the number of cores in the hardware so that the original array is split into p segments, and as a consequence, *parmap* will spawn exactly p system threads. We rely on the operating system to balance system threads among multiple cores.

4.2 Inter-Core Communications

With the mapping of CC algorithms to multicore systems given in Section **??**, and if we assume the divided segments reside locally to each processor, we can see that there are two, and only two, constituent functions in a CC algorithm that involve inter-core communications: the results from co at the end the compression phase are moved over to the global phase, and after the global phase, the results are moved back to each processor as input to the expand function. All the rest constituent functions are mapped to local operations. Note that the constituent functions com_g and com_h are referred to as communication functions, not because they are to be mapped into inter-core communications at the implementation level, but rather they realize the dependency relations between different sections in the logic domain.

Let $S = (P_0, \ldots, P_{p-1})$ be a multicore system with p cores used by a CC algorithm, and, without loss of generality, P_0 be the appointed core for the global computation, then by the mapping of $parmap$ from Section **??**, one can see that

- At the end of the compression phase, each P_i ,for $0 < i < p$, sends one piece of data to P_0.
- At the beginning of the expansion phase, each P_i receives a piece of data from P_0.

If we go beyond a Haskell implementation, in the *Message Passing Interface* (MPI) [**?**], there are two supported communication patterns, *gather* and *scatter*, that perform precisely the above two operations respectively. It is therefore straightforward to support the communication in CC algorithms with MPI. Other options, including MP, PThreads, Intel's Thread Building Blocks [**?**], and Microsoft's Parallel Task Library [**?**], can all be used instead.

5. Performance Analysis

Since parallel programs generally incur some overhead over the best known sequential counterparts for the same problems, it is a good practice to understand and quantify the overhead asymptotically. In this section, we show that the overhead of CC algorithms in both operational and communicational aspects are minimum, which also translates to linear speedups on multicore systems.

5.1 Operational Optimality

Given a program P, its *operational complexity*, written $\psi(P)$, is the total number of operations that P performs, as a function of the problem size. We say two programs P_1 and P_2 are *consistent* with each other, written $P_1 \sim P_2$, in operational complexity if and only if $\psi(P_1) = \Theta(\psi(P_2))$ [1].

THEOREM 5.1. *Let f be a CC program with base function f_s (see Def.* **??***), then $f \sim f_s$. In other words, a CC program is consistent with its base function.*
Proof: besides the sequential base function, all other constituents in the CC program takes time independent of problem size.

Given a problem f, its *operational complexity*, written $\phi(f)$, is the minimum number of operations f inherently requires, as a function of the problem size. We say a program F that solves problem f is *operationally optimal* for f, written $F \propto_o f$ if and only if $\psi(F) = O(\phi(f))$.

It follows from the above definition and Theorem **??** that

THEOREM 5.2. *Given a problem f, F a CC program that solves f, and f_s the sequential base function for F, then $F \propto_o f$ if and only if $f_s \propto_o f$.*

The above theorem gives a convenient way to check on the operational optimality of CC programs. From which one can easily verify that

THEOREM 5.3. *The CC algorithms Algo* **??** *for scan, Algo.* **??** *for nested scan, Algo.* **??** *for second order difference equations, Algo.* **??** *for Fibonacci sequence, Algo.* **??** *for banded linear systems, Algo,* **??** *for tridiagonal linear systems are operationally optimal.*

5.2 Communicational Optimality

Given a multicore program P, its *communicational complexity*, written $\delta(P)$, is the total number of inter-core communications that P performs, as a function of the number of cores p. We say two programs P_1 and P_2 are *consistent in communication*, written $P_1 \approx P_2$, in communication complexity if and only if $\delta(P_1) = \Theta(\delta(P_2))$.

Given a problem f over input X partitioned into p disjoint and non-empty subsets of X, we say its *communication complexity*, written $\gamma(f)$, is the minimum number of references crossing the partitions that f inherently requires, as a function of the number of partitions m. We say a multicore program F solving problem f is *communication optimal* for f, written F $\propto_c f$ if and only if $\delta(F) = O(\gamma(f))$.

THEOREM 5.4. *The CC algorithms Algo* **??** *for scan, Algo.* **??** *for nested scan, Algo.* **??** *for second order difference equations, Algo.* **??** *for Fibonacci sequence, Algo.* **??** *for banded linear systems, Algo,* **??** *for tridiagonal linear systems are all communication optimal.*
Proof: Let f be any of the above problems, X the input for f. When X is partitioned into any p disjoint and non-empty blocks. Since the final solution of f over X depends on at least one piece of data in each of the m blocks, the communication complexity of f, $\gamma(f(p)) = \Omega(p)$ [2]. But the CC algorithm for f has communication complexity $\delta f(p) = O(p)$. Therefore, the CC algorithm for f is communication optimal.

[1] $f = \Theta(g)$ if and only if $f = O(g)$ and $g = O(f)$

[2] given f and g, f is said to be at least of the order of g, written $f = \Omega(g)$, if $g = O(f)$

5.3 Linear Speedups

Let f be a program, $T(f, n, p)$ the time to carry out f on input size n and p cores. Then *speedup* of the f is then

$$S(n) = T(f, n, 1)/T(f, n, p) \quad (4)$$

It then follows that

THEOREM 5.5. *Let p be the number of cores, n size of the input. If $p = o(n)$ [3] , then, the CC algorithms Algo* **??** *for scan, Algo.* **??** *for nested scan, Algo.* **??** *for second order difference equations, Algo.* **??** *for Fibonacci sequence, Algo.* **??** *for banded linear systems, Algo,* **??** *for tridiagonal linear systems have asymptotical speed up linear to the number of cores p.*
*Proof: In all the above algorithms, the compression and expansion phases take $O(n/p)$ time, and the global phase takes $O(p)$ time. The total time is then $T(f, n, p) = O(n/p) + O(p)$. Since $p = o(n), T(f, n, p) = O(n/p)$. By (**??**):*

$$S(n) = T(f, n, 1)/T(f, n, p) = O(n)/O(n/p) = O(p) \quad (5)$$

Also to be observed is

THEOREM 5.6. *The computational time of the global phase in a CC algorithm is a function of the number of cores p, and independent of the size n of the problem.*
Proof: obvious.

The above implies that if the sequential base constituents of a CC algorithm is an optimal one sequentially, then the CC algorithm is also an optimal multicore program in the sense that (1) it is a consistent algorithm; (2) it has linear speedup.

In Figure **??**, we plotted the speed up curve of some CC programs in Haskell on an Intel Xeon machine runing Linux OS with seven cores available to us. We omit the benchmark for nested scan because it is already implemented in terms of scan. All programs run entirely in memory, and we measure the speed by calculating the wall time each program takes from start to finish. Observe that the speedups for the three different problems are all nearly perfectly linear to the number of cores used for the computation.

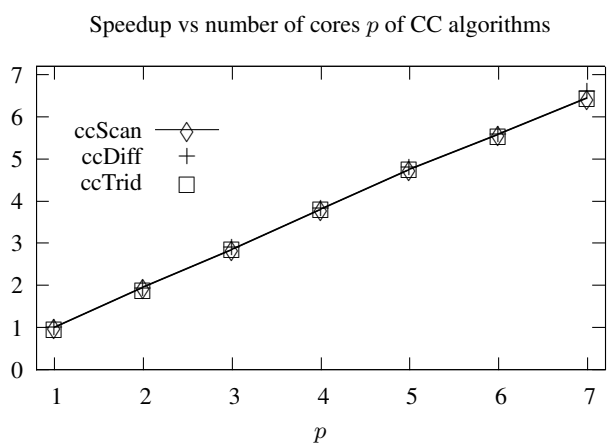

Figure 4. Speedup curve of CC programs in Haskell for scan, second order difference equation, and tridiagonal problem for $n = 10^6$ on multicore system with seven cores.

Theorem **??** may appear to be a direct violation to the Amdahl's Law [**?**] or Gustafson's Law [**?**]. There is a simple explanation to it. Both of the two laws assume some fixed percentage of either parallel or sequential portion of a program. Since this is not a valid assumption to be made for the problems we consider in this paper, neither Amdahl's nor Gustafson's Law is relevant here.

[3] $f(x) = o(g(x))$ if $\lim_{x \to \inf} f(x)/g(x) = 0$

6. Characteristics

In the above section, it is shown that, for a broad range of problems, CC paradigm can deliver multicore solutions optimal in computation and communication with linear speed ups. It is however unclear, what are the common characteristics of the problems that are subject to the CC programming paradigm with ideal performance.

To answer the above question, let us first introduce the notion of the $\mathcal{CC}$ *class*.

DEFINITION 6.1. *A problem is in the class of $\mathcal{CC}$ if and only it is subject to the CC form of (Def.* **??** *) with an unbounded compression ratio (Section* **??***).*

It follows that

THEOREM 6.1. *Scan, nested scan, second order linear difference equations, Fibonacci sequence, banded linear triangular system of bandwidth two, tridiagonal linear systems are in the class of $\mathcal{CC}$.*

To characterize problems in $\mathcal{CC}$, we need the following notions:

DEFINITION 6.2. *Let F be a function over input X. The reference graph of F is the pair $G = (V, R)$, where $V = \{x \mid x \in X\}$, and R is the binary relation, where $x_1 \; R \; x_2$ if and only if x_1 refers to x_2 in F.*

For instance, the reference graph for the problem of second order difference equations is a chain of vertices, each of which, with the exception of the first two, has two directed edges connecting it to the two previous ones respectively. Since this binary relation generally is not symmetric, the graph is directed.

Given a graph $G = (V, R)$, a *cut* is a binary partition of the vertices, and the *size of a cut* is the number of edges between the two partitions. A cut is *maximum* if its size is larger than any other cut.

Now we are in the position to identify a necessary condition for problems to be in the class of $\mathcal{CC}$:

THEOREM 6.2. *Let f be a problem in $\mathcal{CC}$, then there exist a reference graph $G = (V, R)$ for f with maximum cut independent of $|V|$, where $|V|$ denotes the cardinality of V.*
Proof: suppose this is not the case, we then can then use the reference graph as defined by the CC algorithm for that of the f. This graph however has maximum cut bounded by constant, leading to a contradiction.

The problem of second order difference equation for instance has a reference graph that meets the above condition.

It should come as no surprise that all problems are not known to possess reference graphs with constant bounded maximum cut as required by Theorem **??**. FFT and Bitonic Sort are examples of such problems.

Next, we are to show that the class of $\mathcal{CC}$ is characterized not only by the property of the reference graphs, but also by the complexity classes:

THEOREM 6.3. *Let $\mathcal{L}$ be the class of problems with computational complexity of $O(n)$ [4] , where n is the size of the problem. Then $\mathcal{CC} \subset \mathcal{L}$.*
Proof: suppose there is a problem $f \in \mathcal{CC}$, and $f \notin \mathcal{L}$.

[4] Here, the $O(n)$ refers to the linear complexity of a problem on a Turing machine.

Let $T(f, n, 1) = O(g(n))$, where g is not linear to n, f_s the base function of f. The time to compute f_s on each core will be $g(n/p)$. If we simulate the CC program for p cores on one core, the total time will be $O(mg(n/m))$. Since g is more than linear with n, it follows that $O(mg(n/m)) < O(g(n))$, which lead to a contradiction.

Theorem **??** and **??** point out rather severe limitations on the power of the compress-and-conquer paradigm. However, this does not invalidate the claim that the $\mathcal{CC}$ class contains a broad range of problems. Moreover, there are problems which, though in themselves are not in the class of $\mathcal{CC}$, but contain component(s) which are. Matrix multiplication, for instance, is clearly not in the class of $\mathcal{L}$, however, its main component, the inner product of a row and a column from the two factor matrices, is in the class of $\mathcal{CC}$ and can indeed be computed with a CC program.

7. Variations and Generalizations

7.1 Parallelized Core-Phase

Observe that in the CC form of Def. **??** we have chosen to apply the sequential base function over the compressed problem during the global phase, and as a result, the global phase computation is mapped into the internal computation inside a single appointed core (P_0, see Section **??**). Alternatively, one could choose a parallel program for the global phase. It can be shown, however, unless the number of cores is sufficiently large, the alternative parallel approach brings no benefit to performance, but only complicates the programming requirement. For if one goes that way, he must provide a separate parallel version of the base function in addition to the sequential version which is shared in all the three phases under the proposed scheme.

7.2 Specialized Sequential Function

An interesting aspect of CC is that the sequential function f_s is applied three times, one in each of the three phases:

1. In compression phase, f_s only partially solves each segment of the original input data;
2. The compressed results form a much smaller problem in the global phase, which is completely solved by f_s;
3. The solution to the compressed problem is expanded to modify each segment of the original data, which then are completely solved by f_s.

For this reason, we'll call f_s the *generalized solver* for a given problem. But in order to re-use the same f_s, we have to retain the original data until the last phase. A consequence made more apparent by the monadic cc_m is that in the compression phase it has to make copies of the input segments otherwise f_s would modify them in place. This is of course an implementation issue that can be addressed, for instance, by some fusion technique. A more fundamental question is: *can we re-use the result of f_s from the compression phase without having to keep the original data around?*

The answer is *yes*. Instead of relying on just one f_s for all phases, we can take another sequential function g_s that we call a *specialized solver*, and formulate a different CC algorithm below:

$$
\begin{aligned}
&cc'\ d\ c\ co\ xp\ com_g\ com_h\ f_s\ g_s = post\ .\ first\ core\ .\ pre\\
&\quad \text{where } pre\ \ = unzip\ .\ map\ ((co \times id)\ .\ f_s)\ .\ d\\
&\qquad\qquad core = d\ .\ com_h\ .\ f_s\ .\ com_g\ .\ c\\
&\qquad\qquad post\ = c\ .\ map\ (g_s\ .\ xp)\ .\ (uncurry\ zip)\\
&\qquad\qquad first\ f\ (x, y) = (f\ x, y)\\
&\qquad\qquad f \times g\ x\qquad = (f\ x, g\ x)
\end{aligned}
$$

Just like the original algorithm, *cc'* still contains three phases, but in the compression phase, it actually passes the results from function f_s directly to the expansion phase, and function g_s would pick up from where f_s has left and work out a complete solution with the expanded information obtained from the global phase.

In terms of complexity, cc' is on the same order as cc. But in an actual implementation, it may perform better because the specialized solver g_s may require less computation steps than the generalized solver since it has a partial solution to start with. Theoretically, however, we still prefer the original CC formulation in Def. **??** which is easier to reason about due for its simplicity.

7.3 Higher Order CC

A compress-and-conquer with a sequential base function is said to be of *first order*. Inductively, a compress-and-conquer is said to be of a $(k+1)$-th order CC algorithm if its base function f_s is a *k-th order compress-and-conquer*.

Let us consider a second order compress-and-conquer, with arity n at top level, m the bottom level. It can be mapped to a multicore system with n interconnected nodes, each with m cores. It is easy to show that

THEOREM 7.1.
(1) A second $(k+1)$-th order CC is operation and communication optimal if and only if its (k-th order) base function is.
(2) The speedup of a second order compress-and-conquer with arities n and m at top and base levels respectively mapped to n nodes with m cores is respectively linear to n and m.

Observe that second order CC form provides a simple and elegant framework to program hierarchical systems with multiple nodes of multicore units with guaranteed optimal performance.

It should also be obvious the above theorem can be generalized to CC algorithms with order greater than two.

8. Relation to Divide-and-Conquer

Divide-and-conquer (DC) has been shown to be one of the most effective paradigms for deriving elegant and efficient parallel solutions to a wide variety of problems [**? ? ?**]. Both DC and CC solves a problem by dividing it into sub-problems. However, the arity of the division in DC is some constance such as two, while CC uses division with arity variable in the number of processing units of the system; DC is recursive, while CC is not; A DC algorithms usually is a different algorithm from the best known sequential counterpart altogether, while CC algorithm is always derived from a sequential algorithm for the same problem.

It should also be pointed out that the two paradigms are not equivalent in their computational power. Given Theorem **??**, the computational power of compress-and-conquer is necessarily strictly weaker than that of divide-and-conquer.

Also to note is for problems that can be solved under either paradigm, the two paradigms do not necessarily lead to the same performance. Scan, for example, although the problem has $O(n)$ operational complexity, a DC algorithm would require $O(nlog(n))$ operations. In contrast, as shown in Theorem **??**, an CC algorithm would be operational optimal.

Finally, we would like to point out that automatic transformation between DC and CC programs are possible when under certain conditions. Our previous work on divide-and-conquer introduced the notion of *pre-* and *post- morphism* as algebraic models for DC, and it was pointed out that a broad range of scientific problems can be solved with three types of communications, namely, last-k, correspondent, and mirror-image [**? ?**]. It can be shown that a postmorphism [**? ?**] algorithm with last-k communication can be automatically transformed into a CC program, and vice versa, which limited by space will has to be elaborated elsewhere.

9. Related Work

Much effort has been made to support high level programming for multicore computing. Some noticeable examples are the Threading Building Blocks from Intel [**?**], Parallel Task Library from Microsoft, and the Data Parallel Haskell project [**?**] from the functional programming community. The CC paradigm proposed here differs from any of the above approaches in a number of fundamental ways. Firstly, it does not expose any of the mechanisms related to multicore architecture such as thread, mutex, and task queues. Secondly, its does not expose any imperative constructs such parallel-for or parallel loops. Finally, instead of relying on programming constructs such as reduce and scan, it provides a much a higher order form from which the constructs could be derived.

Solving a problem through compression is not an entirely new idea. There is a known technique in parallel computing, referred to as *odd-even reduction*. Ladner and Fischer [**?**] used this technique in an elegant parallel scan algorithm. With odd-even reduction, a problem is recursively reduced in size by a factor of two. As a result, the number of steps required is logarithmic to the size of the problem during both the reduction and expansion phase. In contrast, the CC paradigm has unbounded compression ratio, and takes one step during both the compression and expansion. Another obvious difference is that an algorithm based on odd-even reduction is totally a different algorithm from its sequential counterpart, while a CC algorithm employs the sequential counterpart as the core of its computation.

Nested data parallelism has been shown to be an expressive and effective approach to multicore programming [**? ?**], which can be traced back to the earlier work on the language NESL and nested scan [**? ?**]. From data structure point of view, both Data Parallel Haskell and compress-and-conquer introduced new kinds of arrays operations. The two approaches however have salient differences in nature. First of all, the division of arrays in the former are non-polymorphic in that the result depends on the values of the array entries through the use of array comprehension (e.g. the division used in quick-sort), while in the latter, polymorphic structural operations are of fundamentally importance to the paradigm (non-polymorphic operations can be implemented with polymorphic operations). Secondly, in spite of a large number of primitives built into the parallel arrays of the former, data communication is completely hidden, and programmers have to trust the compiler in doing a good job at balancing tasks. In contrast, communication in the latter is first-class citizen. Thirdly, monadic composition (in its comprehension form) is the main theme in the former, while higher order functional forms are the center pieces of the latter.

M. Cole et. al. with their work on programming skeletons have shown how higher order functions can be adapted to work with non-declarative languages for the purpose of parallel programming [**? ?**] Under their framework, as a higher form, compress-and-conquer can be considered as another algorithmic skeleton different but related to divide-and-conquer, which they have identified as an important parallel algorithmic skeleton.

10. Conclusion

We proposed CC as efficient paradigm for multicore computation, showed how it can be implemented as Haskell higher order functions. The expressive power of the paradigm is illustrated with its application to a number of problems including scan, difference equations, banded linear systems, and linear tridiagonal systems. The optimality of CC programs are proven and confirmed by the benchmarks of the CC programs on multicore machine. Besides the linear speedup, the CC paradigm also could reduce the complexity of multlicore programming for it allows a sequential program to be used as the core component for the computation on multicore system. While not all problems are subject to the paradgim, the computational power is shown to subsume that of scan, nested scan, and mapReduce.

Acknowledgement This research was supported in part by a grant from Microsoft Research.