Double-Scan: Introducing and Implementing a New Data-Parallel Skeleton

Holger Bischof and Sergei Gorlatch

Technical University of Berlin, Germany

Abstract. We introduce a new reusable component for parallel programming, the *double-scan skeleton*. For this skeleton, we formulate and formally prove sufficient conditions under which the double-scan can be parallelized, and develop its efficient MPI implementation. The solution of a tridiagonal system of equations is considered as our case study. We describe how this application can be developed using the double-scan and report experimental results for both absolute performance and performance predictability of the skeleton-based solution.

1 Introduction

This work is motivated by the search for convenient, reusable, adaptable components for building parallel applications. We pursue the approach based on *skeletons* – typical algorithmic patterns of parallelism. The programmer composes an application using skeletons as high-level language constructs, whose implementations for different parallel machines are provided by a compiler or a library. Formally, skeletons are viewed as higher-order functions, customizable by means of application-specific functional parameters. Well-known examples of practical skeleton-based programming systems include P3L [9] and Skil [2] in the imperative setting, as well as Eden [3] and HDC [7] in the functional world.

This paper demonstrates by reference to an application case study – the solution of a tridiagonal system of linear equations – how a new data-parallel skeleton called *double-scan* is identified, implemented and added to an existing inventory of components.

The contributions and structure of the paper are as follows:

– We describe a basic skeleton repository containing well-known data-parallel skeletons, such as map, reduce, zip, and two scans – left and right (Section 2).
– We express the problem of solving a tridiagonal system of equations using the basic skeletons, demonstrating the need for a new skeleton (Section 3).
– We introduce the new double-scan skeleton – a composition of two scans, one of which has a non-associative base operator – and prove the sufficient condition under which it can be parallelized (Section 4).
– We demonstrate how the performance of programs using the double-scan skeleton can be predicted in advance and demonstrate both the absolute performance and the possibility of predicting the performance of our MPI implementation by experiments on a Cray T3E (Section 5).

We conclude the paper by discussing our results in the context of related work.

2 Basic Data-Parallel Skeletons

In this section, we present some basic data-parallel skeletons as higher-order functions defined on non-empty lists, function application being denoted by juxtaposition, i. e. $f\,x$ stands for $f(x)$:

– Map: Applying a unary function f to all elements of a list:

$$\mathit{map}\, f\ [x_1, \dots, x_n] \ = \ [f\ x_1,\ \dots,\ f\ x_n]$$

– Red: Combining the list elements using a binary associative operator $\oplus$:

$$\mathit{red}(\oplus)([x_1, \dots, x_n]) \ = \ x_1 \oplus \cdots \oplus x_n$$

– Zip: Component-wise application of a binary operator $\odot$ to a pair of lists of equal length:

$$\mathit{zip}(\odot)([x_1, \dots, x_n], [y_1, \dots, y_n]) \ = \ [\,(x_1 \odot y_1), \dots, (x_n \odot y_n)\,]$$

– Scan-left and scan-right: Computing prefix sums of a list by traversing the list from left to right (or vice versa) and applying a binary operator $\oplus$:

$$\begin{aligned}\mathit{scanl}(\oplus)([x_1, \dots, x_n]) &= [\,x_1,\ (x_1 \oplus x_2), \dots, (\cdots(x_1 \oplus x_2) \oplus x_3) \oplus \cdots \oplus x_n)\,]\\ \mathit{scanr}(\oplus)([x_1, \dots, x_n]) &= [\,(x_1 \oplus \cdots \oplus (x_{n-2} \oplus (x_{n-1} \oplus x_n) \cdots),\ \dots,\ x_n\,]\end{aligned}$$

We call these functions skeletons because each of them describes a whole class of functions, obtainable by substituting application-specific operators for parameters $\oplus$, $\odot$ and f.

Our skeletons have obvious data-parallel semantics: the asymptotic parallel complexity is constant for *map* and *zip* and logarithmic for *red* and both scans, if $\oplus$ is associative. If $\oplus$ is non-associative, then *red* and the scans are computed sequentially with linear time complexity.

3 Case Study: Tridiagonal System Solver

In this section, we consider an application example and try to parallelize it using the skeletons introduced in Section 2. Our case study is concerned with solving a tridiagonal system of linear equations, $A \cdot x = b$, where A is an $n \times n$ matrix representing coefficients, x a vector of unknowns and b the right-hand-side vector. The only values of matrix A unequal to 0 are on the main diagonal as well as above and below it (we call them the upper and lower diagonal, respectively), as demonstrated by equation (1).

$$\begin{pmatrix} a_{12} & a_{13} & & & \\ a_{21} & a_{22} & a_{23} & & \\ & \ddots & \ddots & \ddots & \\ & & a_{n-1,1} & a_{n-1,2} & a_{n-1,3} \\ & & & a_{n,1} & a_{n,2} \end{pmatrix} \cdot x \ = \ \begin{pmatrix} a_{14} \\ a_{24} \\ \vdots \\ a_{n-1,4} \\ a_{n,4} \end{pmatrix} \tag{1}$$

Our first step is to cast the tridiagonal system in the list notation since our skeletons work on lists. We choose a representation comprising a list of rows, each inner row i consisting of four values: the value $a_{i,1}$ that is part of the lower diagonal of matrix A, the value $a_{i,2}$ at the main diagonal, the value $a_{i,3}$ at the upper diagonal, and the value $a_{i,4}$ that is the i-th component of the right-hand-side vector b.

To make the first and last row consist of four values, too, we add to the matrix the fictitious zero elements a_{11} and $a_{n,3}$. The fictitious values obviously do not change the solution of the original tridiagonal system: imagine two additional unknowns, x_0 and x_{n+1}, and two additional columns, one on the left of the original matrix and the other on the right, whose values are equal to zero.

We can thus represent the whole tridiagonal system as the following list of rows, each of which is a quadruple:

$$\big[\,(a_{11}, a_{12}, a_{13}, a_{14}),\ (a_{21}, a_{22}, a_{23}, a_{24}),\ \ldots,\ (a_{n1}, a_{n2}, a_{n3}, a_{n4})\,\big]$$

A typical algorithm for solving a tridiagonal system is Gaussian elimination (see e. g. [8,10]) which eliminates the lower and upper diagonal of the matrix as shown in Figure 1. Note that both the first and last column in the figure consist of fictitious zero elements.

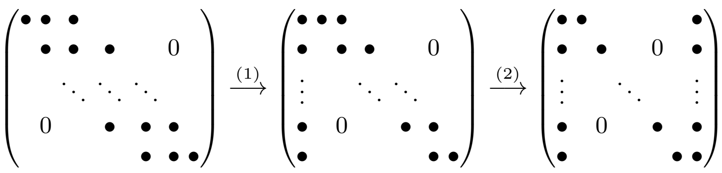


Fig. 1. The intuitive algorithm for solving a tridiagonal system of equations consists of two stages: (1) elimination of the lower diagonal; (2) elimination of the upper diagonal.

The two stages of the algorithm traverse the list of rows, applying operators denoted by ① and ②, which are informally defined as follows:

1. The first stage eliminates the lower diagonal by traversing matrix A from top to bottom according to the *scanl* pattern and applying operator ① on the rows pairwise:

$$(a_1, a_2, a_3, a_4)\ \textcircled{1}\ (b_1, b_2, b_3, b_4) \ = \ \left(a_1,\ a_3 - \frac{b_2 a_2}{b_1},\ -\frac{b_3 a_2}{b_1},\ a_4 - \frac{b_4 a_2}{b_1}\right)$$

2. The second stage eliminates the upper diagonal of the matrix by a bottom-up traversal, i. e. using the pattern of *scanr* and applying operator ② on pairs of rows:

$$(a_1, a_2, a_3, a_4)\ \textcircled{2}\ (b_1, b_2, b_3, b_4) = \left(a_1 - \frac{b_1 a_3}{b_2},\ a_2,\ -\frac{b_3 a_3}{b_2},\ a_4 - \frac{b_4 a_3}{b_2}\right) \qquad (2)$$

Now we can specify the described Gaussian elimination algorithm as function *tds* (*tridiagonal system*), which performs in two stages:

$$tds \;=\; scanr(②) \circ scanl(①) \tag{3}$$

where $\circ$ denotes function composition from right to left, i. e. $(f \circ g)\ x = f(g(x))$.

If both customizing operators of the scan skeletons in (3), ① and ②, were associative, our task would now be completed: both scan skeletons have data-parallel semantics, and furthermore, they can be directly implemented using, for example, the MPI collective operation `MPI_Scan`. However, since operator ① is not associative, *scanl* in (3) prescribes strictly sequential execution.

An alternative representation of the algorithm eliminates first the upper and then the lower diagonal using two new row operators, ③ and ④:

$$(a_1, a_2, a_3, a_4)\ ③\ (b_1, b_2, b_3, b_4) = \left(a_1,\ a_2 - \frac{b_1 a_3}{b_2},\ -\frac{b_3 a_3}{b_2},\ a_4 - \frac{b_4 a_3}{b_2}\right)$$

$$(a_1, a_2, a_3, a_4)\ ④\ (b_1, b_2, b_3, b_4) = \left(a_1, -\frac{b_2 a_2}{b_1},\ a_3 - \frac{b_3 a_2}{b_1},\ a_4 - \frac{b_4 a_2}{b_1}\right)$$

This version of the algorithm can be specified as follows:

$$tds \;=\; scanl(④) \circ scanr(③) \tag{4}$$

Again, however operator ③ in (4) is not associative and thus the first step of algorithm (4), *scanr*(③) cannot be directly parallelized.

4 The Double-Scan Skeleton and Its Parallelization

This section deals with the algorithmic pattern we identified using the case study in Section 3 – a sequential composition of two scans. We call a composition of two scans, one of which is the "left" and the other the "right" scan, the *double-scan skeleton.* This skeleton has two functional parameters, which are the base operators of the constituting scans. The non-associativity of the first scan in the composition prevents parallelization of the double-scan.

To parallelize the double-scan skeleton, we relate it to a more complex algorithmic pattern than the basic skeletons introduced in Section 2. Let us consider a class of functions called *distributable homomorphisms* (DH), which was first introduced in [6]:

Definition 1. *A function h is a distributable homomorphism iff there exist binary operators $\oplus$ and $\otimes$, such that for arbitrary lists x and y of equal length, which is a power of two, it holds:*

$$h[a] \;=\; [a]\,, \qquad h(x +\!\!\!+\, y) \;=\; zip(\oplus)(h\ x, h\ y) +\!\!\!+\, zip(\otimes)(h\ x, h\ y) \tag{5}$$

For operators $\oplus$ and $\otimes$, we denote the corresponding DH by $h = (\oplus \updownarrow \otimes)$. Its computation schema is illustrated in Figure 2.

In [6], a generic parallel implementation of an arbitrary DH is developed, which works on a logical hypercube. For the sake of brevity, we give the MPI implementation in pseudocode:

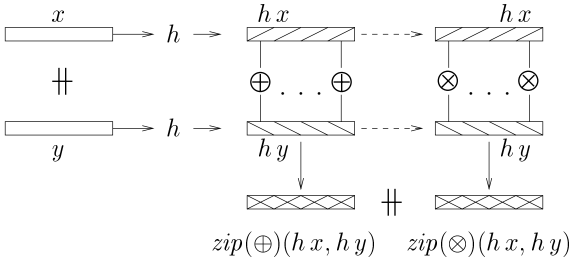


Fig. 2. Distributable homomorphism h on a concatenation of two lists, x and y: apply h to x and y, then combine the results elementwise with operators $\oplus$ and $\otimes$ and concatenate them (bottom of figure).

```
int MPI_DH(void *data, ..., MPI_User_function ⊕, ⊗, ...) {
  for (dim=1; dim<p; dim<<=1) {
    dest = my_rank^dim;      /* rank of hypercube neighbour */
    MPI_Sendrecv(data, ..., dest, temp, ..., dest, ...);
    if (my_rank<dest) data=data⊕temp; else data=temp⊗data; }}
```

Here, `MPI_Sendrecv` acts as a typical operational pattern of the hypercube behaviour: pairwise, bidirectional communication of the neighbouring nodes in dimension `dim`, followed by the application of either $\oplus$ or $\otimes$ in each of the nodes.

The novel result of this paper is the following sufficient condition under which the double-scan skeleton can be expressed using the DH pattern:

Theorem 1. *Let ①, ②, ③, ④ be binary operators, where ② and ④ are associative (①, ③ need not be associative). If the following equation holds:*

$$scanr(②) \circ scanl(①) \;=\; scanl(④) \circ scanr(③) \quad \overset{\text{def}}{=} \; s \tag{6}$$

then s from (6) can be represented as follows:

$$s \;=\; map(\pi_1) \circ \oplus\updownarrow\otimes \;\circ\; map(triple) \tag{7}$$

with the operators $\oplus$ and $\otimes$ defined as follows:

$$\begin{pmatrix}a_1\\a_2\\a_3\end{pmatrix} \oplus \begin{pmatrix}b_1\\b_2\\b_3\end{pmatrix} = \begin{pmatrix}a_1 ② (a_3 ① b_2)\\a_2 ② (a_3 ① b_2)\\(a_3 ③ b_2) ④ b_3\end{pmatrix} \quad \begin{pmatrix}a_1\\a_2\\a_3\end{pmatrix} \otimes \begin{pmatrix}b_1\\b_2\\b_3\end{pmatrix} = \begin{pmatrix}(a_3 ③ b_2) ④ b_1\\a_2 ② (a_3 ① b_2)\\(a_3 ③ b_2) ④ b_3\end{pmatrix} \tag{8}$$

Here, *triple* maps an element to a triple, $triple\, a = (a, a, a)$, and function π_1 extracts the first element of a triple, $\pi_1(a, b, c) = a$.

We do not present the theorem's proof here owing to the lack of space; it is contained in a technical report [1].

To apply the general result of Theorem 1 to our example of a tridiagonal system solver, it remains to be shown that operators ② in (3) and ④ in (4) are associative. For example, the associativity of ② defined in (2) is demonstrated below using the associativity of addition and multiplication and the distributivity of multiplication over addition:

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} \textcircled{2} \left[\begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \textcircled{2} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix} \right] = \begin{pmatrix} a_1 - \frac{b_1 a_3}{b_2} + \frac{c_1 b_3 a_3}{c_2 b_2} \\ a_2 \\ \frac{c_3 b_3 a_3}{c_2 b_2} \\ a_4 - \frac{b_4 a_3}{b_2} + \frac{c_4 b_3 a_3}{c_2 b_2} \end{pmatrix} = \left[\begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{pmatrix} \textcircled{2} \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \right] \textcircled{2} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{pmatrix}$$

Analogously, the associativity of operator $\textcircled{4}$ can be proved. Now a parallel implementation of the DH skeleton can be used to compute the *tds* function.

5 Performance Prediction and Measurements

In this section, we look at the performance of programs using the double-scan skeleton from two perspectives: (1) whether performance for a particular application on a particular parallel machine can be predicted in advance, and (2) whether absolute performance is competitive with that of a hand-coded solution.

Programming with skeletons offers a major advantage in terms of performance prediction: performance has to be estimated only once for a skeleton. This generic estimate can then be tuned to a particular machine and application, rather than redoing the estimation procedure for each new machine and application. We demonstrate below possible tuning steps in performance prediction: we derive an estimate for the DH skeleton, and then tune it: first to a particular machine (Cray T3E), and then to a particular application (tridiagonal system solver).

Generic Performance Estimate. Let p be the number of processes and m the data-block size per process. Our implementation of the DH skeleton consists of $\log_2 p$ steps, each performing a bidirectional communication with data of length m and the computation with $\oplus$ or $\otimes$. Operators $\oplus$ and $\otimes$ are applied elementwise to the data, consisting of m elements. The resulting time estimate is:

$$t = \log_2 p \cdot (t_s + m \cdot (t_w + t_c)) \tag{9}$$

where t_s is the communication startup time, t_w the time needed to communicate one word, and t_c the maximum time for one computation with $\oplus$ or $\otimes$.

Estimate Tuned to Cray T3E. Variables t_s and t_w are machine-dependent. On a Cray T3E, t_s is approximately $16.4\,\mu s$. The value of t_w also depends on the size of the data type: $t_w = d \cdot t_B$, where t_B is the time needed to communicate one byte, and d is the byte count of the data type. Measurements show that for large array sizes the bidirectional bandwidth on our machine is approximately 300 MB/s, i. e. $t_B \approx 0.0033\,\mu s$. Inserting these values of t_s and t_B into (9) leads to the following runtime estimate for the double-scan skeleton on a Cray T3E:

$$t_{\text{Cray}} = \log_2 p \cdot (16.4\,\mu s + m \cdot (d \cdot 0.0033\,\mu s + t_c)) \tag{10}$$

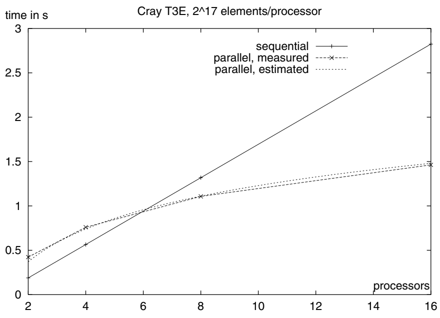
Estimate Tuned to the Tridiagonal System Solver. The estimate for a particular machine can be further specialized for a particular application, i. e. an instance of the skeleton. Let us predict the runtime of the tridiagonal system solver on a Cray T3E. In computing *tds*, our data type is a triple of quadruples containing double values and has a size of $d = 96$ Bytes. Measurements show that one operator $\oplus$ or $\otimes$ in (8) for tridiagonal systems requires time $t_c \approx 2.44\,\mu s$ (measured by executing $\oplus$ in a loop over a large array). Inserting d and t_c into (10) results in:

$$t_{\text{Cray-tds}} = \log_2 p \cdot (16.4\,\mu s + m \cdot 2.76\,\mu s) \tag{11}$$

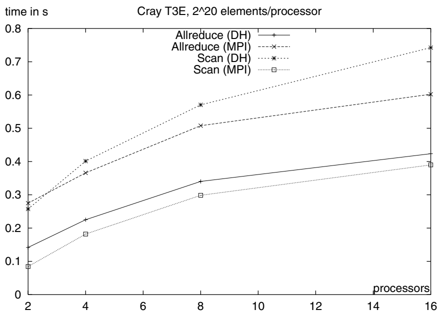
The next specialization step is to substitute a particular problem size m. We compute with 2^{17} elements per process, where one element is a quadruple of float values in the sequential algorithm and a triple of quadruples of float values in the parallel algorithm; the *tds* function is computed elementwise on the input data. The resulting estimate for our problem size is:

$$T = \log_2 p \cdot 0.36\,s$$

Figure 3(a) compares the estimated and measured time for the tridiagonal system solver and demonstrates the high quality of our estimates.



Runtimes of the sequential vs. parallel (estimated and measured) version of the tridiagonal system solver.



Comparison `MPI_DH` vs. `MPI_Scan` and `MPI_Allreduce` on Cray T3E with 2^{20}

Fig. 3. Measurements

Absolute Performance of DH Implementation. To show that our DH implementation is competitive with hand-coded implementations, we tested it under fairly hard conditions. Since allreduce and scan are both DHs ([6]), we use the `MPI_DH` function to compute scan and allreduce and compare it to the performance of two native collective MPI implementations – `MPI_Scan` and `MPI_Allreduce`. In other words, we compare the performance of particular instantiations of our generic DH implementation with specially developed, hand-coded native MPI implementations, optimized for a particular machine.

Figure 3(b) shows the results for a Cray T3E with 2^{20} elements per process using elementwise integer addition as the base operator. Note that the overall number of elements grows proportionally to the number of processes. The results of measurements show that our DH implementation, despite its simplicity, exhibits very competitive performance.

6 Related Work and Conclusion

The desire to be able to name and reuse "programming patterns", i. e. to capture them in the form of parameterizable abstractions, has been a driving force in the evolution of high-level programming languages in general. In the sequential setting, design patterns [5] and components [11] are recent examples of this. In parallel programming, where algorithmic aspects have traditionally been of special importance, the approach using algorithmic skeletons [4] has emerged. Related work on skeletons is manifold and was partially cited in the introduction.

In this paper, we systematically developed a parallel implementation for solving a tridiagonal system of equations using a novel generic program component (the double-scan skeleton), which is reusable for other classes of applications. The use of the skeleton also allowed a systematic performance prediction, the results of which have been confirmed in machine experiments.

References

1. H. Bischof, S. Gorlatch, and E. Kitzelmann. The double-scan skeleton and its parallelization. Technical Report 2002/06, Technische Universität Berlin, 2002.
2. G. Botorog and H. Kuchen. Efficient parallel programming with algorithmic skeletons. In L. Bougé et al., editors, *Euro-Par'96: Parallel Processing*, Lecture Notes in Computer Science 1123, pages 718–731. Springer-Verlag, 1996.
3. S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden coordination model for distributed memory systems. In *High-Level Parallel Programming Models and Supportive Environments (HIPS)*. IEEE Press, 1997.
4. M. I. Cole. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. PhD thesis, University of Edinburgh, 1988.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elemets of reusable object-oriented software*. Addison Wesley, 1995.
6. S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96: Parallel Processing, Vol. II*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.
7. C. A. Herrmann and C. Lengauer. HDC: A higher-order language for divide-and-conquer. *Parallel Processing Letters*, 10(2–3):239–250, 2000.
8. F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publ., 1992.
9. S. Pelagatti. *Structured development of parallel programs*. Taylor&Francis, 1998.
10. M. J. Quinn. *Parallel Computing*. McGraw-Hill, Inc., 1994.
11. C. Szyperski. *Component software: beyond object-oriented programming*. Addison Wesley, 1998.