Cost Optimality and Predictability of Parallel Programming with Skeletons

Holger Bischof, Sergei Gorlatch, and Emanuel Kitzelmann

Technical University of Berlin, Germany
{bischof|gorlatch|jemanuel}@cs.tu-berlin.de

Abstract. Skeletons are reusable, parameterized components with well-defined semantics and pre-packaged efficient parallel implementation. This paper develops a new, provably cost-optimal implementation of the DS (*double-scan*) skeleton for the divide-and-conquer paradigm. Our implementation is based on a novel data structure called *plist* (pointed list); implementation's performance is estimated using an analytical model. We demonstrate the use of the DS skeleton for parallelizing a tridiagonal system solver and report experimental results for its MPI implementation on a Cray T3E and a Linux cluster: they confirm the performance improvement achieved by the cost-optimal implementation and demonstrate its good predictability by our performance model.

1 Introduction

A promising approach to improve and systematize the parallel programming process is to use well-defined patterns of parallelism, called *skeletons* [4]. The programmer expresses an application using skeletons as reusable, parameterized components, whose efficient implementations for particular parallel machines are provided by a compiler or library. A rich collection of both very basic and complex parallel skeletons have been proposed and implemented in systems like P3L, HDC, Skil, SKElib etc. [3,6,7,11], and have a potential to be actively used in future problem solving environments [5].

This paper addresses two questions that have a major impact on the choice and use of skeletons in the programming process: (1) whether the parallel cost of a particular skeleton implementation, i. e. its time-processor product [15], is optimal, and (2) whether the performance of a skeleton-based program can be predicted in advance early on in the design process. We answer these questions for two fairly complicated skeletons introduced by ourselves earlier, both embodying the divide-and-conquer paradigm: DH (distributable homomorphism) [8] and DS (double-scan) [1].

The contributions and organization of the paper are as follows:

– We introduce basic data-parallel skeletons and our two skeletons DH and DS, and prove conditions for the generic parallel implementations of both DH and DS to be cost-optimal (Section 2).

H. Kosch, L. Böszörményi, H. Hellwagner (Eds.): Euro-Par 2003, LNCS 2790, pp. 682–693, 2003.

– We introduce a novel data structure called *plists (pointed lists)*, develop a new parallel DS-implementation that uses plists internally, estimate the runtime of this implementation and prove its cost optimality (Section 3).
– We report experimental results for a tridiagonal system solver based on the DS-skeleton on a Cray T3E and a Linux cluster using MPI. They demonstrate both a substantial performance improvement achieved by our cost-optimal implementation and the high quality of the analytical runtime prediction (Section 4).

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

2 Skeletons and Their Cost Properties

A skeleton can be formally viewed as a higher-order function, customizable for a particular application by means of functional parameters that are provided by the application programmer. The first parallel skeletons studied in the literature were traditional second-order functions known from functional programming: map, reduce, scan, etc. They are defined on non-empty lists as follows, 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:

$$map\, f\ [x_1, \ldots, x_n] \ = \ [f\ x_1, \ \ldots, \ f\ x_n]$$

– Zip: Component-wise application of a binary operator $\oplus$ to a pair of lists of equal length (it is similar to the Haskell function `zipWith`):

$$zip(\oplus)([x_1, \ldots, x_n], [y_1, \ldots, y_n]) \ = \ [\,(x_1 \oplus y_1), \ldots, (x_n \oplus y_n)\,]$$

– 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 $\oplus$:

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

We call these second-order functions "skeletons" (or patterns) because each of them describes a whole class of functions, obtainable by substituting application-specific operators for parameters $\oplus$ and f.

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

The need to manage important classes of applications led to the introduction of more complex skeletons, e. g. different variants of divide-and-conquer, etc. In [8], we defined one such skeleton, the DH (distributable homomorphism):

Definition 1. *The DH skeleton is a higher-order function with two parameter operators, $\oplus$ and $\otimes$, defined as follows for arbitrary lists x and y of equal length, which is a power of two:*

$$\begin{aligned} dh\,(\oplus,\otimes)\,[a] &= [a]\,, \\ dh\,(\oplus,\otimes)\,(x +\!\!+\, y) &= zip(\oplus)(dh\ x, dh\ y) +\!\!+\, zip(\otimes)(dh\ x, dh\ y) \end{aligned} \tag{1}$$

The DH skeleton is a special form of the well-known divide-and-conquer paradigm: to compute *dh* on a concatenation of two lists, $x +\!\!+\, y$, we apply *dh* to x and y, then combine the results elementwise using *zip* with the parameter operators $\oplus$ and $\otimes$ and concatenate them. For this skeleton, there exists a family of generic parallel implementations, directly expressible in MPI [9].

In [1], we introduced a more special skeleton, *double-scan* (DS), which is more convenient than DH for expressing some application classes:

Definition 2. *For binary operators $\oplus$ and $\otimes$, two double-scan (DS) skeletons are defined:*

$$scanrl(\oplus,\otimes) = scanr(\oplus) \circ scanl(\otimes) \tag{2}$$

$$scanlr(\oplus,\otimes) = scanl(\oplus) \circ scanr(\otimes) \tag{3}$$

where $\circ$ denotes function composition from right to left, i. e. $(f \circ g)\ x = f(g(x))$. Both double-scan skeletons have two functional parameters, which are the base operators of their constituent scans. If the parameter operator $\oplus$ is associative, then as shown in [1], DS can be parallelized.

An important efficiency criterion for parallel algorithms is their *cost*, which is defined as the product of the required time and the number of processors used, i. e. $c = t \cdot p$ (see [15]). A parallel implementation is called *cost-optimal* on p processors, iff its cost equals the cost on one processor, i. e. $p \cdot t_p \in \Theta(t_{seq})$. Here and in the rest of this paper, we use the classical notations o, O, ω and Θ to describe asymptotic behaviour (see, e. g., [12]).

Our first question is whether the known implementations of DH [8] and DS [1] are cost-optimal. We are especially interested in a skeleton's generic implementation, which can be applied to all particular instances of the skeleton.

The *generic DH-implementation* partitions the input list into p blocks of approximately the same size, computes DH locally in the processors, and then performs $\log_2 p$ rounds of pairwise communications and computations in a hypercube-like manner [8]. Its time complexity is $\Theta(t_{seq}(m) + m \cdot \log p)$, where $t_{seq}(m)$ is the time taken to sequentially compute DH on a block of length $m \approx n/p$. There always exists an obvious sequential implementation of the formula (1) on a data block of size m, which has a time complexity of $\Theta(m \cdot \log m)$. So, in the general case, the time complexity of the generic DH-implementation is $\Theta(n/p \cdot \max\{\log(n/p), \log p\})$.

The sequential time complexity of a particular DH instance may also be asymptotically smaller than $m \cdot \log m$. An example relevant for us is the DS skeleton. On the one hand, as proved in [1], the DS skeleton can be expressed as

an instance of the DH skeleton for particular values of parameters $\oplus$ and $\otimes$, with additional pre- and postcomputations performed locally. On the other hand, the sequential time complexity of DS is obviously linear.

The following theorem shows how the cost optimality of the generic DH-implementation, used for a particular instance of the DH-skeleton, depends on the optimal sequential complexity of this instance, i. e. particular application:

Theorem 1. *The generic parallel implementation of DH, when used for a DH instance with optimal sequential time complexity of* t_{seq}*, is*
(a) cost-optimal on $p \in O(n)$ *processors, if* $t_{seq}(n) \in \Theta(n \cdot \log n)$*, and*
(b) non-cost-optimal on $p \in \omega(1)$ *processors, if* $t_{seq}(n) \in o(n \cdot \log p)$*.*

Here is the proof sketch: (a) $c_p \in \Theta(n \cdot \max\{\log(n/p), \log p\}) \subseteq O(n \cdot \log n)$, and (b) $c_p \in \Theta(n \cdot \max\{\log(n/p), \log p\}) \subseteq \Omega(n \cdot \log p)$.

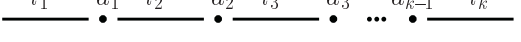
From Theorem 1(b) it follows that the generic DH-implementation is not cost-optimal for the DS skeleton, whose $t_{seq}(n) \in \Theta(n)$. Our next question is whether there exists a cost-optimal implementation of the DS skeleton? Since there are applications that are instances of the DS skeleton and that still have cost-optimal hand-coded implementations, we can strive to find a generic, cost-optimal solution for all instances. This motivates our further search for a better parallel implementation of DS.

3 Towards a Cost-Optimal Double-Scan

In this central section of the paper, we develop a new generic, cost-optimal implementation of the DS skeleton. The implementation makes internal use of a novel special data structure, which, however, remains invisible to the programmer using the skeleton.

3.1 Plists and Functions on Them

We introduce a special intermediate data structure – *pointed lists* (*plists*). A *k-plist*, where $k > 0$, consists of k conventional lists, called segments, and $k - 1$ points between the segments:



If parameter k is irrelevant, we simply speak of a plist instead of a k-plist. Conventional lists are obviously a special case of plists. To distinguish between functions on lists and plists, we prefix the latter with the letter p, e. g. *pmap*. To transform between lists and plists, we use the following two functions:

– $list2plist_k$ transforms a list into a plist, consisting of k segments and $k-1$ points. It partitions an arbitrary list into k segments:

$$list2plist_k\,(l_1 +\!\!+ [a_1] +\!\!+ \cdots +\!\!+ [a_{k-1}] +\!\!+ l_k) = [\,l_1, a_1, \ldots, a_{k-1}, l_k\,]$$

Our further considerations are valid for arbitrary partitions but, in practice of parallelism, one tries to obtain segments of approximately the same size.

– $plist2list_k$ is the inverse of $list2plist_k$, transforming a k-plist into a conventional list:

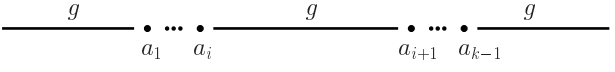
$$plist2list([\,l_1, a_1, \ldots, a_{k-1}, l_k\,]) = l_1 +\!\!+ [a_1] +\!\!+ \cdots +\!\!+ [a_{k-1}] +\!\!+ l_k$$

We now develop a parallel implementation for a distributed version of *scanrl*, function *pscanrl*, which computes *scanrl* on a plist:

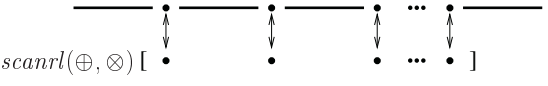
$$scanrl(①, ②) = plist2list \circ pscanrl(①, ②) \circ list2plist_k$$

We introduce the following auxiliary skeletons as higher-order functions on plists, omitting their formal definitions and illustrating them instead graphically:

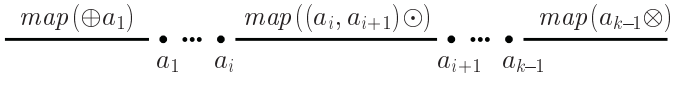
– $pmap\_l\ g$ applies function g, which operates on conventional lists, to all segments of a plist:



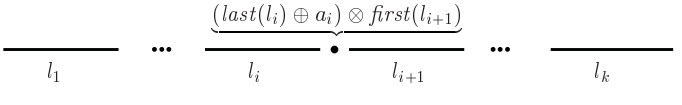
– $pscanrl\_p(\oplus, \otimes)$ applies function $scanrl(\oplus, \otimes)$, defined in (2), to the list containing only the points of the argument plist:



– $pinmap\_l(\odot, \oplus, \otimes)$ modifies each segment of a plist depending on the segment's neighbouring points, using operation $\oplus$ for the left-most segment, $\otimes$ for the right-most segment and operation $\odot$ for all inner segments, where operation $\odot$ is a three-adic operator, i. e. it gets a pair of points and a single point as parameters:



– $pinmap\_p(\oplus, \otimes)$ modifies each single point of a plist depending on the last element of the point's left neighbouring segment and the first element of the point's right neighbouring segment.



The implementations and time complexity analysis of these functions are provided in the next section.

3.2 A Cost-Optimal Implementation of Double-Scan

In this section, we present a theorem that shows that the distributed version of the double-scan skeleton can be expressed using the auxiliary skeletons introduced above. We use here the following definition: a binary operation $\circledast$ is called to be *associative modulo* $\odot$, iff for arbitrary elements a, b, c it holds: $(a \circledast b) \circledast c = (a \odot b) \circledast (b \circledast c)$. Usual associativity is associativity modulo operation *first*, which yields the first element of a pair.

Theorem 2. *Let* ①, ②, ③ *and* ④ *be binary operators, such that* ① *and* ③ *are associative,* $scanrl(①, ②) = scanlr(③, ④)$, *and* ② *is associative modulo* ④. *Moreover, let* ⑤ *be a three-adic operator, such that* $(a, a③c)⑤b = a③(b①c)$. *Then, the double-scan skeleton pscanrl on plists can be implemented as follows:*

$$\begin{aligned} pscanrl(①, ②) \;=\; & pinmap\_l(⑤, ①, ③) \circ pscanrl\_p(①, ②) \\ & \circ pinmap\_p(②, ④) \circ (pmap\_l\; scanrl(①, ②)) \end{aligned} \tag{4}$$

For the theorem's proof, see [2]. To help the user in finding the operator ⑤, we show in [2] how ⑤ can be generated if $(a③)$ is bijective for arbitrary a and if ③ distributes over ①. Since $pscanrl(①, ②) = pscanlr(③, ④)$, equality (4) holds also for $pscanlr(③, ④)$.

Let us analyze the *pscanrl* implementation (4) provided by Theorem 2. On a parallel machine, we partition plists so that each segment and its right "border point" are mapped to a processor. The last processor contains no extra point because there is no point to the right of the last segment in a plist. We further assume that all segments are of approximately the same size.

The right-hand side of (4) consists of four stages executed from right to left, whose parallel time complexity we now study, relying on the graphical representation in Section 3.1:

1. The function $pmap\_l\; scanrl(①, ②)$ can be computed by simultaneously applying $scanrl(①, ②)$ on all processors, provided that the argument plist is partitioned among p processors as described above. Because $scanrl(①, ②) = scanlr(③, ④)$, we can apply either of both on all processors, so the minimum of their runtimes should be taken. Thus, the time complexity is $T_1 = (n/p - 2) \cdot \min\{t_① + t_②, t_③ + t_④\} \in O(n/p)$, where $t_①, \ldots, t_④$ denote the time for one computation with operators ①, ..., ④, respectively.
2. To compute $pinmap\_p(②, ④)$, each processor sends its first element to the preceding processor and receives the first element from the next processor. Then operations ② and ④ are applied to the last element of each processor:

```
/* first buffers 1st element of each processor */
/* first_next receives 1st element of following processor */
MPI_Sendrecv(first, preceding,..., first_next, following,...);
if ( !last_processor ) {
  otwo(last, point);
  ofour(point, first_next);
  /* last denotes the last element of each segment */ }
```

The resulting time complexity is $T_2 = t_s + t_w + t_{②} + t_{④} \in O(1)$, where t_s is the communication startup time and t_w the time needed to communicate one element of the corresponding datatype.

3. As described in Section 3.1, *pscanrl_p*(①, ②) applies *scanrl*(①, ②) on the points of the argument plist, which are distributed across the first $p-1$ processors. Since DS is an instance of the DH skeleton, the generic DH implementation developed in [8] can be used for this step:

```
for (dim=1; dim<p-1; dim<<=1){
  neighbour = my_rank^dim;
  MPI_Sendrecv(data, neighbour, ..., tmp, neighbour, ...);
  if (my_rank < neighbour) oplus( tmp, data);
  else otimes( tmp, data);}
```

   The shown code is a sequence of $\log_2 p$ swaps iterating over the dimensions of the virtual hypercube. A swap consists of pairwise, two-directional communications between neighbouring nodes, followed by a computation in each processor. It takes time $T_3 = \log_2(p-1) \cdot (t_s + 3t_w + t_{\oplus/\otimes}) \in O(\log p)$, $t_{\oplus/\otimes}$ denoting the time for one computation with operator $\oplus$ or $\otimes$ of the DH-skeleton. Operators $\oplus$ and $\otimes$ are defined on triples of values using ①, ... , ④ (see [1]).

4. To compute *pinmap_l*(⑤, ①, ③), each processor sends its last element to the next processor. Then operation ⑤ is applied to the elements of the "inner" processors. The elements of the first processor are manipulated by ①, and the elements of the last processor by ③:

```
/* last_prec receives last element of preceding processor */
MPI_Sendrecv(last, following, ..., last_prec, preceding, ...);

/* elements denotes the datablock of each processor */
if ( first_processor )          // oone will be applied
  for (i=0; i<(m-1); i++)
    oone(elements[i], last);
else if ( last_processor )      // othree will be applied
  for (i=0; i<(m-1); i++)
    othree(last_prec, elements[i]);
else                           // ofive will be applied
  for (i=0; i<(m-1); i++)
    ofive(last_prec, last, elements[i]);
```

   The computations in the processors are mutually independent, which results in a time complexity of $T_4 = t_s + t_w + (n/p - 1) \cdot t_{⑤} \in O(n/p)$, where $t_{⑤}$ denotes the time required by ⑤.

To obtain the overall time complexity of the implementation (4), we sum up the times of the four stages discussed above:

$$
\begin{aligned}
t &= T_1 + T_2 + T_3 + T_4 \qquad (5)\\
&\approx (n/p-1)\cdot(\min\{t_{①}+t_{②}, t_{③}+t_{④}\}+t_{⑤}) + (2+\log_2(p-1))\cdot(t_s+3t_w+t_{\oplus/\otimes})\\
&\in \Theta(n/p+\log p)
\end{aligned}
$$

Recall that, as shown in Section 2, the time complexity of the generic DH-implementation is $\Theta(n/p \cdot \max\{\log(n/p), \log p\})$, so we have substantially improved the asymptotic complexity.

The cost of our new DS-implementation is obviously $\Theta(n + p \cdot \log p)$, which by choosing an appropriate value of p can be made equal to the (linear) cost of DS in the sequential case. Thus, we have proved the following proposition:

Proposition 1. *The parallel implementation* (4) *of the double-scan skeleton is cost-optimal if* $p \in O(n/\log n)$ *processors are used.*

Though in principle a very important characteristic of the quality of parallel implementation, cost optimality characterizes asymptotic behaviour. In practice, the actual performance for a particular application and particular machine are very important. This will be our topic in the next section.

4 Performance Prediction and Case Study

Programming with skeletons offers a major advantage in terms of performance prediction: performance has to be estimated once for a skeleton on a target architecture. In this section, we study how the performance of an application using DS can be predicted based on the estimates for our cost-optimal DS-implementation. This is done by tuning the generic estimate obtained in the previous section to a particular machine and/or application. The order of tuning steps can be chosen depending on the specific goals of the application programmer.

We start with the generic performance estimate for the cost-optimal DS (5) and tune it first to two particular machines, a Cray T3E and a Linux cluster, and then to a particular application case study (tridiagonal system solver).

4.1 Tuning Estimates to Particular Machines

Variables t_s and t_w in (5) 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: $t_w = d \cdot t_{\mathrm{B}}$, 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. $t_{\mathrm{B}} \approx 0.0033\,\mu s$. Inserting these values of t_s and t_{B} into (5), we obtain the following runtime estimate, t_{Cray}, for the double-scan skeleton on a Cray T3E:

$$t_{\text{Cray}} = (n/p-1)\cdot(\min\{t_{①}+t_{②}, t_{③}+t_{④}\}+t_{⑤}) + (2+\log_2(p-1))\cdot(16.4\,\mu s + d\cdot 0.0099\,\mu s + t_{\oplus/\otimes}) \quad (6)$$

On a Linux Pentium IV cluster with SCI interconnect we measured $t_s \approx 25\,\mu s$ and $t_{\text{B}} \approx 0.0123\,\mu s$ for large data sizes. Substituting these values into (5), we obtain the following runtime estimate, t_{Cluster}, for the double-scan skeleton on a Linux cluster:

$$t_{\text{Cluster}} = (n/p-1)\cdot(\min\{t_{①}+t_{②}, t_{③}+t_{④}\}+t_{⑤}) + (2+\log_2(p-1))\cdot(25\,\mu s + d\cdot 0.0369\,\mu s + t_{\oplus/\otimes}) \quad (7)$$

4.2 Tuning Estimates to an Application

By way of an example application, we consider the solution of tridiagonal system of equations, described in detail in [1]. The given system is represented as a list of rows consisting of four values: the value on the main, upper and lower diagonal and the value of the right-hand-side vector. A typical sequential algorithm for solving a tridiagonal system is Gaussian elimination (see e. g. [15,12]) which eliminates the lower and upper diagonal of the matrix in two steps: (1) eliminates the lower diagonal by traversing the matrix from top to bottom according to the *scanl* skeleton using an operator $②_{tds}$; (2) eliminates the upper diagonal of the matrix by a bottom-up traversal, i. e. using the *scanr* skeleton with an operator $①_{tds}$. We can alternatively eliminate first the upper and then the lower diagonal using two other row operators, $③_{tds}$ and $④_{tds}$, see [1] for a formal definition.

Being a composition of two scans, the tridiagonal system solver is a natural candidate to be treated as an instance of the DS skeleton. In [2] we proved that the conditions of Theorem 2 are indeed satisfied, so that our new cost-optimal DS-implementation can be used for the tridiagonal solver. For the solver, implementation (4) operates on lists of rows, which are quadruples of double values, thus having a size of $d = 32$ Bytes. We have measured the operations $①_{tds}, \ldots, ⑤_{tds}, \oplus, \otimes$ of the tridiagonal system solver by executing the operations in a loop over a large array. Operators $\oplus$ and $\otimes$, presented in [2], are the DH operators, defined using $①, \ldots, ④$. Measurement results are presented in Table 1.

Table 1. Measured times in μs for operations of the tridiagonal system solver

architecture	$t_{①}$	$t_{②}$	$t_{③}$	$t_{④}$	$t_{⑤}$	$t_{\oplus/\otimes}$
Cray T3E	0.24	0.36	0.37	0.27	0.43	2.44
Cluster	0.10	0.15	0.15	0.10	0.14	0.73

Inserting operations' times and the value of d into (6) and (7), we obtain:

$$\begin{aligned} t_{\text{Cray-tds}} &= n/p\cdot 1.03\,\mu s + \log_2(p-1)\cdot 19.16\,\mu s + 37.28\,\mu s \\ t_{\text{Cluster-tds}} &= n/p\cdot 0.39\,\mu s + \log_2(p-1)\cdot 26.91\,\mu s + 53.43\,\mu s \end{aligned} \quad (8)$$

The next tuning step is to substitute a particular problem size n into (8): e. g. by substituting $n = 2^{19} \approx 5 \cdot 10^5$, we obtain the following runtime estimate depending on the number of processors:

$$\begin{aligned} T_{\text{Cray-tds-}n=2^{19}} &= 1/p \cdot 0.540\,s + \log_2(p-1) \cdot 19.16\,\mu s + 37.28\,\mu s \\ T_{\text{Cluster-tds-}n=2^{19}} &= 1/p \cdot 0.204\,s + \log_2(p-1) \cdot 26.91\,\mu s + 53.43\,\mu s \end{aligned} \tag{9}$$

Another tuning sequence is to fix the number of processors and vary the problem size. E.g., the estimate for the runtime on 17 processors depending on the problem size is:

$$\begin{aligned} T_{\text{Cray-tds-}p=17} &= n \cdot 0.061\,\mu s + 113.92\,\mu s \\ T_{\text{Cluster-tds-}p=17} &= n \cdot 0.023\,\mu s + 161.07\,\mu s \end{aligned} \tag{10}$$

Here, we assume that one MPI process runs on a processor of a parallel machine.

4.3 Experimental Results

We conducted experiments with the tridiagonal system solver on two machines: (1) a Cray T3E machine with 24 processors of type Alpha 21164, 300 MHz, 128 MB, using native MPI implementation and (2) a Linux cluster with 16 nodes, each consisting of two processors of type Pentium IV, 1.7 GHz, 1 GB, using an SCI port of MPICH.

We are interested in two questions: (1) What is the performance improvement achieved by our cost-optimal implementation as compared with the generic, non-cost-optimal implementation? (2) How well is the performance of the implementation predicted by our analytical model, in particular by formulae (9)–(10)?

The first question is answered by Fig. 1. The cost-optimal solution demonstrates a substantial time improvement compared with the generic, non-cost-optimal version: up to 13 on 17 processors of the Cray T3E and up to 18 on 9 processors of the cluster. Note that we use a logarithmic scale for the time axis.

To answer the second question, we compare in Fig. 1 the predicted runtime using (9) with the measured times for a fixed problem size. The quality of prediction is quite good: the difference to the measured times is less than 12% on both machines.

The number of processors in our generic DS implementation is $p = 2^k + 1$ for some k, because the third step of algorithm (4), *pscanrl_p*(①, ②), applies the generic DH implementation to $p - 1$ points of the argument plist and $p - 1$ is always a power of 2 in the generic DH implementation.

Another interesting point is the absolute speedup, i. e. the parallel version compared to the optimal sequential implementation. Our estimates yield the speedup of approximately $0.6\,p$, which has been confirmed by our measurements.

5 Related Work and Conclusions

Our main contribution is a new, cost-optimal parallel implementation of the DS-skeleton, which is directly applicable for the whole class of applications that

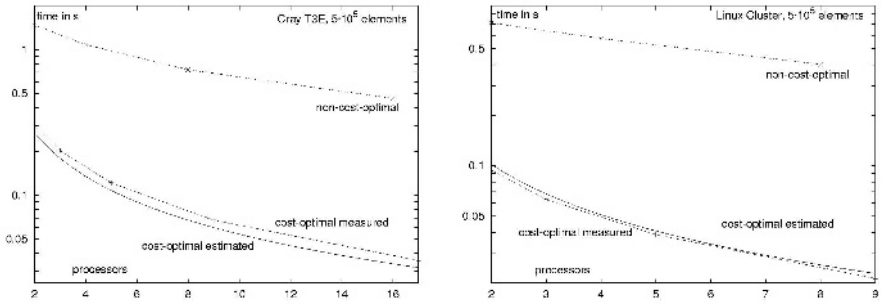


Fig. 1. Runtimes of the tridiagonal solver: cost-optimal vs. non-cost-optimal implementation vs. predicted runtime. Left: on the Cray T3E; Right: on the Linux cluster.

(or their parts) are instances of DS. We have expressed the implementation in MPI to make it portable over different parallel architectures and exploitable in practical skeleton-based programming systems. The cost optimality guarantees not only good runtime but also economical use of processors. We have also formulated a general condition for the cost optimality of the generic implementation of a more general DH skeleton. Cost optimality of divide-and-conquer algorithms was studied earlier in the context of the HDC system [10].

Our implementation makes internal use of the data structure called plist. The plist data structure is new, to the best of our knowledge. A closer look reveals that plists are present, albeit implicitly, in some parallel algorithms: if a block distribution is used, the border points of the blocks being handled specifically, then the data structure used can often be seen as a plist. Our contribution in this respect is in defining and treating explicitly a data structure that traditionally has remained hidden in the design of algorithms.

O'Donnell presented in [14] a bidirectional scan that combines a left and a right scan, independent of each other. By contrast, our DS describes a composition of scans, where the result of the first scan is used by the second scan.

An important advantage of the skeleton approach is good predictability of performance early in the design process. We have demonstrated that the performance of a particular application can be predicted by stepwise tuning of the generic performance estimate of the DS skeleton to a particular target machine and problem size. We have verified experimentally a quite good prediction quality: the error is less than 12%.

The parallelization of our case study – the tridiagonal system solver – is known to be a non-trivial task owing to the sparse structure and restricted amount of potential concurrency [12,13]. It is interesting to observe that our generic DS-implementation, when customized for the tridiagonal solver, is very similar to the implementation by Wang and Mou [17], based on Wang's algorithm [16], which is today probably the solution most widely used in practice. The good speedup of the obtained solution is confirmed by experiments on a Cray T3E and a Linux cluster.

Acknowledgements. We are grateful to the referees, and especially to Christoph A. Herrmann, for improving the original manuscript, to Roman Leshchinskiy and Martin Alt for many fruitful discussions, and to Phil Bacon who helped a great deal in brushing up the presentation.

References

1. Bischof, H., Gorlatch, S.: Double-scan: Introducing and implementing a new data-parallel skeleton. In Monien, B., Feldmann, R., eds.: Euro-Par 2002. Volume 2400 of LNCS., Springer (2002) 640–647
2. Bischof, H., Gorlatch, S., Kitzelmann, E.: The double-scan skeleton and its parallelization. Technical Report 2002/06, Technische Universität Berlin (2002)
3. Botorog, G., Kuchen, H.: Efficient parallel programming with algorithmic skeletons. In Bougé, L., et al., eds.: Euro-Par'96: Parallel Processing. Lecture Notes in Computer Science 1123. Springer-Verlag (1996) 718–731
4. Cole, M.I.: Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation. PhD thesis, University of Edinburgh (1988)
5. Cunha, J.C.: Future generations of problem solving environments. In: Proceedings of the IFIP WG 2.5 Conference on the Architecture of Scientific Software. (2000)
6. Danelutto, M., Pasqualetti, F., Pelagatti, S.: Skeletons for data parallelism in P3L. In Lengauer, C., Griebl, M., Gorlatch, S., eds.: Euro-Par'97. Volume 1300 of LNCS., Springer (1997) 619–628
7. Danelutto, M., Stigliani, M.: SKElib: parallel programming with skeletons in C. In Bode, A., Ludwig, T., Karl, W., Wismüller, R., eds.: EuroPar 2000. Volume 1900 of LNCS., Springer (2000) 1175–1184
8. Gorlatch, S.: Systematic efficient parallelization of scan and other list homomorphisms. In Bougé, L., Fraigniaud, P., Mignotte, A., Robert, Y., eds.: Euro-Par'96: Parallel Processing, Vol. II. Lecture Notes in Computer Science 1124. Springer-Verlag (1996) 401–408
9. Gorlatch, S., Lengauer, C.: Abstraction and performance in the design of parallel programs: overview of the SAT approach. Acta Informatica **36** (2000) 761–803
10. Herrmann, C.A.: The Skeleton-Based Parallelization of Divide-and-conquer Recursions. PhD thesis (2000) ISBN 3-89722-556-5.
11. Herrmann, C.A., Lengauer, C.: HDC: A higher-order language for divide-and-conquer. Parallel Processing Letters **10** (2000) 239–250
12. Leighton, F.T.: Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes. Morgan Kaufmann Publ. (1992)
13. López, J., Zapata, E.L.: Unified architecture for divide and conquer based tridiagonal system solvers. IEEE Transactions on Computers **43** (1994) 1413–1424
14. O'Donnell, J.: Bidirectional fold and scan. In: Functional Programming: Glasgow 1993. Workshops in Computing (1993) 193–200
15. Quinn, M.J.: Parallel Computing. McGraw-Hill, Inc. (1994)
16. Wang, H.H.: A parallel method for tridiagonal equations. ACM Transactions on Mathematical Software **7** (1982) 170–183
17. Wang, X., Mou, Z.: A divide-and-conquer method of solving tridiagonal systems on hypercube massively parallel computers. In: Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing, IEEE Computer Society Press (1991) 810–816