Scalable Computing:
Practice and Experience

Scientific International Journal
for Parallel and Distributed Computing

ISSN: 1895-1767

Universitatea de Vest din Timişoara

Volume 18(1) March 2017

SUBSCRIPTION INFORMATION: please visit http://www.scpe.org

Scalable Computing: Practice and Experience

Volume 18, Number 1, March 2017

TABLE OF CONTENTS

DOI 10.12694/scpe.v18i1.1229
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 18, Number 1, pp. iii–iv. http://www.scpe.org

INTRODUCTION TO THE SPECIAL ISSUE ON PRACTICAL ASPECTS OF HIGH-LEVEL PARALLEL PROGRAMMING

From smartphones to supercomputers, parallel architectures are nowadays pervasive. However parallel programming is still reserved to experienced and trained programmers. The trend is towards the increase of cores in processors and the number of processors in multiprocessor machines: The need for scalable computing is everywhere. But parallel and distributed programming is still dominated by low-level techniques such as send/receive message passing and POSIX threads. Thus high-level approaches should play a key role in the shift to scalable computing in every computer and device.

Algorithmic skeletons (Google's MapReduce being the most well-known skeletal parallelism approach), parallel extensions of functional languages such as Haskell and ML, parallel logic and constraint programming, parallel execution of declarative programs such as SQL queries, meta-programming in object-oriented languages, etc. have produced methods and tools that improve the price/performance ratio of parallel software, and broaden the range of target applications. Also, high level languages offer a high degree of abstraction which ease the development of complex systems. Moreover, being based on formal semantics, it is possible to certify the correctness of critical parts of the applications. The aim of all these languages and tools is to improve and ease the development of applications.

The paper presented in this special issue are extended versions of papers presented at the Practical Aspects of High-Level Parallel Programming (PAPP) workshop that was affiliated to the International Conference on Computational Science (ICCS) from 2004 to 2012, and the Practical Aspects of High-Level Parallel Programming (PAPP) track of the ACM Symposium on Applied Computing (SAC). It also features a paper written specifically for this special issue.

While a lot of work in high-level approaches focus on regular data structures such as arrays and multi-dimensional arrays, the papers of this issue consider more irregular data structures, in particular trees and graphs.

Kaheki, Matsuzaki and Emoto present a new approach for parallel reduction on trees in a distributed memory setting. Their algorithm is based on a suitable serialization of trees and its efficiency is shown for the Bulk Synchronous Parallel (BSP) model. Matzusaki proposes a set of efficient skeletons on distributed trees: their efficient implementation relies on the segmentation of trees based on the idea of m-bridges that ensures high locality, and allows to represent local segments as serialized arrays for high sequential performance.

Dazzi, Carlini, Lulli and Ricci propose Telos, a high-level approach for large graphs processing based on overlay networks, and an implementation on top of Apache Spark.

Hajibaba, Gorgin, and Sharifi present a sequence similarity parallelization technique in the context of another Apache project: Storm a stream processing framework.

I wish to thank Anne Benoît and Fré dé ric Gava, co-chairs of some the previous PAPP workshops and the program committee members of the PAPP workshops and the PAPP ACM SAC track who reviewed the initial papers and the revised versions of this special issue.

Frédéric Loulergue
Northern Arizona University
School of Informatics, Computing and Cyber Systems
Flagstaff, Arizona, USA

DOI 10.12694/scpe.v18i1.1230
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 18, Number 1, pp. 1–15. http://www.scpe.org

EFFICIENT PARALLEL TREE REDUCTIONS ON DISTRIBUTED MEMORY ENVIRONMENTS *

KAZUHIKO KAKEHI† KIMINORI MATSUZAKI‡ AND KENTO EMOTO§

Abstract. A new approach for fast parallel reductions on trees over distributed memory environments is presented. The start point of our approach is to employ the serialized representation of trees. Along this data representation with high memory locality and ease of initial data representation, we developed an parallelized algorithm which shares the essence with the parallel algorithm for parentheses matching problems. Our algorithm not only is proven to be theoretically efficient, but also has a fast implementation in a BSP style.

Key words: Parallel tree reduction, parentheses matching, serialized representation, the tree contraction algorithm, Bulk Synchronous Parallelism.

AMS subject classifications. 68W10, 05C05

1. Introduction. Research and development of parallelized algorithms have been intensively done toward matrices or one dimensional arrays. Looking at recent trends in applications, another data structure has also been calling for efficient parallel treatments: the tree structures. Emergence of XML as a universal data format, which takes the form of a tree, has magnified the impact of parallel and distributed mechanisms toward trees in order to reduce computation time and mitigate limitation of memory.

Consider, as a simple and our running example, a computation *maxPath* to find the maximum of the values each of which is a sum of values in the nodes from the root to each leaf. When it is applied to the tree at the left of Fig. 2.2, the result should be 12 contributed by the path of values 3, −5, 6 and 8 from the root. Recalling the research on parallel treatments on trees, *the parallel tree contractions algorithm*, first proposed by Miller and Reif [34], have been known as one of the most fundamental techniques to realize parallel computation over trees efficiently. One attractive feature of parallel tree contractions is that no matter how imbalanced a tree is, the tree can be reduced in parallel to a single node by repeatedly contracting edges and merging adjacent nodes. This theoretical beauty, however, may not necessarily shine in practice, because of the following two source of problems.

First, parallel tree contractions, originally being developed under the assumption of shared memory environments, have been intensively studied in the context of the PRAM model of parallel computation. This assumption does not apply to the recent trends of popular PC clusters, a common and handy approach for distributed memory environments, where the treatment and cost of data arrangement among processors need taking into account. It should be noted that efficient tree contractions under PRAM model are realized by assigning contractions to different processors each time.

Second, the parallel tree contractions algorithm assumes that the tree structures are kept as linked structures. Such representations using links are not suitable for fast execution, since linked structures often do not fit in caching mechanism, and it is a big penalty on execution time under current processor architectures.

This paper gives a clear solution for parallel tree reductions with its start point to use *serialized forms of trees*. Their notable examples are the serialized (streamed) representations of XML or parenthesized numeric expressions which are obtained by tree traversals. The problems mentioned above are naturally resolved by this choice, since distribution of serialized data among processors and realization of routines running over them with high memory locality are much simpler than that of trees under linked structures.

Our algorithm over the serialized tree representations is developed along with the *parentheses matching*

*An earlier version of this paper appeared in PAPP2007, part of ICCS2007[23].

†Academic Co-Innovation Division, UTokyo Innovation Platform Co., Ltd., 3–40–10 Hongo, Bunkyo-ku, Tokyo 113-0033 Japan (k.kakehi@utokyo-ipc.co.jp)

‡School of Information, Kochi University of Technology, 185 Tosayamadacho-Miyanokuchi, Kami, Kochi 782–8502 Japan (matsuzaki.kiminori@kochi-tech.ac.jp)

§Faculty of Computer Science and Systems Engineering, Kyushu Institute of Technology, 680-4 Kawazu, Iizuka, Fukuoka 820-8502, Japan (emoto@ai.kyutech.ac.jp)

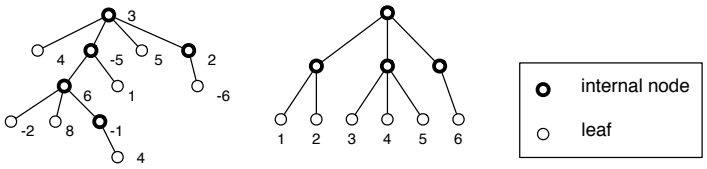


FIG. 2.1. *A rose tree (left), and the rose tree representation of* $[[1,2],[3,4,5],[6]]$ *(right)*

problems. As instances of serialized trees, parallelization of parentheses matching problems, which figures out correspondence between brackets, have plenty of work ([4, 36, 13, 25, 21] for example). Our algorithm, with good resemblance to the one under BSP [43], also has a BSP implementation with three supersteps.

The contributions of our work are briefly summarized as follows:

- *A New viewpoint at the parallel tree contractions algorithm—connection to parentheses matching*: We cast a different view on computing tree structures in parallel. The employed data representation, a serialized form of trees by tree traversals, has massive advantages in the current computational environments: namely XML as popular data representations, cache effects in current processor architectures, and ease in data distribution among popular PC clusters. Tree computations over its serialized representation has much in common with parentheses matching. It is common to see that this problem has connection with trees, like binary tree reconstruction or computation-tree generation, but to the best of our knowledge we are the first to apply the idea of parentheses matching toward parallel tree reductions, and to prove its success.
- *Theoretical and practical efficiency*: The previous work of parallel tree contractions on distributed memory environments, namely hypercube [32] or BSP [16], both of which require $O(N/P \log P)$ execution time. Contrasting to other work, our approach employs serialized tree representations. As a result, we realized an $O(N/P + P)$ algorithm.

This paper is organized as follows. After this Introduction Sect. 2 observes our tree representations and tree reductions. Section 3 explains an additional condition which enables efficient parallelization. We show that this condition is equivalent to that of the parallel tree contractions algorithm. Section 4 develops the parallelized algorithm. Our algorithm consists of three phases, where the first two perform computation along with parentheses matching, and the last reduces a tree of size less than twice of the number of processors. Section 5 supports our claims by some experiments. They demonstrate good scalability in computation. On the other hand, we also observe fluctuation of data transactions. We discuss the related work in Sect. 6. Finally we conclude this paper in Sect. 7 with mentioning future directions.

2. Preliminaries. This section defines the target of our parallelization, namely tree structures and their serialized form, and tree homomorphism.

2.1. Trees and their serialized representation. We treat trees with unbound degree (trees whose nodes can have an arbitrary number of subtrees), which is often referred as "*rose trees*" in functional programming communities.

DEFINITION 2.1. *A data structure RTree is called* rose tree *defined as follows.*

$$RTree\ \alpha = Node\ \alpha\ [Rtree\ \alpha]$$

Fig. 2.1 shows examples of rose trees The left tree is set to be used as our running example. Rose trees are general enough to represent nested lists; the right part of Fig. 2.1 is what $[[1,2],[3,4,5],[6]]$ is formatted into a rose tree, whose internal nodes do not have their value. Binary trees are also covered by limiting degrees of each node to be either zero or two.

As was explained in Introduction, our internal representation is to keep tree-structured data in a serialized manner like XML. For example, if we are to deal with a tree with just two nodes, parent `a` and its child `b`, the serialized equivalent is a list of four elements [`<a>`,`<b>`,`</b>`,`</a>`] . This is a combination of a preorder traversal (for producing the open tag `<a>` and then `<b>`) and a postorder traversal (for producing the close tag `</b>` and `</a>` afterwards). This representation has information enough to obtain back the original tree. In

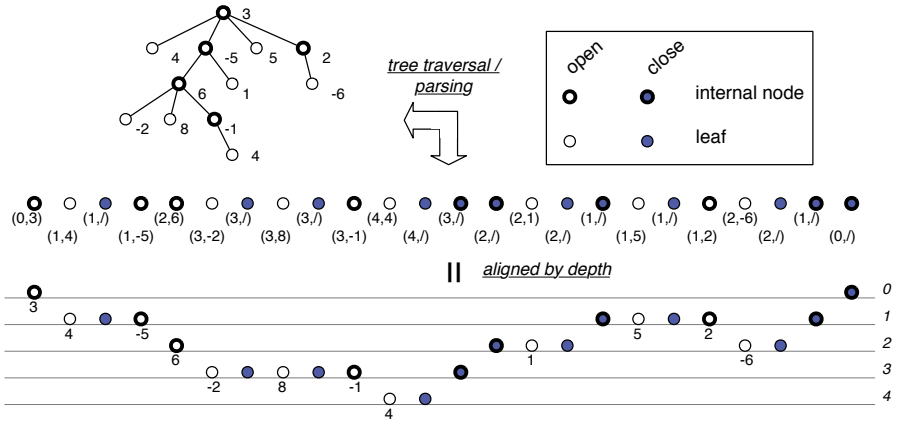


FIG. 2.2. *A rose tree (upper left), its serialized representation as a sequence of pairs (middle) and another representation according to the depth (lower)*

this paper we treat only *well-formed* serialized representation, which is guaranteed to be parsed into trees. To make such serialized representation easy to observe, we assume (1) to ignore information in the closing tag, and (2) to assign the information about the depth in the tree instead. We therefore assume to deal with a list of pairs [(0,a), (1,b), (1,/), (0,/)], where / denotes closing tags. The associated depth information can be easily obtained by prefix sum computation [5].

The sequence of the middle in Fig. 2.2 is our internal representation of the example tree. We later see the information of depth helps us to have insight for deriving parallelized algorithms. Following these figures, list elements describing the entrance (which corresponds to open tags) are called *open*, and ones describing the exit (close tags) are called *close*. These close elements without any value can be soon wiped away when we perform computation.

2.2. Tree homomorphism. Algorithms are often tightly connected with the data structure. A general recursive form naturally arises here, which we call *tree homomorphism* [42].

DEFINITION 2.2. *A function h defined as follows over rose trees is called* tree homomorphism.

$$\begin{aligned} h(\ \mathit{Node}\ a\ [t_1, \ldots, t_n]\ ) &= a \oplus (\ h(t_1) \otimes \cdots \otimes h(t_n)\ ) \\ h(\ \mathit{Leaf}\ a\ ) &= h'(a) \end{aligned}$$

We assume the above operator $\otimes$ is associative and has its unit $\iota_\otimes$.

This definition consists of a function h' for leaves, and two kinds of computation for internal nodes: $\otimes$ to reduce recursive results from subtrees into one, and $\oplus$ to apply the result to the internal node. For later convenience, we define $\ominus$ as $(a, b, c) \ominus e = a \oplus (b \otimes e \otimes c)$, and we call (a, b, c) appearing at the left of $\ominus$ a "*triple*". The computation *maxPath* mentioned in Introduction is also a tree homomorphism.

$$\begin{aligned} \mathit{maxPath}(\ \mathit{Node}\ a\ [t_1, \ldots, t_n]\ ) &= a + (\ \mathit{maxPath}(t_1) \uparrow \cdots \uparrow \mathit{maxPath}(t_n)\ ) \\ \mathit{maxPath}(\ \mathit{Leaf}\ a\ ) &= \mathit{id}(a) \\ &= a \end{aligned}$$

Here, *id* is the identify function, and $\uparrow$ returns the bigger of two numbers whose unit is $-\infty$. When it is applied to the tree in Fig. 2.2, the result should be $12 = 3 + (-5) + 6 + 8$. For other examples please see [27].

3. Parallelization Condition. When we develop parallel implementations we often utilize associativity of computations, in order to change the order of computation with guaranteeing the same result. The framework of tree homomorphism only requires the condition that the operator $\otimes$ is associative. This property of the *horizontal* directions is indeed necessary for parallelization, but not yet sufficient: we currently lack the effective measure toward computation of the *vertical* directions.

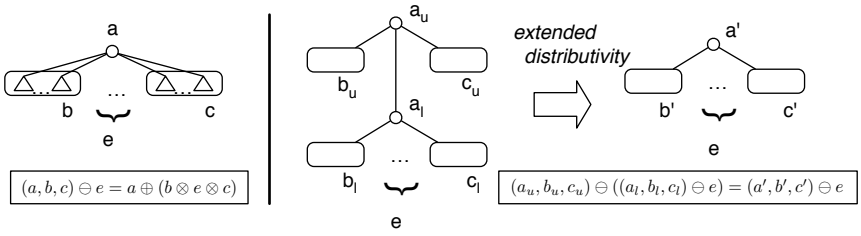


FIG. 3.1. *A* triple *(left) and extended distributivity (right)*

Before we start algorithm development, this section observes an additional property. What we use is known as *extended distributivity* [27]. This property is explained using the operator $\ominus$ as follows.

DEFINITION 3.1. *The operator* $\otimes$ *is said* extended distributive *over* $\oplus$ *when the following equation holds for any* a_u, b_u, c_u, a_l, b_l, c_l, *and* e:

$$\begin{aligned}(a_u,b_u,c_u) \ominus ((a_l,b_l,c_l) \ominus e) &= a_u \oplus (b_u \otimes (a_l \oplus (b_l \otimes e \otimes c_l)) \otimes c_u) \\ &= a' \oplus (b' \otimes e \otimes c') \\ &= (a',b',c') \ominus e\end{aligned}$$

with appropriate functions p_a, p_b *and* p_c *which calculate*

$$\begin{aligned}a' &= p_a(a_u,b_u,c_u,a_l,b_l,c_l)\ , \\ b' &= p_b(a_u,b_u,c_u,a_l,b_l,c_l)\ ,\ \textit{and} \\ c' &= p_c(a_u,b_u,c_u,a_l,b_l,c_l)\ .\end{aligned}$$

Efficient parallel tree reductions require these properties as well as $\otimes$ and $\oplus$ to be constant-time. Our running example satisfies them, implicitly with its characteristic functions $a' = a_u + a_l$, $b' = b_u - a_l \uparrow b_l$, $c' = c_l \uparrow c_u - a_l$, as the following calculation shows.

$$\begin{aligned}&(a_u,b_u,c_u) \ominus (\ (a_l,b_l,c_l) \ominus e\ ) \\ &= a_u + (b_u \uparrow (a_l + (b_l \uparrow e \uparrow c_l)) \uparrow c_u) \\ &= (a_u + a_l) + ((-a_l + b_u \uparrow b_l) \uparrow e \uparrow (c_l \uparrow -a_l + c_u)) \\ &= (\ (a_u + a_l),\ (-a_l + b_u \uparrow b_l),\ (c_l \uparrow -a_l + c_u)\ ) \ominus e\end{aligned}$$

The *triple* and extended distributivity from the viewpoint of their tree structures are depicted in Fig. 3.1. We show the property of extended distributivity is equivalent to the condition for realizing the parallel tree contractions algorithm with respect to rose trees. Parallel tree contractions require two operations *rake* and *compress* are efficiently computed. The operation *rake* is to contract an edge between a leaf node and its parent internal node; the other operation *compress* is to contract the last remaining edge of an internal node. Once these treatments are possible, tree computations are performed in parallel (see Fig. 3.2).

THEOREM 3.2. *Assume computation of* $\oplus$ *and* $\otimes$ *requires constant time. Extended distributivity is equivalent to the conditions for parallel tree contractions with respect to rose trees.*

Proof. [$\Leftarrow$] We are treating rose trees whose nodes have unbound degree. We assume leaves from outer positions into inner positions are gradually raked. A node with its value a can be regarded as $(a, \iota_\otimes, \iota_\otimes)$. Take its leftmost subtree. Let b be its computed value, and y the computed value of the remaining siblings appearing on its right. The reduced value of the subtrees under a is therefore $b \otimes y$. The rake operation is to merge $(a, \iota_\otimes, \iota_\otimes)$ with b, and returns $(a, b, \iota_\otimes)$ using associativity of the operator $\otimes$.

$$\begin{aligned}(a,\iota_\otimes,\iota_\otimes) \ominus (b \otimes y) &= a \oplus (\iota_\otimes(b \otimes y) \otimes \iota_\otimes) \\ &= a \oplus (b \otimes y \otimes \iota_\otimes) \\ &= (a,b,\iota_\otimes) \ominus y\end{aligned}$$

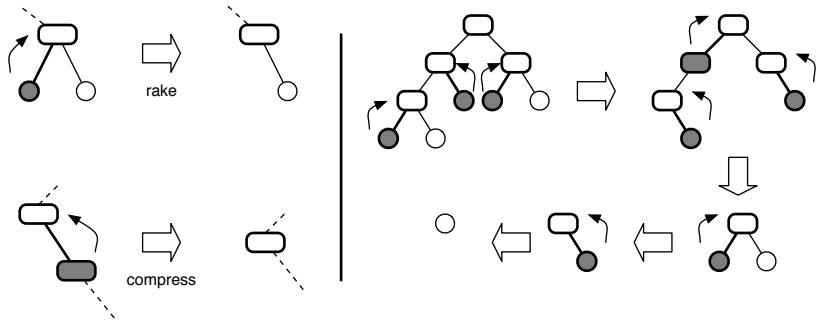


FIG. 3.2. *Tree contractions* rake *(upper left) and* compress *(lower left), and an example of parallel tree contractions (right)*

ROUTINE 4.1. *First phase applied to each fragment*

Input: Sequence $(d_0, a_0), \ldots, (d_{n-1}, a_{n-1})$ $(n \geq 1)$.
Variables: Arrays as, bs, cs (behaving as stacks growing from left to right), an integer d, and a value t.

(1) Set $d \leftarrow d_0$. If a_0 is "/" then $cs \leftarrow [\iota_\otimes, \iota_\otimes]$, $as \leftarrow [\,]$, $bs \leftarrow [\,]$; else $cs \leftarrow [\iota_\otimes]$, $as \leftarrow [a_0]$, $bs \leftarrow [\iota_\otimes]$.
(2) For each i in $\{1, \ldots, n-1\}$
  (2-a) if a_i is not "/" (namely a value), then push a_i to as and $\iota_\otimes$ to bs;
  (2-b) else if as is empty, then push $\iota_\otimes$ to cs, and set $d \leftarrow d_i$;
  (2-c) else pop a' from as and b' from bs.
    if $b' = \iota_\otimes$ (implying a' is a leaf) then $t \leftarrow h'(a')$; else $t \leftarrow a' \otimes b'$;
    if bs is not empty, then pop b'' from bs and push $b'' \otimes t$ to bs; else pop c'' from cs and push $c'' \otimes t$ to cs.
(3) If $\mathrm{P} \neq 1$ then remove $\iota_\otimes$ at the bottom of cs_0 and $cs_{\mathrm{P}-1}$.

In terms of the value c of its rightmost leaf, the rake operation similarly succeeds to produce $(a, \iota_\otimes, c)$. The same arguments apply to the cases in which there are already raked values, namely (a, b, c) in general.

The compress operation is to merge a node (a_u, b_u, c_u) with its subtree whose computed value can be written as $(a_l, b_l, c_l) \ominus e$. This is what extended distributivity deals with, and we successfully have $(a', b', c') = (a_u, b_u, c_u) \ominus ((a_l, b_l, c_l) \ominus e)$.

[⇒] When extended distributivity holds, we have the implementation of the tree homomorphism based on tree contractions over a binary tree representation of rose trees [27]. □

4. Parallelized Algorithm. This section develops a parallel algorithm for tree homomorphism which satisfies extended distributivity. Our algorithm consists of three phases: (1) the first phase applies tree homomorphism toward segments (consecutive subsequences) as much as possible which is distributed to each processor; (2) after communications among processors the second phase performs further reduction using extended distributivity, producing a binary tree as a result whose internal nodes are specified as *triple*s, and size is less than twice of the number of processors; finally (3) the third phase reduces the binary tree into a single value.

As was used in Introduction, we set to use N to denote the number of nodes in the tree we apply computations to, and P the number of available processors. The serialized representation has therefore 2N elements. During the explanation we assume $\mathrm{P} = 4$. Each processor is assumed to have $\mathrm{O}(\mathrm{N}/\mathrm{P})$ local memory, and to be connected by a router that can send messages in a point-to-point manner. Our algorithm developed here involves all-to-all transactions; such a mechanism is available in MPI on many PC clusters. We assume BSP model [43, 33].

4.1. First phase. Each processor applies tree homomorphism to its given segment of size $2\mathrm{N}/\mathrm{P}$. The process is summarized as Routine 4.1. This process leaves fragments of results, in arrays as_i, bs_i, cs_i and an integer d_i for each processor number i. The array as_i is to keep the open elements without their corresponding close element. Each element of as_i can have subtrees before the next one in as_i, and their reduced values are kept

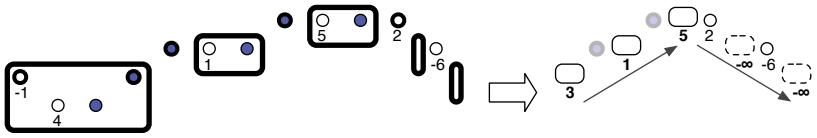
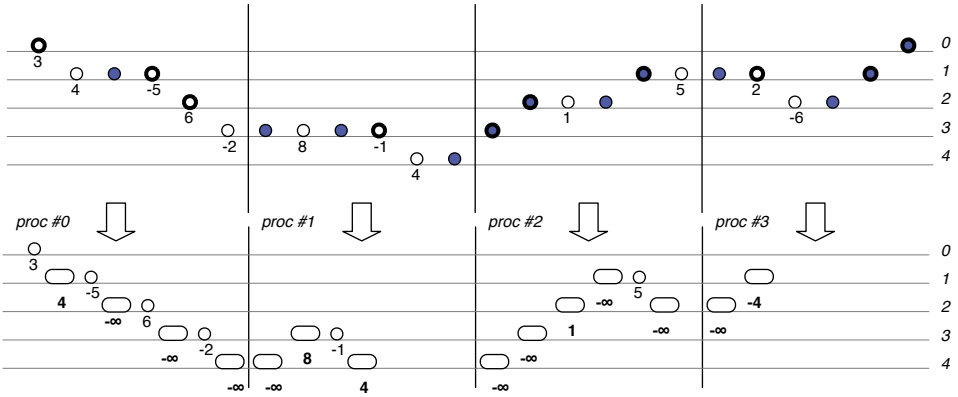


FIG. 4.1. *An illustrating example of the first phase—applying tree homomorphism maxPath to a segment of Fig. 2.2 (lined and dashed ovals indicate values obtained by reduction and $-\infty$ (the unit of $\uparrow$), respectively)*



$cs_0 = [\,]$	$cs_1 = [-\infty, 8]$	$cs_2 = [-\infty, -\infty, 1, -\infty]$	$cs_3 = [-\infty, -4]$
$as_0 = [3, -5, 6, 2]$	$as_1 = [-1]$	$as_2 = [5]$	$as_3 = [\,]$
$bs_0 = [4, -\infty, -\infty, -\infty]$	$bs_1 = [4]$	$bs_2 = [-\infty]$	$bs_2 = [\,]$
$d_0 = 0$	$d_1 = 3$	$d_2 = 1$	$d_3 = 0$

FIG. 4.2. *The results by First phase (upper: illustration, lower: data)*

in bs_i. Similar treatments are done to unmatched close elements, leaving values in cs_i (we remove unmatched close elements thanks to the absence of values). The integer d_i denotes the shallowest depth in processor i. While both of elements in as_i and bs_i are listed in a descending manner, those of cs_i are in ascending manner; the initial elements of as_i and bs_i and the last one of cs_i are at height d_i (except for cs_0 and $cs_{\mathrm{P}-1}$ whose last element at depth 0 is always $\iota_\otimes$ and therefore is set to be eliminated).

In Fig. 4.1 we show a case of the illustrating segment from the 10th element $(3, -1)$ to the 21st $(2, -6)$ of the sequence in Fig. 2.2. We have, as depicted:

$$\begin{array}{rclcrcl} cs & = & [-1 + id(4),\ id(1),\ id(5)] & \qquad & as & = & [2, -6], \qquad\qquad d \ = \ 1. \\ & = & [3, 1, 5], & & bs & = & [-\infty, -\infty], \end{array}$$

Please note that we regard absence of subtrees as an empty forest to which the tree homomorphism returns $-\infty$, the unit of $\uparrow$ (at depth 2 and 3 kept in bs). When we distribute the whole sequence in Fig. 2.2 evenly among four processors (6 elements for each), the results by this phase is shown in Fig. 4.2.

4.2. Second phase. The second phase matches data fragments kept in each processor into *triple*s (a, b, c) using communication between processors. Later, we reduce consecutive occurrences of *triple*s into a value, or into one *triple* by extended distributivity.

When we look carefully at Fig. 4.3, we notice that 3 in as_0 at depth 0 now has five parts at depth 1 as its children: the value 4 in bs_0, a subtree spanning from processors 0 to 2 whose root is -5 in as_0, the value $-\infty$ in cs_2, a subtree from processor 2 to 3 whose root is 5 in as_2, and the value -4 in cs_3. As these subtrees need reducing separately, we focus on the leftmost and the rightmost values in bs_0 and cs_3 (we leave the value $-\infty$ in cs_2 for the time being). We notice that the *group* of the value 3 in as_0 with these two values in processors 0 and 3 forms a *triple* $(3, 4, -4)$.

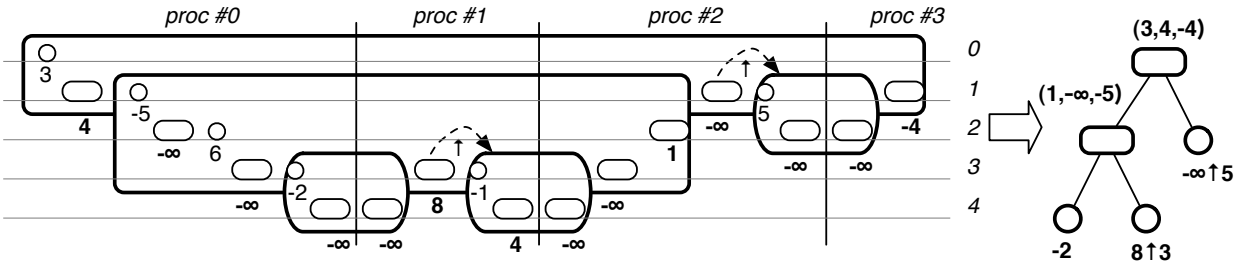


FIG. 4.3. Triple*s between two processors (left) and the resulting tree (right) after the second phase*

Similarly, two elements in as_0 at depth 1 and 2, with two elements each in bs_0 and cs_2 at depth 2 and 3, respectively, form two *triple*s $(-5, -\infty, 1)$ and $(6, -\infty, -\infty)$. The former *triple* indicates a tree that awaits the result of one subtree specified by the latter. This situation is what extended distributivity takes care of, and we can merge two *triple*s (a sequence of *triple*s in general) into one:

$$\begin{aligned}
&(-5, -\infty, 1) \ominus (\ (6, -\infty, -\infty) \ominus e\ ) \\
&\quad = \quad (\ (-5+6),\ (-6-\infty \uparrow -\infty),\ (-\infty \uparrow -6+1)\ ) \ominus e \\
&\quad = \quad (1, -\infty, -5) \ominus e
\end{aligned}$$

for any e. In this way, such *group*s of data fragments in two processors turn into one *triple*.

*Group*s from two adjacent processors are reduced into a single value without any missing subtrees in between. Instead of treating using extended distributivity, the values -2 in as_0 and -1 in as_1 at depth 3, and 5 in as_2 at depth 2 with their corresponding values in bs_i and cs_{i+1} $(i = 0, 1, 2)$ turn into values $id(-2) = -2$, $-1 + (4 \uparrow -\infty) = 3$, $id(5) = 5$, respectively.

We state the following lemma to tell the number of resulting *group*s in total.

LEMMA 4.1. *Given p processors. The second phase produces* group*s of the number at most $2p - 3$.*

Proof. For simplicity we first assume the shallowest depths d_i for $0 < i < p - 1$ are disjoint for each other (both d_0 and d_{p-1} are 0). Under this assumption the equality $R_p = 2p-3$ holds. Proof is given by mathematical induction.

Consider the case $p = 2$. We immediately see that $R_2 = 1 = 2 \cdot 2 - 3$. Assume $R_i = 2i - 3$ holds for $2 \le i < p$, and we have p processors each of which holds the results by Routine 4.1. First, we need to observe that $d_0 = d_{p-1} = 0$ and $d_0 < d_i$ for $0 < i < p - 1$, since we deal with well-formed trees only. By the assumption of disjoint d_i we can find $0 < i < p$ such that $d_i > d_j$ for $0 < j < p$, $i \neq j$. Using the height d_i, we find a *group* between processors 0 and p; we continue investigation of *group*s for each of $i + 1$ processors (from 0 to i) and $p - i$ processors (from i to $p - 1$) with their initial height d_i instead of 0. Hence the following holds, using induction hypothesis.

$$\begin{aligned}
R_p &= 1 + R_{i+1} + R_{p-1-i+1} \\
&= 1 + (2(i+1) - 3) + (2(p-i) - 3) \\
&= 2p - 3
\end{aligned}$$

Inequality appears in case there appear the same shallowest depths. For example, assume there are four processors, and processors 1 and 2 have the same peak depth, namely $d_1 = d_2 > d_0 = d_3 = 0$. In this case, the *group*s are created between processors 0 and 3, 0 and 1, 1 and 2, 2 and 3. This partitioning produces 4 *group*s, one smaller than $5 = 2 \cdot 4 - 3$. Generalization of this argument proves $R_P \le 2P - 3$. □

This lemma guarantees that, the number of *group*s this phase produces is less than twice of the number of processors. Notice that the *group*s form a tree (Fig. 4.3, right). This observation enables us to have another explanation on the number of *group*s. Given p processors, we soon notice that there exist *group*s between two adjacent processors. They are regarded as leaves, without any missing subtrees in between. The number of

ROUTINE 4.2. *Second phase computing groups among processors.*

Input: Sequence (p, d_p) is given in the ascending order of p.
Variables: A stack is used whose top is referred as (p_s, d_s).

(1) Push the first pair $(0, 0)$ on a stack
(2) For each i in $\{1, \ldots, \mathrm{P} - 1\}$
  (2-1) prepare a variable $d \leftarrow \infty$.
  (2-2) while $d_i < d_s$, produce $M^{p_s \leftrightarrow i}_{[d_s, d]}$, set $d \leftarrow d_s$ and pop from the stack;
  (2-3) if $d_i = d_s$, then produce $M^{p_s \leftrightarrow i}_{[d_s, d]}$ and $M^{\overleftrightarrow{-}}_{[d_i, d_s]}$, and pop from the stack; else produce $M^{p_s \leftrightarrow i}_{[d_i, d]}$.
  (2-4) push (i, d_i).
(3) Finally eliminate the last mating pair (that is $M^{\overleftrightarrow{-}}_{[0,0]}$).

leaves are therefore one smaller than the processor numbers, namely $p - 1$. Other *group*s behave as *internal nodes* which have two or more subtrees (otherwise we can simplify two *group*s of direct filiation into one *group* using extended distributivity). In case the number of leaves are fixed, binary trees have the largest number of internal nodes among trees in general. The number of internal nodes in a binary tree is one smaller than that of leaves, and if there are $p - 1$ leaves we have $p - 2$ internal nodes. Hence the number of *group*s is no more than $2p - 3$.

The Routine 4.2 figures out *group*s among processors. $M^{p_l \leftrightarrow p_r}_{[d_u, d_l]}$ denotes a *group* between processors p_l and p_r whose data fragments span from the depth d_u until d_l (∞ in d_l indicates "everything starting from d_u"). This Routine inserts $M^{\overleftrightarrow{-}}_{[d_u, d_l]}$ as a dummy *group* in case the same d appear among more than two consecutive processors. It is assumed to be reduced into $\iota_\ominus$, a virtual left unit of $\ominus$ (namely $\iota_\ominus \ominus e = e$). This routine produces $2\mathrm{P} - 3$ *group*s in the *post-order* traversal over the binary tree.

The remaining task in this phase is to perform data transactions of as_i, bs_i, cs_i among processors according to the *group*s information $M^{p_l \leftrightarrow p_r}_{[d_u, d_l]}$, and apply further computation toward each *group*. There can be a couple of approaches of data transaction. One simple idea is to transfer fragments of cs_i to their corresponding processors. When we apply computation for each *group*, the computed value at the shallowest depth d_p in each processor are associated to the *group* on their right for later computation by $\otimes$ (8 in cs_1, $-\infty$ in cs_2; see Fig. 4.3).

4.3. Third phase. This last phase compiles obtained *triple*s or values and reduce them into a single value. As Fig. 4.3 shows, the obtained *triple*s and values in the previous phase form a binary tree of size $2\mathrm{P} - 3$ (including dummies by $M^{\overleftrightarrow{-}}_{[0,0]}$). We collect *triple*s or values in one processor, and apply tree reduction in $\mathrm{O}(\mathrm{P})$ time.

4.4. Cost estimation. The whole procedures are summarized as Algorithm 4.1. As the summary of this section, we estimate the cost of this algorithm.

C_h, C_{ed}, and C_t are, respectively, the computational cost of tree homomorphism (using h', $\otimes$ and $\oplus$), that of merging two *triple*s into one by extended distributivity using its characteristic functions, and that of *group*s generation by Routine 4.2. The length of an array xs is written as $|xs|$. The cost ratio of communication compared to computation is written as g, and L is the time required for barrier synchronization among all processors.

The cost is summarized in Table 4.1. Step (1) takes time linear to the data size in each processor. The memory requirement of this sequential procedure (Routine 4.1) is $\mathrm{O}(\mathrm{N}/\mathrm{P})$ as well. The worst case in terms of the size of the results occurs when an sequence of only open (or close) elements is given, resulting in two arrays as_i and bs_i (or cs_i) of the length $2\mathrm{N}/\mathrm{P}$ for each. Step (2-2) takes $\mathrm{O}(\mathrm{P})$ time and space.

Step (2-3) requires detailed analysis. The cost depends on how each processor sends out its fragment of data. This depends on the size of cs_i, namely $|cs_i|$. When we analyze the worst case, it is possible that a processor sends out all of its cs_i whose size can be at most $2\mathrm{N}/\mathrm{P}$. Therefore the worst cost is estimated as $g \cdot 2\mathrm{N}/\mathrm{P}$, and this is $\mathrm{O}(\mathrm{N}/\mathrm{P})$.

Similar analysis applies to Step (2-4), by changing the viewpoint from the transmitter to the receiver. It is often the case that the computational cost C_{ed} is heavier than C_h. The worst case occurs when the length of

ALGORITHM 4.1. *The whole procedure of tree reduction.*

Input: Assume the serialized representation of a tree of size N (the length of the list is 2N) is partitioned into P sublists, which are distributed among processors $0, \ldots, P-1$.

First phase:

(1) Each processor sequentially performs tree computation using Routine 4.1, and produces arrays as_i, bs_i, cs_i and the shallowest depth d_i .

Second phase:

(2-1) All values of d_i are shared through global communication using all-to-all transactions.

(2-2) Each processor performs Routine 4.2 to figure out the structures *group*s have.

(2-3) Each processor transmits fragments of cs_i to their corresponding processor according to the information obtained in Step (2-2).

(2-4) Each processor reduces *group*s into single values or single *triple*s.

Third phase:

(3-1) Values and *triple*s obtained in Step (2-4) are collected to processor 0.

(3-2) Processor 0 reduces the binary tree into the result.

TABLE 4.1
Cost estimation of our algorithm under BSP

Step	computation	communication	synch.
(1)	$C_1 \cdot 2N/P$		
(2-1)		$g \cdot P$	L
(2-2)	$C_3 \cdot P$		
(2-3)		$\max_i\{g \cdot \lvert cs_i\rvert\}$	L
(2-4)	$\leq\ C_{ed} \cdot \max_i\{\lvert as_i\rvert\}$		
(3-1)		$\leq\ g \cdot P$	L
(3-2)	$C_h \cdot (\ (2P-3)-1\ )$		

as_i is $2N/P$, and when the *group*s in that processor behave as internal nodes, requiring computation of extended distributivity.

After Step (2-4) we have a binary tree of size $2P-3$ whose nodes are distributed among P processors. The processor 0 collects these results and applies the final final reduction. The cost for the final computation (3-2) is therefore $O(P)$.

We conclude this section by stating the following theorem.

THEOREM 4.2. *Tree homomorphism with extended distributivity has a BSP implementation with three supersteps of at most* $O(P + N/P)$ *communication cost for each.*

5. Experiments. We performed experiments using our implementation using C++ and MPI. The environment we used was semi-uniform PC clusters which consist of 2.4GHz or 2.8GHz processors with 2GB memory each and are connected with Gigabit Ethernet. The compiler and MPI library are gcc 4.1.1 and MPICH 1.2.7. We tested the efficacy of our algorithm using two kinds of experiments. The first is a querying operations over given trees. These computations are specified as a tree homomorphism using operations over matrices of size 10×10. The second is what we have seen as the running example, namely *maxPath*. The former examines the cases where heavy computation is involved, while the latter with simple computations will exhibit the communication costs which our algorithm introduces. We prepared trees of size 1,000,000 in three types, namely (F) a flat tree, (M) a monadic tree, and (R) 10 examples of randomly generated trees. A flat tree of size n is a tree with $n-1$ leaves just below the root. A monadic tree is a tree-view of a list, and has internal nodes with just one subtree for each. We executed the program over each type using 2^i processors ($i = 0, \ldots, 6$). We

TABLE 5.1
Execution times of the first experiments (querying, time in seconds)

	Random Trees (R)				Flat (F)		Monadic (M)	
p	dist.	comp.	min.	max.	dist.	comp.	dist.	comp.
1	0.088	5.546	5.494	5.613	0.090	3.833	0.109	11.924
2	0.210	2.814	2.753	2.892	0.208	1.886	N.A.	N.A.
4	0.252	1.493	1.469	1.527	0.250	1.037	0.248	12.474
8	0.308	0.751	0.736	0.762	0.301	0.529	0.301	6.810
16	0.339	0.377	0.367	0.381	0.354	0.287	0.332	3.939
32	0.375	0.191	0.188	0.196	0.366	0.138	0.395	2.057
64	0.447	0.101	0.097	0.105	0.459	0.070	0.422	1.587

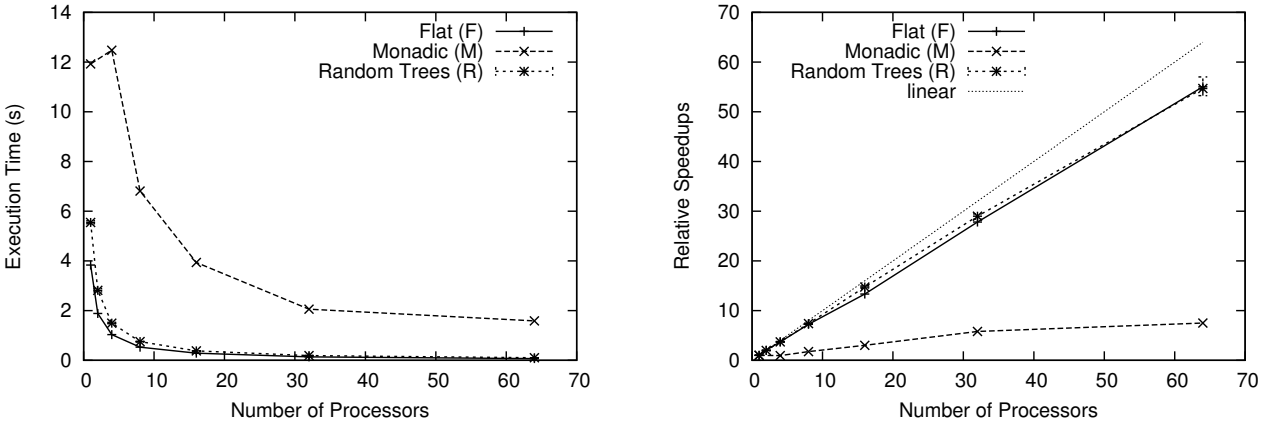


FIG. 5.1. *Plots of Table 5.1 by total execution time (left) and by speedups of computation time (right)*

repeated the same experiments five times, and their average times of the initial data distribution and of the computation itself ("dist." and "comp.", respectively) are summarized in the following tables (measurement in second). Randomly shaped trees (R) has various shapes and their computation time can naturally vary. The minimum and maximum are also put in Tables ("min." and "max.", respectively).

Table 5.1 shows the results of the first experiments of querying operations. As their plots in the left of Fig. 5.1 indicates, our algorithm exhibited good scalability. Results of (M) fell behind the other two. We can point out three reasons. The first reason is failure to use caches effectively. Routine 1 in (1) which uses arrays in a stack-like manner can enjoy caching effects when the corresponding open and close elements are located closely. The serialized representation of monadic trees is a sequence of open elements and a sequence of close elements afterwards. This hampers locality, and we can observe the penalties of high cache misses under a single processor $\mathrm{P} = 1$ where no parallelization is taken place. The second reason is communication and computation cost in Steps (2-3) and (2-4). We cannot apply any computation with monadic trees at (1), and this step has to leave as_i, bs_i and cs_j intact with their length $2\mathrm{N}/\mathrm{P}$ ($0 \leq i < \mathrm{P}/2$, $\mathrm{P}/2 \leq j < \mathrm{P}$). The third reason is communication anomalies when larger data are passed at Step (2-3). We will analyze this using the next experiments.

The results of the second experiments using our running example *maxPath* appear in Table 5.2. Since the required computations are quite cheap, improvements by parallelization are limited, and it does not pay off to perform parallelization for this data size. Instead, these experiments exhibit the cost of parallelization, especially that of data transactions. We make two notes on anomalies of communications. The first is that the variance of execution time becomes large as the number of processors increases. We observed that there were at large 10 msecs difference in execution time under $\mathrm{P} = 64$. The flat trees has least communication and computation cost, but there appears inversion phenomena that our algorithm ran faster over random trees than over a flat tree ($\mathrm{P} = 4, 8, 16$). The second is about congestion of network, which apparently happens toward the

TABLE 5.2
Execution times of the second experiments (maxPath, time in second)

	Random Trees (R)				Flat (F)		Monadic (M)	
p	dist.	comp.	min.	max.	dist.	comp.	dist.	comp.
1	0.088	0.055	0.054	0.058	0.086	0.044	0.086	0.086
2	0.207	0.028	0.027	0.032	0.226	0.022	0.210	0.170
4	0.252	0.016	0.014	0.019	0.248	0.018	0.250	0.086
8	0.309	0.013	0.007	0.016	0.306	0.016	0.301	0.049
16	0.342	0.010	0.005	0.012	0.355	0.011	0.333	0.031
32	0.375	0.013	0.004	0.028	0.397	0.011	0.366	0.022
64	0.450	0.019	0.007	0.033	0.456	0.007	0.490	0.139

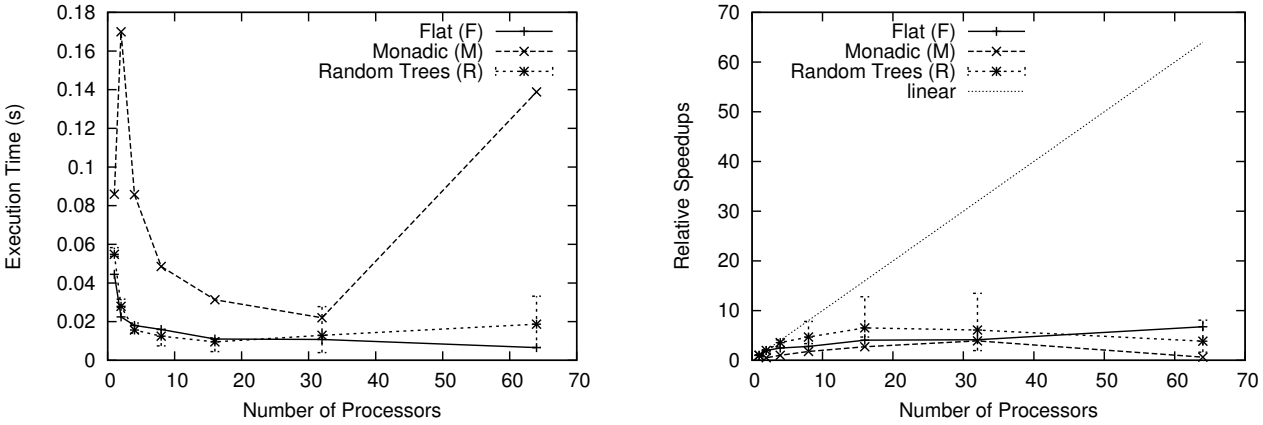


FIG. 5.2. *Plots of Table 5.2 by total execution time (left) and by speedups of computation time (right)*

monadic tree under $P = 64$. Monadic trees require Step (2-3) to transmit data of large size, but the amount of whole transmitted data through network does not change as $\sum_i |cs_i| = 2N/P + O(P)$ for any $P \geq 2$. Our algorithm with all-to-all transactions has to have vulnerability to the configuration and status of networks.

As a final note, it is natural that no difference existed in terms of costs of initial data distribution regardless of the shape of the trees.

6. Related Work. This section compares our proposed framework with related work. It mainly spans parallel tree contractions and list ranking algorithms, flattening transformation, parentheses matching, list homomorphism, and MapReduce-based implementation.

6.1. Parallel tree contractions. The parallel tree contractions algorithm, first proposed by Miller and Reif [34], are very important parallel algorithms for trees. Many researchers have devoted themselves to developing efficient implementations on various parallel models [18, 14, 1, 3, 31, 32, 16, 2]. Among researches based on shared memory environments, Gibbons and Rytter developed an optimal algorithm on CREW PRAM [18]; Abrahamson et al. developed an optimal and efficient algorithm in $O(N/P + \log P)$ on EREW PRAM [1].

Recently parallel tree contractions as well as list ranking, which serves the basis of parallel tree contractions, have been analyzed under the assumption of distributed memory environments. Mayr and Werchner showed $O(\lceil N/P \rceil \log P)$ implementations on hypercubes or related hypercubic networks [31, 32]. Dehne et al. solved list ranking and tree contractions on CGM/BSP using $O(N/P \log P)$ parallel time [16]. Sibeyn proposed a list ranking algorithm which aimed at reduction of communication costs [40].

Our refined algorithm runs in $O(N/P + P)$ and is much improved result which comes close to the algorithms under PRAM model. The advantage of our algorithm is not limited to the theoretical aspect. As our experiments demonstrated, the data representation we employed suits current computer architectures which extensively relies on caching mechanism for fast execution. Linked structures, namely dynamic data structures, involve pointers

and their data fragments can scatter over the memory heap. Flexible though they are in terms of structure manipulation like insertion and deletion, scattered data can easily lack locality.

The second advantage is the cost and concerns of data distribution. In case we have sophisticated distribution approaches of tree structures we can employ the EREW-PRAM parallel tree contraction algorithm. One such technique is based on m-bridges [38]. This technique translates a binary tree into another binary tree in each of whose nodes locates a subtrees of the original tree partitioned into almost the same size. Our group has another implementation for parallel tree computation based on this approach [41, 29, 28, 26]. The drawback of the approach using m-bridge is its cost. When trees are kept distributively among processors, the algorithm for realizing m-bridges requires Euler tour and list ranking, which as a result spoils theoretical complexity and running time in total. It should be noted that the parallelized algorithm presented in this paper runs in $O(N/P + P)$. This cost is theoretically comparable to the EREW-PRAM parallel tree contractions algorithm, using ignorable cost and troubles of initial data distribution.

6.2. Nested parallelism and flattening transformation. Blelloch's *nested parallelism* and the language NESL addresses the importance of data-parallel computation toward nested structures (often in the form of nested lists), where the length of each list can differ [6]. The idea proposed is *flattening* of the structures [7]. The importance of this problem domain ignited a lot of researches afterwards, theoretically and in real compilers [8, 39, 10].

Our idea presented in this paper is one instance of flattening transformation where segment descriptors are diffused into the flattened data. To cope with irregularity of tree structures we format trees into its serialized representation with an additional tags indicating the depth. The flattening of tree structures has already been researched. Prins and Palmer developed data-parallel language Proteus, and nesting trees were treated [37, 35]. Chakravarty and Keller extended the structure to involve trees and recursive data structures in general [24, 9]. Their computation framework, however, stayed to treat horizontal computation in parallel. As far as we are aware, this paper is the first to relate flattening transformations with the parallel tree contractions to derive parallelism in the vertical direction.

6.3. Parentheses matching. The process of our algorithm development much resembles parentheses matching algorithms, or the All Nearest Smaller Values Problems (often called ANSV for short). Their algorithms have been analyzed under CRCW PRAM [4], EREW PRAM [36], hypercube [25] and BSP [21]. The resemblance naturally comes from how tree structures are translated in sequence; the generation of *group*s in Routine 2 follows the process to find matching of ANSV developed in [4]. We have made a step forward to apply computation over the data structures. The implementation under BSP has complexity of $O(N/P + P)$. Our algorithm naturally has complexity comparable to it.

He and Huang analyzed that the cost of data transaction becomes constant toward sequences of random values [21]. Unfortunately this observation does not hold when we are to compute tree reductions. The first reason is that we have to transmit not only the information of shallowest depth d_i, but the computed results cs_i to realize reduction computations. Secondly, their style of analysis based on the assumption of random input does not apply since the serialized representation of trees has certain properties. For example, given a sequence $A = a_0, a_1, \ldots, a_{2n-1}$ which consists of open and close elements. In order to be a well-formed serialized representation, A has exactly n open and n close elements, and the number of open elements in any subsequence of A, namely $a_0, \ldots a_i$ $(i < 2n)$, has to be no less than that of close elements. While we haven't developed qualitative analyses so far, the experiments using randomly generated trees in this paper indicates that the communication cost stays in a reasonable amount.

6.4. List homomorphism. List homomorphism is a model that plays an important role for developing efficient parallel programs [13, 19, 20, 22]. A function h^{L} is a homomorphism if there exist an associative operator $\oplus$ and a unary function g such that

$$\begin{aligned} h^{\mathrm{L}}(x +\!\!+ y) &= h^{\mathrm{L}}\ x \odot h^{\mathrm{L}}\ y\ , \\ h^{\mathrm{L}}\ [a] &= g\ a\ . \end{aligned}$$

This function can be efficiently implemented in parallel since it ideally suits for divide-and-conquer, bottom-up computation: a list is divided into two fragments x and y recursively, and the computations of $h^{\mathrm{L}}\ x$ and

$h^{\mathrm{L}}\ y$ can be carried out in parallel. For instance, the function *sum*, which computes the sum of all the elements in a list, is a homomorphism because the equations $sum\ (x +\!\!+ y) = sum\ x + sum\ y$ and $sum\ [a] = a$ hold. This indicates that we can reduce parallel programming into construction of list homomorphism.

In further detail, parallel environments for distributed lists evaluate list homomorphism in two phases after distribution of data: (a) the local computation at each processor (using g and $\odot$), and (b) the global computation among processors (using $\odot$). Assume the computation of g and $\odot$ requires constant time. By evenly distributing consecutive elements to processors, the phase (a) takes $\mathrm{O}(\mathrm{N}/\mathrm{P})$ cost. The phase (b) reduces these P values into a single value tree-recursively in $\log \mathrm{P}$ iterations. The total cost is therefore estimated as $\mathrm{O}(\mathrm{N}/\mathrm{P} + \log \mathrm{P})$.

Tree-recursive data communication, which is the basis of efficient execution for list homomorphism, however, seems restrictive for this problem. Its resulting complexity is, even with nested use of homomorphism, $\mathrm{O}(\log^2 \mathrm{N})$ for abundance of processors. We made a test implementation for this approach, and we observed the scalability was tamed much earlier: the speedup ratios from $\mathrm{P} = 16$ to 32, and from 32 to 64 using the first experiments of querying were, respectively, 1.81 and 1.26, and they did not catch up the respective ratios 1.97 and 1.89 obtained by the approach in this paper.

6.5. MapReduce-based Implementation. MapReduce is a framework for large-scale data processing proposed by Google [15], and its open-source implementation in Hadoop [44] is now widely used. Though the conventional programming model of MapReduce is for unordered data (sets), with some extension in Hadoop MapReduce we can deal with ordered data including serialized representation of trees. Recently, dealing XML documents on MapReduce has been studied actively. For example, Choi at al. developed an XML querying system on Hadoop MapReduce [11]. They also proposed algorithms for labeling XML trees with MapReduce [12]. It is worth noting that the idea similar to parentheses matching was used in the latter to process unmatching tags in XML fragments.

The serialized representation of trees and flexibility of partitioning in our approach is also suitable for the implementation of tree manipulations on MapReduce. Based on the earlier version of this work, Emoto and Imachi developed a MapReduce algorithm for tree reductions [17]. Furthermore, the algorithm was extended to tree accumulations by Matsuzaki and Miyazaki [30], which can be implemented with two-round MapReduce computation.

7. Concluding Remarks. In this paper we have developed a new approach for parallel tree reductions. The essence of our approach is to make good use of the serialized representation of trees in terms of initial data distribution and computation using high memory locality. The results by the local computation forms a binary tree of size less than twice of the number of processors and each node has size $\mathrm{O}(\mathrm{N}/\mathrm{P})$. Our algorithm has a BSP implementation with three supersteps. Our test implementation showed good scalability in general, but we also observed network fluctuation under PC clusters.

This research will be improved in the following aspects. Our experiments exhibited good performance toward random inputs. It is an interesting mathematical question how much amount of transactions and computation we have to execute in Steps (2-3) and (2-4) in average. The flexibility in partitioning our representations will be beneficial under heterogeneous environments with different processor ability. After the process of *group* generation, data can be reallocated among processors, regardless of their computational power. Theoretical support for these issues are needed.

Acknowledgment. This research was partially supported by the Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Young Scientists (B), 17700026, 2005–2007.

REFERENCES

[1] K. R. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. M. Przytycka, *A simple parallel tree contraction algorithm*, J. Algorithms, 10 (1989), pp. 287–302.

[2] D. A. Bader, S. Sreshta, and N. R. Weisse-Bernstein, *Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs) (extended abstract)*, in High Performance Computing - HiPC 2002, 9th International Conference, Bangalore, India, December 18-21, 2002, Proceedings, S. Sahni, V. K. Prasanna, and U. Shukla, eds., vol. 2552 of Lecture Notes in Computer Science, Springer, 2002, pp. 63–78.

[3] R. P. K. Banerjee, V. Goel, and A. Mukherjee, *Efficient parallel evaluation of CSG tree using fixed number of processors*, in Solid Modeling and Applications, 1993, pp. 137–146.
[4] O. Berkman, B. Schieber, and U. Vishkin, *Optimal doubly logarithmic parallel algorithms based on finding all nearest smaller values*, J. Algorithms, 14 (1993), pp. 344–370.
[5] G. E. Blelloch, *Scans as primitive parallel operations*, IEEE Trans. Computers, 38 (1989), pp. 1526–1538.
[6] ———, *Programming parallel algorithms*, Commun. ACM, 39 (1996), pp. 85–97.
[7] G. E. Blelloch and G. Sabot, *Compiling collection-oriented languages onto massively parallel computers*, J. Parallel Distrib. Comput., 8 (1990), pp. 119–134.
[8] D. C. Cann, *Retire fortran? A debate rekindled*, Commun. ACM, 35 (1992), pp. 81–89.
[9] M. M. T. Chakravarty and G. Keller, *More types for nested data parallel programming*, in Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000., M. Odersky and P. Wadler, eds., ACM, 2000, pp. 94–105.
[10] M. M. T. Chakravarty, G. Keller, R. Lechtchinsky, and W. Pfannenstiel, *Nepal — nested data parallelism in Haskell*, in Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference Manchester, UK August 28-31, 2001, Proceedings, R. Sakellariou, J. A. Keane, J. R. Gurd, and L. Freeman, eds., vol. 2150 of Lecture Notes in Computer Science, Springer, 2001, pp. 524–534.
[11] H. Choi, K.-H. Lee, S.-H. Kim, Y.-J. Lee, and B. Moon, *HadoopXML: A suite for parallel processing of massive XML data with multiple twig pattern queries*, in Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM'12), ACM, 2012, pp. 2737–2739.
[12] H. Choi, K.-H. Lee, and Y.-J. Lee, *Parallel labeling of massive XML data with MapReduce*, Journal of Supercomputing, 67 (2014), pp. 408–437.
[13] M. Cole, *Parallel programming with list homomorphisms*, Parallel Processing Letters, 5 (1995), pp. 191–203.
[14] R. Cole and U. Vishkin, *Optimal parallel algorithms for expression tree evaluation and list ranking*, in VLSI Algorithms and Architectures, 3rd Aegean Workshop on Computing, AWOC 88, Corfu, Greece, June 28 - July 1, 1988, Proceedings, J. H. Reif, ed., vol. 319 of Lecture Notes in Computer Science, Springer, 1988, pp. 91–100.
[15] J. Dean and S. Ghemawat, *MapReduce: Simplified data processing on large clusters*, in 6th Symposium on Operating System Design and Implementation (OSDI2004), December 6–8, 2004, San Francisco, California, USA, 2004, pp. 137–150.
[16] F. K. H. A. Dehne, A. Ferreira, E. Cáceres, S. W. Song, and A. Roncato, *Efficient parallel graph algorithms for coarse-grained multicomputers and BSP*, Algorithmica, 33 (2002), pp. 183–200.
[17] K. Emoto and H. Imachi, *Parallel tree reduction on MapReduce*, in Proceedings of the International Conference on Computational Science (ICCS 2012), vol. 9 of Procedia Computer Science, Elsevier, 2012, pp. 1827–1836.
[18] A. Gibbons and W. Rytter, *An optimal parallel algorithm for dynamic expression evaluation and its applications*, in Foundations of Software Technology and Theoretical Computer Science, Sixth Conference, New Delhi, India, December 18-20, 1986, Proceedings, K. V. Nori, ed., vol. 241 of Lecture Notes in Computer Science, Springer, 1986, pp. 453–469.
[19] S. Gorlatch, *Constructing list homomorphisms*, Tech. Report MIP-9512, Fakultät für Mathematik und Informatik, Universität Passau, August 1995.
[20] Z. N. Grant-Duff and P. G. Harrison, *Parallelism via homomorphisms*, Parallel Processing Letters, 6 (1996), pp. 279–295.
[21] X. He and C. Huang, *Communication efficient BSP algorithm for all nearest smaller values problem*, J. Parallel Distrib. Comput., 61 (2001), pp. 1425–1438.
[22] Z. Hu, H. Iwasaki, and M. Takeichi, *Formal derivation of efficient parallel programs by construction of list homomorphisms*, ACM Trans. Program. Lang. Syst., 19 (1997), pp. 444–461.
[23] K. Kakehi, K. Matsuzaki, and K. Emoto, *Efficient parallel tree reductions on distributed memory environments*, in Computational Science - ICCS 2007, 7th International Conference, Beijing, China, May 27 - 30, 2007, Proceedings, Part II, Y. Shi, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, eds., vol. 4488 of Lecture Notes in Computer Science, Springer, 2007, pp. 601–608.
[24] G. Keller and M. M. T. Chakravarty, *Flattening trees*, in Euro-Par '98 Parallel Processing, 4th International Euro-Par Conference, Southampton, UK, September 1-4, 1998, Proceedings, D. J. Pritchard and J. Reeve, eds., vol. 1470 of Lecture Notes in Computer Science, Springer, 1998, pp. 709–719.
[25] D. Kravets and C. G. Plaxton, *All nearest smaller values on the hypercube*, IEEE Trans. Parallel Distrib. Syst., 7 (1996), pp. 456–462.
[26] K. Matsuzaki, *Efficient implementation of tree accumulations on distributed-memory parallel computers*, in Computational Science - ICCS 2007, 7th International Conference, Beijing, China, May 27 - 30, 2007, Proceedings, Part II, Y. Shi, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, eds., vol. 4488 of Lecture Notes in Computer Science, Springer, 2007, pp. 609–616.
[27] K. Matsuzaki, Z. Hu, K. Kakehi, and M. Takeichi, *Systematic derivation of tree contraction algorithms*, Parallel Processing Letters, 15 (2005), pp. 321–336.
[28] K. Matsuzaki, Z. Hu, and M. Takeichi, *Parallel skeletons for manipulating general trees*, Parallel Computing, 32 (2006), pp. 590–603.
[29] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu, *A library of constructive skeletons for sequential style of parallel programming*, in Proceedings of the 1st International Conference on Scalable Information Systems, Infoscale 2006, Hong Kong, May 30-June 1, 2006, X. Jia, ed., vol. 152 of ACM International Conference Proceeding Series, ACM, 2006, p. 13.
[30] K. Matsuzaki and R. Miyazaki, *Parallel tree accumulations on MapReduce*, International Journal of Parallel Programming, 44 (2016), pp. 466–485.
[31] E. W. Mayr and R. Werchner, *Optimal routing of parentheses on the hypercube*, J. Parallel Distrib. Comput., 26 (1995), pp. 181–192.

[32] ———, *Optimal tree contraction and term matching on the hypercube and related networks*, Algorithmica, 18 (1997), pp. 445–460.
[33] W. F. McColl, *Scalable computing*, in Computer Science Today: Recent Trends and Developments, J. van Leeuwen, ed., vol. 1000 of Lecture Notes in Computer Science, Springer, 1995, pp. 46–61.
[34] G. L. Miller and J. H. Reif, *Parallel tree contraction and its application*, in 26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985, IEEE Computer Society, 1985, pp. 478–489.
[35] D. W. Palmer, J. F. Prins, and S. Westfold, *Work-efficient nested data-parallelism*, in Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation (Frontiers'95), FRONTIERS '95, Washington, DC, USA, 1995, IEEE Computer Society, pp. 186–.
[36] S. K. Prasad, S. K. Das, and C. C. Chen, *Efficient EREW PRAM algorithms for parentheses-matching*, IEEE Trans. Parallel Distrib. Syst., 5 (1994), pp. 995–1008.
[37] J. Prins and D. W. Palmer, *Transforming high-level data-parallel programs into vector operations*, in Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), San Diego, California, USA, May 19-22, 1993, M. C. Chen and R. Halstead, eds., ACM, 1993, pp. 119–128.
[38] M. Reid-Miller, G. L. Miller, and F. Modugno, *List ranking and parallel tree contraction*, in Synthesis of Parallel Algorithms, J. Reif, ed., Morgan Kaufmann, 1993, ch. 3, pp. 115–194.
[39] J. Riely and J. Prins, *Flattening is an improvement*, in Static Analysis, 7th International Symposium, SAS 2000, Santa Barbara, CA, USA, June 29 - July 1, 2000, Proceedings, J. Palsberg, ed., vol. 1824 of Lecture Notes in Computer Science, Springer, 2000, pp. 360–376.
[40] J. F. Sibeyn, *One-by-one cleaning for practical parallel list ranking*, Algorithmica, 32 (2002), pp. 345–363.
[41] *Sketo project home page*. `http://sketo.ipl-lab.org/`.
[42] D. B. Skillicorn, *Parallel implementation of tree skeletons*, J. Parallel Distrib. Comput., 39 (1996), pp. 115–125.
[43] L. G. Valiant, *A bridging model for parallel computation*, Commun. ACM, 33 (1990), pp. 103–111.
[44] T. White, *Hadoop: The Definitive Guide*, O'Reilly Media / Yahoo Press, 2012.

Edited by: Frédéric Loulergue
Received: September 16, 2016
Accepted: January 17, 2017

DOI 10.12694/scpe.v18i1.1231
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 18, Number 1, pp. 17–34. http://www.scpe.org

EFFICIENT IMPLEMENTATION OF TREE SKELETONS ON DISTRIBUTED-MEMORY PARALLEL COMPUTERS*

KIMINORI MATSUZAKI†

Abstract. Parallel tree skeletons are basic computational patterns that can be used to develop parallel programs for manipulating trees. In this paper, we propose an efficient implementation of parallel tree skeletons on distributed-memory parallel computers. In our implementation, we divide a binary tree to segments based on the idea of m-bridges with high locality, and represent local segments as serialized arrays for high sequential performance. We furthermore develop a cost model for our implementation of parallel tree skeletons. We confirm the efficacy of our implementation with several experiments.

Key words: Parallel skeletons, Tree, Cost model, Parallel programming

AMS subject classifications. 68N19, 68W10

1. Introduction. *Parallel tree skeletons*, first formalized by Skillicorn [33, 34], are basic computational patterns of parallel programs manipulating trees. By using parallel tree skeletons, we can develop parallel programs without considering low-level parallel implementation and details of parallel computers. There have been several studies on the systematic methods of developing parallel programs by means of parallel tree skeletons [6, 19, 21, 35, 36].

For efficient parallel tree manipulations, tree contraction algorithms have been intensively studied [1, 5, 24, 25, 38]. Many tree contraction algorithms were given on various parallel computational models, for instance, EREW PRAM [1], Hypercubes [24], and BSP/CGM [5]. Several parallel tree manipulations were developed based on the tree contraction algorithms [1, 7]. For tree skeletons, Gibbons et al. [11] developed an implementation algorithm based on tree contraction algorithms.

In this paper, we propose an efficient implementation of parallel tree skeletons for binary trees on distributed-memory parallel computers. Compared with the implementations so far, our implementation has three new features.

First, it has less overheads of parallelism. Locality is one of the most important properties in developing efficient parallel programs especially for distributed-memory computers. We adopt the technique of m-bridges [8, 30] in the basic graph theory to divide binary trees into segments with high locality.

Second, it has high sequential performance. The performance of sequential computation parts is as important as that of the communication parts. We represent a local segment as a serialized array and implement local computation in tree skeletons with loops rather than recursive functions.

Third, it has a cost model. We formalize a cost model of our parallel implementation. The cost model helps us to divide binary trees with good load balance.

We have implemented tree skeletons in C++ and MPI, and they are available as part of the skeleton library SkeTo [22]. We confirm the efficacy of our implementation of tree skeletons with several experiments.

This paper is organized as follows. In the following Sect. 2, we introduce parallel tree skeletons with two examples. In Sect. 3, we discuss the division of binary trees and representation of divided trees. In Sect. 4, we develop an efficient implementation and a cost model of tree skeletons on distributed-memory parallel computers. Based on this cost model, we discuss the optimal division of binary trees in Sect. 5. We then show several experiment results in Sect. 6. We review related work in Sect. 7, and make concluding remarks in Sect. 8.

2. Parallel Tree Skeletons.

*An earlier version of this paper appeared in PAPP2007, part of ICCS2007[16]. This work was partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (B) No. 17300005, and the Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Young Scientists (B) No. 18700021.

†School of Information, Kochi University of Techonology, 185 Tosayamadacho-Miyanokuchi, Kami, Kochi 782–8502 Japan. (matsuzaki.kiminori@kochi-tech.ac.jp)

$$
\begin{aligned}
&\mathsf{map} :: (\alpha \to \gamma, \beta \to \delta, \mathit{BTree}\langle\alpha,\beta\rangle) \to \mathit{BTree}\langle\gamma,\delta\rangle\\
&\mathsf{map}(k_l, k_n, \mathit{BLeaf}(a)) \quad\ = \mathit{BLeaf}(k_l(a))\\
&\mathsf{map}(k_l, k_n, \mathit{BNode}(l,b,r)) = \mathit{BNode}(\mathsf{map}(k_l,k_n,l), k_n(b), \mathsf{map}(k_l,k_n,r))\\
&\\
&\mathsf{reduce} :: ((\alpha,\beta,\alpha)\to\alpha, \mathit{BTree}\langle\alpha,\beta\rangle)\to\alpha\\
&\mathsf{reduce}(k, \mathit{BLeaf}(a)) \quad\ = a\\
&\mathsf{reduce}(k, \mathit{BNode}(l,b,r)) = k(\mathsf{reduce}(k,l), b, \mathsf{reduce}(k,r))\\
&\\
&\mathsf{uAcc} :: ((\alpha,\beta,\alpha)\to\alpha, \mathit{BTree}\langle\alpha,\beta\rangle)\to\mathit{BTree}\langle\alpha,\alpha\rangle\\
&\mathsf{uAcc}(k, \mathit{BLeaf}(a)) \quad\ = \mathit{BLeaf}(a)\\
&\mathsf{uAcc}(k, \mathit{BNode}(l,b,r)) = \mathbf{let}\ b' = \mathsf{reduce}(k, \mathit{BNode}(l,b,r))\\
&\qquad\qquad\qquad\qquad\ \mathbf{in}\ \mathit{BNode}(\mathsf{uAcc}(k,l), b', \mathsf{uAcc}(k,r))\\
&\\
&\mathsf{dAcc} :: ((\gamma,\beta)\to\gamma, (\gamma,\beta)\to\gamma, \gamma, \mathit{BTree}\langle\alpha,\beta\rangle)\to\mathit{BTree}\langle\gamma,\gamma\rangle\\
&\mathsf{dAcc}(g_l, g_r, c, \mathit{BLeaf}(a)) \quad\ = \mathit{BLeaf}(c)\\
&\mathsf{dAcc}(g_l, g_r, c, \mathit{BNode}(l,b,r)) = \mathbf{let}\ l' = \mathsf{dAcc}(g_l, g_r, g_l(c,b), l)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \ r' = \mathsf{dAcc}(g_l, g_r, g_r(c,b), r)\\
&\qquad\qquad\qquad\qquad\qquad\ \mathbf{in}\ \mathit{BNode}(l', c, r')
\end{aligned}
$$

FIG. 2.1. *Definition of parallel tree skeletons.*

2.1. Binary Trees. Binary trees are trees whose internal nodes have exactly two children. In this paper, leaves and internal nodes of a binary tree may have different types. The datatype of binary trees whose leaves have values of type α and internal nodes have values of type β is defined as follows.

$$
\begin{aligned}
\text{data } \mathit{BTree}\langle\alpha,\beta\rangle = {} & \mathit{BLeaf}(\alpha)\\
\mid {} & \mathit{BNode}(\mathit{BTree}\langle\alpha,\beta\rangle, \beta, \mathit{BTree}\langle\alpha,\beta\rangle)
\end{aligned}
$$

Functions that manipulate binary trees can be defined by pattern matching. For example, function *root* that returns the value of the root node is as follows.

$$
\begin{aligned}
&\mathit{root} :: \mathit{BTree}\langle\alpha,\alpha\rangle \to \alpha\\
&\mathit{root}(\mathit{BLeaf}(a)) \quad = a\\
&\mathit{root}(\mathit{BNode}(l,b,r)) = b
\end{aligned}
$$

2.2. Parallel Tree Skeletons. Parallel (binary-)tree skeletons are basic computational patterns manipulating binary trees in parallel. In this section, we introduce a set of basic tree skeletons proposed by Skillicorn [33, 34] with minor modifications for later discussion of their implementation (Fig. 2.1).

The tree skeleton map takes two functions k_l and k_n and a binary tree, and applies k_l to each leaf and k_n to each internal node. Though there are tree skeletons called zip or zipwith that take multiple trees of the same shape, we omit discussing them since computation of them is almost the same as that of the map skeleton.

The parallel skeleton reduce takes a function k and a binary tree, and collapses the tree into a value by applying the function k in a bottom-up manner. The parallel skeleton uAcc (upwards accumulate) is a shape-preserving manipulation, which also takes a function k and a binary tree and computes (reduce k) for each subtree.

The parallel skeleton dAcc (downwards accumulate) is another shape-preserving manipulation. This skeleton takes two functions g_l and g_r, an accumulative parameter c and a binary tree, and computes a value for each node by updating the accumulative parameter c in a top-down manner. The update is done by function g_l for the left child, and by function g_r for the right child.

Since a straightforward divide-and-conquer computation according to the definition in Fig. 2.1 has computational cost linear to the height of the tree, it may be inefficient if the input tree is ill-balanced. To guarantee existence of efficient parallel implementations, we impose some conditions on the parameter functions of tree skeletons. The map skeleton requires no condition. For the reduce, uAcc and dAcc skeletons, we formalize the conditions for parallel implementations as existence of auxiliary functions satisfying a certain closure property.

The reduce and uAcc skeletons called with parameter function k require existence of four auxiliary functions ϕ, ψ_n, ψ_l, and ψ_r satisfying the following equations.

$$\begin{aligned} &\text{a: } k(l,b,r) = \psi_n(l,\phi(b),r) \\ &\text{b: } \psi_n(\psi_n(x,l,y),b,r) = \psi_n(x,\psi_l(l,b,r),y) \\ &\text{c: } \psi_n(l,b,\psi_n(x,r,y)) = \psi_n(x,\psi_r(l,b,r),y) \end{aligned} \tag{2.1}$$

Equations (2.1.b) and (2.1.c) represent the closure property that functions ψ_n, ψ_l, and ψ_r should satisfy. Function ϕ lifts the computation of function k to the domain with closure property as in Equation (2.1.a). We denote the function k satisfying the condition as $k = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_u$.

The dAcc skeleton called with parameter functions g_l and g_r requires existence of auxiliary functions ϕ_l, ϕ_r, ψ_u, and ψ_d satisfying the following equations.

$$\begin{aligned} &\text{a: } g_l(c,b) = \psi_d(c,\phi_l(b)) \\ &\text{b: } g_r(c,b) = \psi_d\ c,\phi_r(b)) \\ &\text{c: } \psi_d(\psi_d(c,b),b') = \psi_d(c,\psi_u(b,b')) \end{aligned} \tag{2.2}$$

Equation (2.2.c) shows the closure property that functions ψ_d and ψ_u should satisfy. As shown in Equations (2.2.a) and (2.2.b), computations of g_l and g_r are lifted up to the domain with closure property by functions ϕ_l and ϕ_r, respectively. We denote the pair of functions (g_l, g_r) satisfying the condition as $(g_l, g_r) = \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_d$.

2.3. Examples. To see how we can develop parallel programs with these parallel tree skeletons, we consider the following two problems.

Sum of Values. Consider computing the sum of node values of a binary tree. We can define function *sum* for this problem simply by using the reduce skeleton.

$$\begin{aligned} sum\ t = \ &\mathsf{reduce}(add3, t) \\ &\textbf{where } add3(l,b,r) = l + b + r \end{aligned}$$

In this case, since the operator used is only the associative operator +, we have $add3 = \langle id, add3, add3, add3 \rangle_u$, where id is the identity function.

Prefix Numbering. Consider numbering the nodes of a binary tree in the prefix traversing order. We can define function *prefix* for this problem by using the tree skeletons map, uAcc, and dAcc as follows. Note that the result of the uAcc skeleton is a pair of number of nodes of left subtree and number of nodes for each node.

$$\begin{aligned} prefix\ t = \ &\textbf{let } t' = \mathsf{uAcc}(k, \mathsf{map}(f, id, t)) \\ &\textbf{in } \mathsf{dAcc}(g_l, g_r, 0, t') \\ &\textbf{where } f(a) = (0,1) \\ &\qquad k((l_l,l_s),b,(r_l,r_s)) = (l_s, l_s + 1 + r_s) \\ &\qquad g_l(c,(b_l,b_s)) = c + 1 \\ &\qquad g_r(c,(b_l,b_s)) = c + b_l + 1 \end{aligned}$$

Similar to the case of *sum*, auxiliary functions for the functions g_l and g_r are simply given as $(g_l, g_r) = \langle \lambda(b_l,b_s).1, \lambda(b_l,b_s).b_l + 1, +, + \rangle_d$. For function k, auxiliary functions are given as follows by applying the derivation technique discussed in [21].

$$\begin{aligned} &k = \langle \phi, \psi_n, \psi_l, \psi_r \rangle \\ &\textbf{where } \phi(b) = (1,0,0,1) \\ &\qquad \psi_n((l_l,l_s),(b_0,b_1,b_2,b_3),(r_l,r_s)) \\ &\qquad\qquad = (b_0 \times l_s + b_1 \times (l_s + 1 + r_s) + b_2,\ l_s + 1 + r_s + b_3) \\ &\qquad \psi_l((l_0,l_1,l_2,l_3),(b_0,b_1,b_2,b_3),(r_l,r_s)) \\ &\qquad\qquad = (0,\ b_0 + b_1,\ (b_0 + b_1) \times l_3 + b_1 \times (1 + r_s) + b_2,\ l_3 + 1 + r_s + b_3) \\ &\qquad \psi_r((l_l,l_s),(b_0,b_1,b_2,b_3,b),(r_0,r_1,r_2,r_3,r)) \\ &\qquad\qquad = (0,\ b_1,\ b_1 \times r_3 + b_0 \times l_s + b_1 \times (1 + l_s) + b_2,\ r_3 + 1 + l_s + b_3) \end{aligned}$$

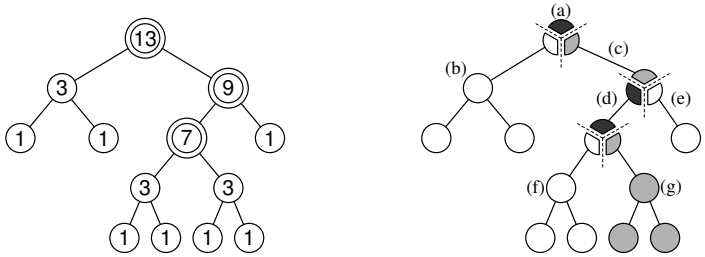


FIG. 3.1. *An example of m-critical nodes and m-bridges. Left: In this binary tree, there are three 4-critical nodes denoted by the doubly-lined circles. The number in each node denotes the number of nodes in the subtree. Right: For the same tree there are seven 4-bridges, (a)–(g), each of which is a set of connected nodes.*

It is worth noting that the set of auxiliary functions is not unique. For example, we can define another set of auxiliary functions in which an intermediate value has a flag and two values.

In general, auxiliary functions are more complicated than the original function as we have seen in this example. This complexity of auxiliary functions would introduce overheads in the derived parallel algorithms.

3. Division of Binary Trees with High Locality. To develop efficient parallel programs on distributed-memory parallel computers, we need to divide data into smaller parts and distribute them to the processors. Here, the division of data should have the following two properties for efficiency of parallel programs. The first property is *locality*. The data distributed to each processor should be adjacent. If two elements adjacent in the original data are distributed to different processors, then we often need communications between the processors. The second property is *load balance*. The number of elements distributed to each processor should be nearly equal since the cost of local computation is often proportional to the number of elements.

It is easy to divide a list with these two properties, that is, for a given list of N elements we simply divide the list into P sublists each having N/P elements. It is, however, difficult to divide a tree satisfying the two properties due to the nonlinear and irregular structure of binary trees.

In this section, we introduce a division of binary trees based on the basic graph theory [8, 30], and show how to represent the distributed tree structures for efficient implementation of tree skeletons.

3.1. Graph-Theoretic Results for Division of Binary Trees. We start by introducing some graph-theoretic results [8, 30]. Let $size(v)$ denote the number of nodes in the subtree rooted at node v.

DEFINITION 3.1 (m-Critical Node). *Let m be an integer. A node v is called an* m-critical node, *if*

- *v is an internal node, and*
- *for each child v' of v inequality $\lceil size(v)/m \rceil > \lceil size(v')/m \rceil$ holds.*

The m-critical nodes divide a tree into sets of adjacent nodes (m-bridges) as shown in Fig. 3.1.

DEFINITION 3.2 (m-Bridge). *Let m be an integer. An* m-bridge *is a set of adjacent nodes divided by m-critical nodes, that is, a largest set of adjacent nodes in which m-critical nodes are only at the root or bottom.*

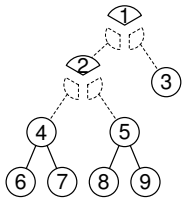
In the following of this paper, we assume that each local segment given by dividing a tree is an m-bridge. The global structure of m-bridges also forms a binary tree.

The m-critical nodes and the m-bridges have several important properties. The following two lemmas show properties of the m-critical nodes and the m-bridges in terms of the global shape of them.

LEMMA 3.3. *If v_1 and v_2 are m-critical nodes then their least common ancestor is also an m-critical node.*

LEMMA 3.4. *If B is an m-bridge of a tree then B has at most one m-critical node among the leaves of it.*

The root node in each m-bridge, except the m-bridge that includes the global root node, is an m-critical node. If we remove the root m-critical node if it exists, it follows from Lemma 3.4 and Definition 3.2 that the m-bridge has at most one m-critical node at its bottom.



$gt = [\square^{\mathsf{N}}, \square^{\mathsf{N}}, \square^{\mathsf{L}}, \square^{\mathsf{L}}, \square^{\mathsf{L}}]$

$segs = [\,[\,1^{\mathsf{C}}\,],$
$\quad [\,2^{\mathsf{C}}\,],$
$\quad [\,4^{\mathsf{N}}, 6^{\mathsf{L}}, 7^{\mathsf{L}}\,],$
$\quad [\,5^{\mathsf{N}}, 8^{\mathsf{L}}, 9^{\mathsf{L}}\,],$
$\quad [\,3^{\mathsf{L}}\,]\,]$

FIG. 3.2. *Array representation of divided binary trees. Each local segment of segs is assigned to one of processors and is not shared. Labels* L, N *and* C *denote a leaf, a normal internal node, and an m-critical node, respectively. Each m-critical node is included in the parent segment.*

The following three lemmas are related to the number of nodes in an m-bridge and the number of m-bridges in a tree. Note that the first two lemmas hold on general trees while the last lemma only holds on binary trees.

LEMMA 3.5. *The number of nodes in an m-bridge is at most* $m + 1$.

LEMMA 3.6. *Let* N *be the number of nodes in a tree then the number of m-critical nodes in the tree is at most* $2N/m - 1$.

LEMMA 3.7. *Let* N *be the number of nodes in a binary tree then the number of m-critical nodes in the binary tree is at least* $(N/m - 1)/2$.

Let N be the number of nodes and P be the number of processors. In the previous studies [8, 18, 30], we divided a tree into m-bridges using the parameter m given by $m = 2N/P$. Under this division we obtain at most $(2P - 1)$ m-bridges and thus each processor handles at most two m-bridges. Of course this division enjoys high locality, but it has poor load balance since the maximum number of nodes passed to a processor may be $2N/P$, which is twice of that for the best lead-balancing case.

In Sect. 5, we will adjust the value m for better division of binary trees based on the cost model of tree skeletons. The idea is to divide a binary tree into more m-bridges using smaller m so that we obtain enough load balance while keeping the overheads caused by loss of locality small.

3.2. Data Structure for Distributed Segments. The performance of the sequential computation parts is as important as that of the communication parts.

Generally speaking, tree structures are often implemented using pointers or references. There are, however, two problems in this implementation for large-scale tree applications. First, much memory is required for pointers. Considering trees of integers or real numbers, for example, we can see that the pointers use as much memory as the values do. Furthermore, if we allocate nodes one by one, more memory is consumed to enable each of them to be deallocated. Second, locality is often lost. Recent computers have a cache hierarchy to bridge the gap between the CPU speed and the memory speed, and cache misses greatly decrease the performance especially in data-intensive applications. If we allocate nodes from here and there then the probability of cache misses increases.

To resolve these problems, we represent a binary tree with arrays. We represent a tree divided by the m-bridges using one array *gt* for the global structure and one array of arrays *segs* for the local segments, each of which is given by serializing the tree in the order of prefix traversal. Note that arrays in *segs* are distributed among processors, while each local segment exists in only one processor. Figure 3.2 illustrates the array representation of a distributed tree. Since adjoining elements are aligned one next to another in this representation, we can reduce cache misses.

We introduce some notations for the discussion of implementation algorithms in the next section. Some values may be attached to the global structure, and we write $gt[i]$ to access to the value attached to ith element of global structure. If ith segment in *segs* is distributed on pth processor, we denote $pr(i) = p$. For a given serialized array for a segment *seg*, we use $seg[i]$ to denote the ith value in the serialized array, and use isLeaf($seg[i]$), isNode($seg[i]$) and isCritical($seg[i]$) to check whether the ith node is a leaf, an internal node, and an m-critical node, respectively. Function isRoot(p) checks if the processor p is the root processor or not.

TABLE 4.1
Parameters of the cost model.

$t_p(f)$	computational time of function f using p processors
N	the number of nodes in the input tree
P	the number of processors
m	the parameter for m-critical nodes and m-bridges
M	the number of segments given by division of trees
L_i	the number of nodes in the ith segment
D_i	the depth of the m-critical node in the ith segment
c_α	the time needed for communicating one data of type α
$\|\alpha\|$	the size of a value of type α

4. Implementation and Cost Model of Tree Skeletons. In this section, we show an implementation and a cost model of the tree skeletons on distributed-memory parallel computers. We implement the local computation in tree skeletons using loops and stacks on the serialized arrays, which play an important role in reducing the cache misses and achieving high performance in the sequential computation parts.

We define several parameters for discussion of the cost model (Table 4.1). We assume a homogeneous distributed-memory environments as the computation environment. The computational time of function f executed with p processors is denoted by $t_p(f)$. In particular, $t_1(f)$ denotes the cost of sequential computation of f. Parameter N denotes the number of nodes, and P denotes the number of processors. Parameter m is used for m-critical nodes and m-bridges, and M denotes the number of segments after the division. For the ith segment, in addition to the parameter of the number of nodes L_i, we introduce parameter D_i indicating the depth of the critical node. Parameter c_α denotes the communication time for a value of type α. The size of a value of type α is denoted by $|\alpha|$.

The cost of tree skeletons can be uniformly given as the sum of the maximum local-computation cost and the global-computation cost as follows.

$$\max_p \sum_{pr(i)=p} (L_i \times t_l + D_i \times t_d) + M \times t_m$$

In the expression of the cost, $\sum_{pr(i)=p}$ denotes the summation of cost for local segments associated to processor p. The parameter t_l indicates the cost of computation that is applied to each node in a segment, the parameter t_d indicates the cost of computation that is applied to the nodes on the path from the root to the m-critical node, and the parameter t_m denotes the communication cost required for each segment.

In fact, we can extend the implementation of tree skeletons and the cost model to the bulk-synchronous parallel (BSP) model [37]. The cost of a BSP algorithm is given by the sum of costs of supersteps, which consists of local computation followed by communication and barrier synchronization. The cost of a superstep is given by an expression of the form $w + hg + l$ where w is the maximum cost of local computation, h is the size of messages, g is the cost of communicating a message of size one, and l is the cost of the barrier synchronization. Note that for the parameter g we have $c_\alpha = |\alpha|g$ for any type α.

4.1. Map. Since there is no dependency among nodes in the computation of the map skeleton, we can implement the map skeleton by applying function MAP_LOCAL to each local segment, where the MAP_LOCAL function applies function k_l to each leaf and function k_n to each internal node and the m-critical node in a local segment. The implementation of the map skeleton is given in Fig. 4.1.

In a local segment with L_i nodes, the number of leaves is at most $L_i/2+1$ and the number of internal nodes including the m-critical node is at most $L_i/2 + 1$. Ignoring small constants we can specify the computational cost of the MAP_LOCAL function as

$$t_1(\text{MAP\_LOCAL}) = \frac{L_i}{2} \times t_1(k_l) + \frac{L_i}{2} \times t_1(k_n) \ .$$

```
map(k_l, k_n, (gt, segs))
  for i ← 0 to gt.size − 1: begin
    if pr(i) == p then segs'[i] ← MAP_LOCAL(k_l, k_n, segs[i]); endif
  end
  return (gt, segs');

MAP_LOCAL(k_l, k_n, seg)
  for i ← 0 to seg.size − 1: begin
    if isLeaf(seg[i])     then seg'[i] ← k_l(seg[i]); endif
    if isNode(seg[i])     then seg'[i] ← k_n(seg[i]); endif
    if isCritical(seg[i]) then seg'[i] ← k_n(seg[i]); endif
  end
  return seg';
```

FIG. 4.1. *Implementation of* map *skeleton.*

The cost of the map skeleton is as follows.

$$t_P(\mathsf{map}(k_l, k_n, t)) = \max_p \sum_{pr(i)=p} L_i \times \frac{t_1(k_l) + t_1(k_n)}{2}$$

On the BSP model, we can implement the map skeleton with a single superstep without communication. Thus, the BSP cost is given as follows.

$$t_P^{\langle \mathrm{BSP} \rangle}(\mathsf{map}(k_l, k_n, t)) = \max_p \sum_{pr(i)=p} L_i \times \frac{t_1(k_l) + t_1(k_n)}{2} + l$$

4.2. Reduce. We then show an implementation and its cost of the reduce skeleton called with function k and auxiliary functions $k = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_u$. Let the input binary tree have type $BTree\langle \alpha, \beta \rangle$ and intermediate values for auxiliary functions have type γ. The implementation of the reduce skeleton is shown in Fig. 4.2.

Step 1. Local Reduction. The bottom-up computation of the reduce skeleton can be computed by a traversal on the array from right to left using a stack for the intermediate results. Firstly we apply REDUCE_LOCAL function to each local segment to reduce it to a value. In the REDUCE_LOCAL function,

- we apply functions ϕ and either ψ_l or ψ_r to the m-critical node and its ancestors, and
- we apply function k to the other internal nodes.

Here, applying the function k is cheaper than applying function ϕ and ψ_n, even though $k(l, n, r) = \psi_n(l, \phi(n), r)$ holds with respect to the results of functions. To specify where the m-critical node or its ancestor is in the stack, we use a variable d that indicates the position. Note that in the computation of the REDUCE_LOCAL function, at most one element in the stack has the value of the m-critical node or its ancestors.

In this step, functions ϕ and either ψ_l or ψ_r are applied to the m-critical node and its ancestors (D_i nodes) and function k is applied to the other internal nodes ($(M_i/2 - D_i)$ nodes). Thus, the cost of REDUCE_LOCAL is given as follows.

$$t_1(\mathrm{REDUCE\_LOCAL}) = D_i \times (t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r))) + \left(\frac{L_i}{2} - D_i\right) \times t_1(k)$$

Step 2. Gathering Local Results to Root Processor. In the second step, we gather the local results to the root processor. The communication cost is given by the number of leaf segments of type α and the number of internal segments of type γ.

$$t_P(\text{Step 2}) = \frac{M}{2} \times c_\alpha + \frac{M}{2} \times c_\gamma$$

Let the gathered values be put in array gt on the root processor after this step.

```
reduce(k, (gt, segs)) where k = ⟨φ, ψn, ψl, ψr⟩u
  for i ← 0 to gt.size − 1: begin
    if pr(i) == p then gt[i] ← REDUCE_LOCAL(k, φ, ψl, ψr, segs[i]); endif
  end
  gather_to_root(gt);
  if isRoot(p) then return REDUCE_GLOBAL(ψn, gt); endif

REDUCE_LOCAL(k, φ, ψl, ψr, seg)
  stack ← ∅; d ← −∞;
  for i ← seg.size − 1 to 0: begin
    if isLeaf(seg[i]) then stack ← seg[i]; d ← d + 1; endif
    if isNode(seg[i]) then
      lv ← stack; rv ← stack;
      if d == 0       then stack ← ψl(lv, φ(seg[i]), rv);
      else if d == 1 then stack ← ψr(lv, φ(seg[i]), rv); d ← 0;
      else                stack ← k(lv, seg[i], rv); d ← d − 1;
      endif
    if isCritical(seg[i]) then stack ← φ(seg[i]); d ← 0; endif
  end
  top ← stack; return top;

REDUCE_GLOBAL(ψn, gt)
  stack ← ∅;
  for i ← gt.size − 1 to 0: begin
    if isLeaf(gt[i]) then stack ← gt[i]; endif
    if isNode(gt[i]) then lv ← stack; rv ← stack; stack ← ψn(lv, gt[i], rv); endif
  end
  top ← stack; return top;
```

FIG. 4.2. *Implementation of* reduce *skeleton*

Step 3. Global Reduction on Root Processor. Finally, we compute the result of the reduce skeleton by applying the REDUCE_GLOBAL function to the array of local results. This computation is performed on the root processor. We compute the result by applying ψ_n for each internal node in a bottom-up manner, which is implemented by a traversal with a stack on the array of the global structure from right to left.

In this step the function ψ_n is applied to each internal segment and the cost of REDUCE_GLOBAL is

$$t_1(\text{REDUCE\_GLOBAL}) = \frac{M}{2} \times t_1(\psi_n) \ .$$

In summary, the cost of the reduce skeleton is given as follows.

$$\begin{aligned} & t_P(\mathsf{reduce}\ k) \\ & \quad = \max_p \sum_{pr(i)=p} t_1(\text{REDUCE\_LOCAL}) + t_P(\text{Step 2}) + t_1(\text{REDUCE\_GLOBAL}) \\ & \quad = \max_p \sum_{pr(i)=p} \left(L_i \times \frac{t_1(k)}{2} + D_i \times (-t_1(k) + t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r))) \right) \\ & \qquad + M \times \frac{c_\alpha + c_\gamma + t_1(\psi_n)}{2} \end{aligned}$$

On the BSP model, we can implement the reduce skeleton with two supersteps: one consists of Steps 1 and

2; the other consists of Step 3. Thus, the BSP cost is given as follows.

$$t_P^{\langle \mathrm{BSP} \rangle}(\mathsf{reduce}\ k) = \max_p \sum_{pr(i)=p} \left(L_i \times \frac{t_1(k)}{2} + D_i \times (-t_1(k) + t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r))) \right) + M \times t_1(\psi_n)/2 + M \times \frac{|\alpha| + |\gamma|}{2} \times g + 2l$$

4.3. Upwards Accumulate. Next, we develop an implementation of the uAcc skeleton called with function k and auxiliary functions $k = \langle \phi, \psi_n, \psi_l, \psi_r \rangle_u$. Similar to the reduce skeleton, let the type of input binary tree be $BTree\langle \alpha, \beta \rangle$ and the type of intermediate values be γ. The implementation of the uAcc skeleton is shown in Fig. 4.3.

Step 1. Local Upwards Accumulation. In the first step, we apply function uAcc_local to each segment and compute local upwards accumulation and reduction. This function puts the intermediate results to array *seg′* if a node has no m-critical node as descendants, since the result value is indeed the result of the uAcc skeleton. This function puts nothing to array *seg′* if a node is either the m-critical node or an ancestor of the m-critical node. Returned values are the result of local reduction and the array *seg′*.

In the computation of the uAcc_local function, ϕ and either of ψ_l or ψ_r are applied to each node of the m-critical node and its ancestors (D_i nodes), and k is applied to the other internal nodes ($(L_i/2 - D_i)$ nodes). We obtain the cost of the uAcc_local function as

$$t_1(\text{uAcc\_local}) = D_i \times (t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r))) + \left(\frac{L_i}{2} - D_i \right) \times t_1(k) \ .$$

Note that this cost is the same as that of reduce_local function.

Step 2. Gathering Results of Local Reductions to Root Processor. In the second step, we gather the results of the local reductions on the global structure *gt* of the root processor. From each leaf segment a value of type α is transferred, and from each internal segment a value of type γ is transferred. Since the number of leaf segments and the number of internal segments are $M/2$ respectively, the communication cost of the second step is

$$t_P(\text{Step 2}) = \frac{M}{2} \times c_\alpha + \frac{M}{2} \times c_\gamma \ .$$

Step 3. Global Upward Accumulation on Root Processor. In the third step, we compute the upwards accumulation for the global structure *gt* on the root processor. Function uAcc_global performs sequential upwards accumulation using function ψ_n. In uAcc_global, we apply function ψ_n to each internal segment of *gt*, and the cost of the third step is given as

$$t_1(\text{uAcc\_global}) = \frac{M}{2} \times t_1(\psi_n) \ .$$

Step 4. Distributing Global Result. In the fourth step, we send the result of global upwards accumulation to processors, where two values are sent to each internal segment and no values are sent to each leaf segment. Since all the values have type α after the global upwards accumulation, the communication cost of the fourth step is given as

$$t_P(\text{Step 4}) = M \times c_\alpha \ .$$

Step 5. Local Updates for Each Internal Segment. In the last step, we apply function uAcc_update to each internal segment. At the beginning of the function, the two values pushed in the previous step are pushed to the stack. These two values correspond to the results of children of the m-critical node. Note that in the last step we only compute the missing values in the segment *seg′*, which is given in the local upwards accumulation (Step 1).

```
uAcc(k, (gt, segs)) where k = ⟨φ, ψ_n, ψ_l, ψ_r⟩_u
  for i ← 0 to gt.size − 1: begin
    if pr(i) == p then (gt[i], segs'[i]) ← uAcc_local(k, φ, ψ_l, ψ_r, segs[i]); endif
  end
  gather_to_root(gt)
  if isRoot(p) then gt' ← uAcc_global(ψ_n, gt); endif
  distribute_from_root(gt')
  for i ← 0 to gt.size − 1: begin
    if pr(i) == p ∧ isNode(gt'[i]) then
      segs'[i] ← uAcc_update(k, segs[i], segs'[i], gt'[i]) endif
  end
  return (gt', segs')

uAcc_local(k, φ, ψ_l, ψ_r, seg)
  stack ← ∅; d ← −∞;
  for i ← seg.size − 1 to 0: begin
    if isLeaf(seg[i]) then seg'[i] ← seg[i]; stack ← seg'[i]; d ← d + 1; endif
    if isNode(seg[i]) then
      lv ← stack; rv ← stack;
      if d == 0       then stack ← ψ_l(lv, φ(seg[i]), rv); d ← 0;
      else if d == 1 then stack ← ψ_r(lv, φ(seg[i]), rv); d ← 0;
      else            seg'[i] ← k(lv, seg[i], rv); stack ← seg'[i]; d ← d − 1; endif
    endif
    if isCritical(seg[i]) then stack ← φ(seg[i]); d ← 0; endif
  end
  top ← stack; return(top, seg');

uAcc_global(ψ_n, gt)
  stack ← ∅;
  for i ← gt.size − 1 to 0: begin
    if isLeaf(gt[i]) then gt'[i] ← gt[i]; endif
    if isNode(gt[i]) then lv ← stack; rv ← stack; gt'[i] ← ψ_n(lv, gt[i], rv); endif
    stack ← gt'[i];
  end
  return(gt');

uAcc_update(k, seg, seg', (lc, rc))
  stack ← ∅; stack ← rc; stack ← lc; d ← −∞;
  for i ← seg.size − 1 to 0: begin
    if isLeaf(seg[i]) then stack ← seg'[i]; d ← d + 1; endif
    if isNode(seg[i]) then
      lv ← stack; rv ← stack;
      if d == 0       then seg'[i] ← k(lv, seg[i], rv); stack ← seg'[i];
      else if d == 1 then seg'[i] ← k(lv, seg[i], rv); stack ← seg'[i]; d ← 0;
      else            stack ← seg'[i]; d ← d − 1; endif
    if isCritical(seg[i]) then
      lv ← stack; rv ← stack;
      seg'[i] ← k(lv, seg[i], rv); stack ← seg'[i]; d ← 0;
    endif
  end
  return(seg');
```

FIG. 4.3. *Implementation of* uAcc *skeleton*

In this step, function k is applied to the nodes on the path from the m-critical node to the root node for each internal segment. Noting that the depth of the m-critical nodes is D_i, we can give the cost of UACC_UPDATE as

$$t_1(\text{UACC\_UPDATE}) = D_i \times t_1(k) \ .$$

In summary, using the functions defined so far, we can implement the uAcc skeleton. The cost of the uAcc skeleton is given as follows.

$$\begin{aligned} & t_P(\mathsf{uAcc}(k,t)) \\ & \quad = \max_p \sum_{pr(i)=p} t_1(\text{UACC\_LOCAL}) + t_P(\text{Step 2}) + t_1(\text{UACC\_GLOBAL}) \\ & \qquad + t_P(\text{Step 4}) + \max_p \sum_{pr(i)=p} t_1(\text{UACC\_UPDATE}) \\ & \quad = \max_p \sum_{pr(i)=p} \left(L_i \times \frac{t_1(k)}{2} + D_i \times (t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r))) \right) \\ & \qquad + M \times (3c_\alpha + c_\gamma + t_1(\psi_n))/2 \end{aligned}$$

On the BSP model, we can implement the uAcc skeleton with three supersteps: the first one consists of Steps 1 and 2; the second one consists of Steps 3 and 4; the last one consists of Step 5. Thus, the BSP cost is given as follows.

$$\begin{aligned} & t_P^{\langle \text{BSP} \rangle}(\mathsf{uAcc}\ k) \\ & \quad = \max_p \sum_{pr(i)=p} \left(L_i \times \frac{t_1(k)}{2} + D_i \times (t_1(\phi) + \max(t_1(\psi_l), t_1(\psi_r))) \right) \\ & \qquad + M \times t_1(\psi_n)/2 + M \times (3|\alpha| + |\gamma|)/2 \times g + 3l \end{aligned}$$

4.4. Downwards Accumulate. Finally, we develop an implementation and the cost of the dAcc skeleton called with a pair of functions (g_l, g_r) and auxiliary functions $(g_l, g_r) = \langle \phi_l, \phi_r, \psi_u, \psi_d \rangle_d$. Let the input binary tree have type $BTree\langle \alpha, \beta \rangle$, the accumulative parameter c have type γ, and the intermediate values for auxiliary functions have type δ. The implementation of the dAcc skeleton is shown in Fig. 4.4.

Step 1. Computing Local Intermediate Values. In the first step, we compute for each internal segment two local intermediate values, which are used in updating the accumulative parameter from the root node to the both children of the m-critical node. To minimize the computation cost, we first find the m-critical node and then compute two values only on the path from the root node to the m-critical node. We implement this computation by function DACC_PATH, in which the computation is done by a traversal on the array from right to left with an integer d instead of a stack. Two variables *toL* and *toR* are the intermediate values.

In this step we apply ψ_u twice and either ϕ_l or ϕ_r for each of the ancestors of the m-critical nodes (D_i nodes). Omitting some small constants, the cost of the DACC_PATH function is given as

$$t_1(\text{DACC\_PATH}) = D_i \times (\max(t_1(\phi_l), t_1(\phi_r)) + 2t_1(\psi_u)) \ .$$

Step 2. Gathering Local Results to Root Processor. In the second step, we gather the local results of the internal segments to the root processor. Since the two intermediate values have type δ and the number of internal segments is $M/2$, the communication cost in the second step is given as

$$t_P(\text{Step 2}) = M \times c_\delta \ .$$

The pair of local results from each internal segment is put to the array of the global tree structure gt.

Step 3. Global Downwards Accumulation. In the third step, we compute global downwards accumulation on the root processor. We implement this global downwards accumulation DACC_GLOBAL with a forward traversal using a stack. Firstly, the initial value of accumulative parameter is pushed to the stack, and then the

accumulative parameter in the stack is updated with the pair of local results given in the previous step. The result of global accumulation is the accumulative parameter passed to the root node for each segment.

The DACC_GLOBAL function applies function ψ_d twice for each internal segment in the global structure. The computational cost of the DACC_GLOBAL function is

$$t_1(\text{DACC\_GLOBAL}) = M \times t_1(\psi_d) \ .$$

Step 4. Distributing Global Result. In the fourth step, we distribute the result of global downwards accumulation to the corresponding processor. Since each result of global downwards accumulation has type γ, the communication cost of the fourth step is

$$t_P(\text{step 4}) = M \times c_\gamma \ .$$

Step 5. Local Downwards Accumulation. Finally, we compute local downwards accumulation for each segment. The initial value c' of the accumulative parameter is given in the previous step. Note that the definition of DACC_LOCAL function is just the same as the sequential version of the downwards accumulation on the serialized array if we assume the m-critical node as a leaf.

The local downwards accumulation applies functions g_l and g_r for each internal node. Since the number of the internal nodes is $L_i/2$, the computational cost of the DACC_LOCAL function is given as

$$t_1(\text{DACC\_LOCAL}) = \frac{L_i}{2} \times (t_1(g_l) + t_1(g_r)) \ .$$

Summarizing the discussion so far, the cost of the dAcc skeleton is given as follows.

$$\begin{aligned}
&t_P(\mathsf{dAcc}) \\
&\quad = \max_p \sum_{pr(i)=p} t_1(\text{DACC_PATH}) + t_P(\text{Step 2}) + t_1(\text{DACC_GLOBAL}) \\
&\qquad + t_P(\text{Step 4}) + \max_p \sum_{pr(i)=p} t_1(\text{DACC_LOCAL}) \\
&\quad = \max_p \sum_{pr(i)=p} \left(L_i \times \frac{t_1(g_l) + t_1(g_r)}{2} + D_i \times (\max(t_1(\phi_l), t_1(\phi_r)) + 2t_1(\psi_u)) \right) \\
&\qquad + M \times (c_\delta + t_1(\psi_d) + c_\gamma)
\end{aligned}$$

On the BSP model, we can implement the dAcc skeleton with three supersteps: the first one consists of Steps 1 and 2; the second one consists of Steps 3 and 4; the last one consists of Step 5. Thus, the BSP cost is given as follows.

$$\begin{aligned}
&t_P^{\langle \text{BSP} \rangle}(\mathsf{dAcc}) \\
&\quad = \max_p \sum_{pr(i)=p} \left(L_i \times \frac{t_1(g_l) + t_1(g_r)}{2} + D_i \times (\max(t_1(\phi_l), t_1(\phi_r)) + 2t_1(\psi_u)) \right) \\
&\qquad + M \times t_1(\psi_d) + M \times (|\delta| + |\gamma|) \times g + 3l
\end{aligned}$$

5. Optimal Division of Binary Trees Based on Cost Model. As we stated at the beginning of Sect. 3, locality and load balance are two important issues in developing efficient parallel programs in particular on distributed-memory parallel computers. When we divide and distribute a binary tree using the m-bridges, we enjoy good locality with large m while we enjoy good load balance with small m. Therefore, we need to find an appropriate value for m to achieve both good locality and good load balance.

First, we show relations among parameters of the cost model. From Lemma 3.5 and the representation of local segments in Fig. 3.2, we have

$$L_i \leq m \ . \tag{5.1}$$

```
dAcc(g_l, g_r, c, (gt, segs)) where (g_l, g_r) = ⟨φ_l, φ_r, ψ_u, ψ_d⟩_d
  for i ← 0 to gt.size − 1: begin
    if pr(i) == p ∧ isNode(gt[i]) then
      gt[i] ← dAcc_path(φ_l, φ_r, ψ_u, segs[i]); endif
  end
  gather_to_root(gt)
  if isRoot(p) then gt' ← dAcc_global(ψ_d, c, gt); endif
  distribute_from_root(gt')
  for i ← 0 to gt.size − 1: begin
    if pr(i) == p then segs'[i] ← dAcc_local(g_l, g_r, gt'[i], segs[i]); endif
  end
  return (gt', segs')

dAcc_path(φ_l, φ_r, ψ_u, seg)
  d ← −∞;
  for i ← seg.size − 1 to 0: begin
    if isLeaf(seg[i]) then d ← d + 1; endif
    if isNode(seg[i]) then
      if d == 0 then
        toL ← ψ_u(φ_l(seg[i]), toL); toR ← ψ_u(φ_l(seg[i]), toR);
      else if d == 1 then
        toL ← ψ_u(φ_r(seg[i]), toL); toR ← ψ_u(φ_r(seg[i]), toR); d ← 0;
      else
        d ← d − 1;
      endif
    endif
    if isCritical(seg[i]) then toL ← φ_l(seg[i]); toR ← φ_r(seg[i]); d ← 0; endif
  end
  return (toL, toR);

dAcc_global(ψ_d, c, gt)
  stack ← ∅; stack ← c;
  for i ← 0 to gt.size − 1: begin
    if isLeaf(gt[i]) then gt'[i] ← stack; endif
    if isNode(gt[i]) then
      gt'[i] ← stack; (toL, toR) ← gt[i];
      stack ← ψ_d(gt'[i], toR); stack ← ψ_d(gt'[i], toL);
    endif
  end
  return gt';

dAcc_local(g_l, g_r, c', seg)
  stack ← ∅; stack ← c';
  for i ← 0 to seg.size − 1: begin
    if isLeaf(seg[i]) then seg'[i] ← stack; endif
    if isNode(seg[i]) then
      seg'[i] ← stack; stack ← g_r(seg'[i], seg[i]); stack ← g_l(seg'[i], seg[i]); endif
    if isCritical(seg[i]) then seg'[i] ← stack; endif
  end
  return seg';
```

FIG. 4.4. *Implementation of* dAcc *skeleton*

Since the height of a tree is at least a half of the number of nodes, we obtain

$$D_i \leq L_i/2 \leq m/2 \ . \tag{5.2}$$

From Lemmas 3.6 and 3.7, the number of local segments M is bound with the number N of nodes and the parameter m as follows.

$$\frac{1}{2}\left(\frac{N}{m} - 1\right) \leq M \leq \frac{2N}{m} - 1 \tag{5.3}$$

By inequality (5.2), the general form of the cost can be transformed into the following simpler form by considering the worst case.

$$\max_p \sum_{pr(i)=p} (L_i \times t_l + D_i \times t_d) + M \times t_m \leq (\max_p \sum_{pr(i)=p} L_i) \times \left(t_l + \frac{t_d}{2}\right) + M \times t_m$$

We then bound the maximum number of nodes on a processor, $\max_p \sum_{pr(i)=p} L_i$. We distribute the local segment to processors so as to obtain good load balance, and one easy way to implement the load balancing is greedy distribution of the local segments from the largest one. By this greedy distribution, the difference between the maximum number of nodes $\max_p \sum_{pr(i)=p} L_i$ and the minimum number of nodes $\min_p \sum_{pr(i)=p} L_i$ is less than or equal to the maximum number of nodes in a segment. Since the maximum number of nodes in a local segment is m as stated in inequality (5.1) and the total number of nodes in the original binary tree is N, we can bound the maximum number of nodes distributed to a processor as follows:

$$\max_p \sum_{pr(i)=p} L_i \leq \frac{N}{P} + m$$

where P denotes the number of processors. By substituting this inequality to the cost, we can bound the cost of the worst case.

$$\max_p \sum_{pr(i)=p} (L_i \times t_l + D_i \times t_d) + M \times t_m \leq \left(\frac{N}{P} + m\right) \times \left(t_l + \frac{t_d}{2}\right) + M \times t_m \tag{5.4}$$

Now we want to minimize the worst-case cost given in the right-hand side of inequality (5.4). By substituting the parameter M (inequality (5.3)), the worst-case cost is bound with respect to m. We can bound the worst-case cost for smaller m as

$$\left(\frac{N}{P} + m\right) \times \left(t_l + \frac{t_d}{2}\right) + M \times t_m \leq \left(\frac{N}{P} + m\right) \times \left(t_l + \frac{t_d}{2}\right) + \frac{1}{2}\left(\frac{N}{m} - 1\right) \times t_m \ ,$$

and we can bound the worst-case cost for larger m as

$$\left(\frac{N}{P} + m\right) \times \left(t_l + \frac{t_d}{2}\right) + M \times t_m \leq \left(\frac{N}{P} + m\right) \times \left(t_l + \frac{t_d}{2}\right) + \frac{2N}{m} - 1 \times t_m \ .$$

From these bounds, we can minimize the worst-case cost for some value m in the following range.

$$\sqrt{\frac{t_m}{2t_l + t_d}}\sqrt{N} \leq m \leq 2\sqrt{\frac{t_m}{2t_l + t_d}}\sqrt{N} \tag{5.5}$$

This range of the parameter m is much smaller than that used in the previous studies [8, 18, 30]. In Sect. 6, we will show experiment results that support the range.

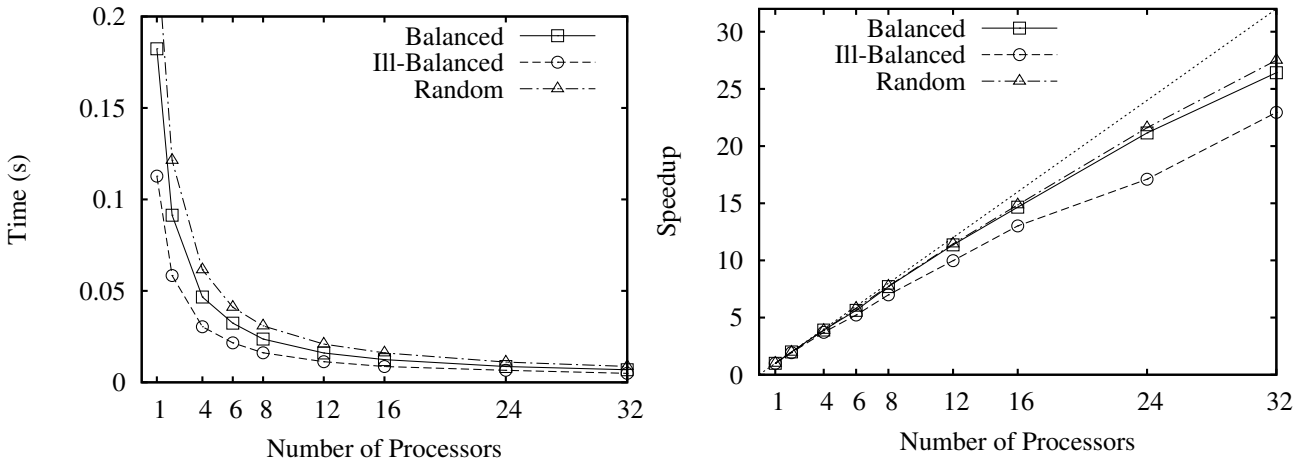


FIG. 6.1. *For the sum of node values, execution times and speedups against sequential program plotted to the number of processors. The parameter m for the division of trees is m = 53,600.*

6. Experiment Results. To confirm the efficiency of the implementation algorithm for binary-tree skeletons, we implemented binary tree skeletons in C++ and MPI and made several experiments. We used our PC-cluster of uniform PCs with two Pentium 4 2.4-GHz CPUs (one CPU is used for each PC) and 2-GByte memory connected with Gigabit Ethernet. The compiler and MPI library used are gcc 4.1.1 and MPICH 1.2.7, respectively.

We used the skeletal parallel programs of the two examples in Sect. 2. The input trees are (1) a balanced tree, (2) a randomly generated tree and (3) a fully ill-balanced tree, each having 16,777,215 ($= 2^{24} - 1$) nodes.

Figures 6.1 and 6.2 show the general performance of tree skeletons. Each execution time excludes the initial data distribution and final gathering. The speedups are plotted against the efficient sequential implementation of the program, which is implemented on the array representing binary trees based on the same sequential algorithm. As seen in these plots, our implementation shows good scalability even against the efficient sequential programs. By the m-bridges, the balanced tree is divided into leaf segments of the same size and internal segments consisting only of one node. Therefore, the overhead caused by parallelism is very small for the balanced binary tree, and the implementation achieves almost linear speedups against the sequential program. For the random tree, the average depth of the m-critical nodes is so small that the implementation achieves good performance close to that for the balance tree. The fully ill-balanced tree, however, is divided into leaf segments consisting of one node and internal segments with their m-critical node at the depth $D_i = L_i/2$. From the cost model and its parameters, the skeletal parallel program has overheads caused by the factor of depth of the m-critical nodes. In fact, the experimental results show that the skeletal parallel program runs slower for the fully ill-balanced tree than for the other two inputs.

To analyze the cost model and the range of the parameter m more in detail, we made more experiments for the randomly generated tree by changing the value of the parameter m. We measured the parameters t_l, t_d, and t_m of the cost model with a small tree with 999,999 nodes, and estimated the value of them as $t_l = 0.057$ μs, $t_d = 0.03$ μs, and $t_m = 71$ μs. By substituting the parameters, we can expect good performance of the skeletal program under the range $90{,}000 < m < 180{,}000$. Figure 6.3 (left) plots the execution times to the number of processors for three values of the parameter m. As we can see from this figure, the implementation achieves good performance for a wide range of m. Figure 6.3 (right) plots the execution times to the parameter m. This figure shows that the performance gets worse if the parameter m is too small ($m < 5{,}000$) or too large ($m > 200{,}000$). For the parameter m in the range above the skeletal program achieves near the best performance, and we conclude that the cost model and the estimation of the parameter m is useful for efficient implementations.

7. Related Work. Tree contraction algorithms, whose idea was first proposed by Miller and Reif [25], are very important parallel algorithms for efficient manipulations of trees. Many researchers have devoted themselves

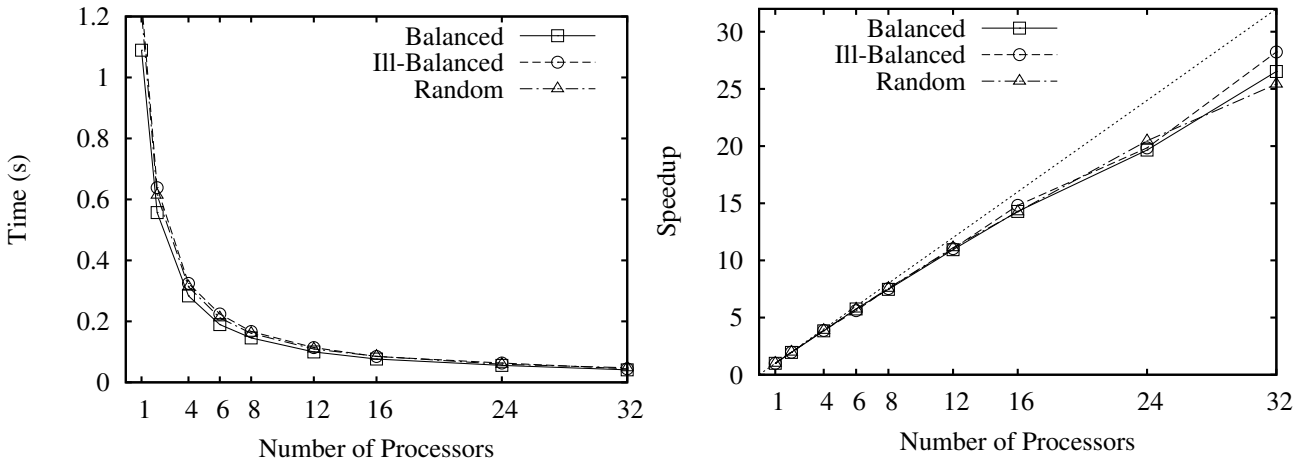


FIG. 6.2. *For the prefix numbering problem, execution times and speedups against sequential program plotted to the number of processors. The parameter m for the division of trees is $m = 53{,}600$.*

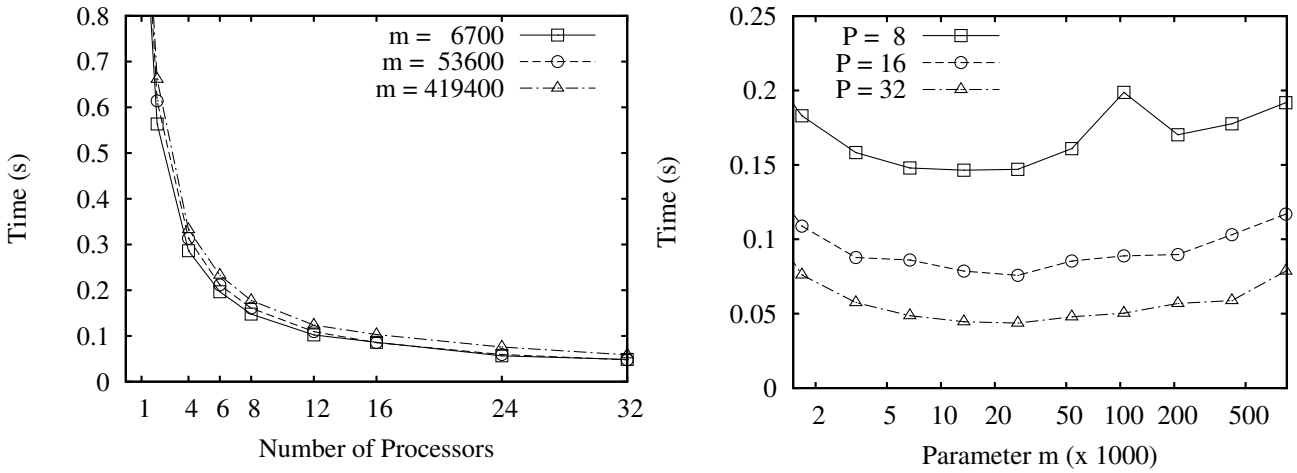


FIG. 6.3. *For the prefix numbering problem, execution times plotted to the number of processors and to the parameter m. The input trees are from the same randomly generated tree divided with different parameter m.*

to developing efficient implementations of tree contraction algorithms on various parallel models [1, 2, 3, 4, 5, 9, 13, 23, 24, 38]. Among them, Gibbons and Rytter developed a cost-optimal algorithm on CREW PRAM [9]; Abrahamson et al. developed a cost-optimal and practical algorithm on EREW PRAM [1]; Miller and Reif showed implementations on hypercubes or related networks [23, 24]; and recently more efficient implementations are discussed [2, 38] for symmetric multiprocessors (SMP) and chip-level multiprocessing (CMP). A lot of tree programs have been described by the tree contraction algorithms [3, 4, 9, 12, 17, 26, 27, 28, 29].

There have been several studies on the implementations of parallel tree skeletons [10, 11, 15, 18, 32, 33, 34]. Gibbons et al. [11, 33] have developed an implementation of tree skeletons based on tree contraction algorithms. Their algorithm can be used on many parallel computers, due to various implementation algorithms of tree contraction algorithms on various parallel computers. Skillicorn [34] and our previous paper [18] have discussed implementations of tree skeletons based on the division of trees. Compared with these implementation algorithms, our implementation is unique in terms of data structure of local segments for better sequential performance and the cost model supporting good division of trees. As far as we are aware, we are the first who implement tree skeletons as a parallel skeleton library. Our implementation of tree skeletons will be available as a part of SkeTo library [22]. Based on the implementation of the earlier work, Sato and Matsuzaki [31] improved its interface to support flexible manipulation of trees.

In terms of manipulations of general trees, which are formalized as parallel rose-tree skeletons [20], some of them are implemented efficiently in parallel [15, 32]. Sevilgen et al. [32] has shown an implementation algorithm for tree accumulations on general trees where rather strict conditions are requested for efficient implementation. Kakehi et al. [14] has developed an efficient implementation of tree reduction on general trees based on the serialized representation like XML formats.

8. Conclusion. In this paper, we have developed an efficient implementation of parallel tree skeletons. Not only our implementation shows good performance even against sequential programs, but also the cost model of the implementation helps us to divide a tree into segments with good load balance. The implementation is available as part of SkeTo library (`http://sketo.ipl-lab.org/`). One of our future work is to develop a profiling system to determine the parameter m more accurately.

REFERENCES

[1] K. R. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. M. Przytycka, *A simple parallel tree contraction algorithm*, J. Algorithm., 10 (1989), pp. 287–302.

[2] D. A. Bader, S. Sreshta, and N. R. Weisse-Bernstein, *Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs) (extended abstract)*, in Proceedings of 9th International Conference on High Performance Computing (HiPC 2002), Springer, 2002, pp. 63–78.

[3] R. P. K. Banerjee, V. Goel, and A. Mukherjee, *Efficient parallel evaluation of CSG tree using fixed number of processors*, in ACM Symposium on Solid Modeling Foundations and CAD/CAM Applications, May 19–21, 1993, Montreal, Canada, 1993, pp. 137–146.

[4] R. Cole and U. Vishkin, *The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time*, Algorithmica, 3 (1988), pp. 329–346.

[5] F. K. H. A. Dehne, A. Ferreira, E. Cáceres, S. W. Song, and A. Roncato, *Efficient parallel graph algorithms for coarse-grained multicomputers and BSP*, Algorithmica, 33 (2002), pp. 183–200.

[6] H. Deldari, J. R. Davy, and P. M. Dew, *Parallel CSG, skeletons and performance modeling*, in Proceedings of the Second Annual CSI Computer Conference (CSICC'96), 1996, pp. 115–122.

[7] K. Diks and T. Hagerup, *More general parallel tree contraction: Register allocation and broadcasting in a tree*, Theor. Comput. Sci., 203 (1998), pp. 3–29.

[8] H. Gazit, G. L. Miller, and S.-H. Teng, *Optimal tree contraction in EREW model*, in Proceedings of the Princeton Workshop on Algorithms, Architectures, and Technical Issues for Models of Concurrent Computation, 1987, pp. 139–156.

[9] A. Gibbons and W. Rytter, *An optimal parallel algorithm for dynamic expression evaluation and its applications*, in Proceedings of the 6th Conference on Foundations of Software Technology and Theoretical Computer Science, Springer, 1986, pp. 453–469.

[10] J. Gibbons, *Computing downwards accumulations on trees quickly*, in Proceedings of the 16th Australian Computer Science Conference, 1993, pp. 685–691.

[11] J. Gibbons, W. Cai, and D. B. Skillicorn, *Efficient parallel algorithms for tree accumulations*, Sci. Comput. Program., 23 (1994), pp. 1–18.

[12] X. He, *Efficient parallel algorithms for solving some tree problems*, in 24th Allerton Conference on Communication, Control and Computing, 1986, pp. 777–786.

[13] G. Jens, *Communication and memory optimized tree contraction and list ranking*, Tech. report, INRIA, Unité de recherche, Rhône-Alpes, Montbonnot-Saint-Martin, FRANCE, 2000.

[14] K. Kakehi, K. Matsuzaki, and K. Emoto, *Efficient parallel tree reductions on distributed memory environments*, in Proceedings of the 7th International Conference on Computational Science (ICCS 2007), Part II, Springer, 2007, pp. 601–608.

[15] K. Kakehi, K. Matsuzaki, K. Emoto, and Z. Hu, *An practicable framework for tree reductions under distributed memory environments*, Tech. Report METR 2006-64, Department of Mathematical Informatics, Graduate School of Information Science and Technology, the University of Tokyo, 2006.

[16] K. Matsuzaki, *Efficient implementation of tree accumulations on distributed-memory parallel computers*, in Proceedings of the 7th International Conference on Computational Science (ICCS 2007), Part II, Springer, 2007, pp. 609–616.

[17] K. Matsuzaki, Z. Hu, K. Kakehi, and M. Takeichi, *Systematic derivation of tree contraction algorithms*, Parallel Process. Lett., 15 (2005), pp. 321–336.

[18] K. Matsuzaki, Z. Hu, and M. Takeichi, *Implementation of parallel tree skeletons on distributed systems*, in Proceedings of The Third Asian Workshop on Programming Languages and Systems (APLAS'02), 2002, pp. 258–271.

[19] ———, *Parallelization with tree skeletons*, in Proceedings of the 9th International Euro-Par Conference (Euro-Par 2003), Springer, 2003, pp. 789–798.

[20] ———, *Parallel skeletons for manipulating general trees*, Parallel Comput., 32 (2006), pp. 590–603.

[21] ———, *Towards automatic parallelization of tree reductions in dynamic programming*, in Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2006), ACM Press, 2006, pp. 39–48.

[22] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu, *A library of constructive skeletons for sequential style of parallel programming*, in InfoScale '06: Proceedings of the 1st international conference on Scalable information systems, vol. 152 of ACM International Conference Proceeding Series, ACM Press, 2006.

[23] E. W. Mayr and R. Werchner, *Optimal routing of parentheses on the hypercube*, J. Parallel Dist. Com., 26 (1995), pp. 181–192.
[24] ———, *Optimal tree contraction and term matching on the hypercube and related networks*, Algorithmica, 18 (1997), pp. 445–460.
[25] G. L. Miller and J. H. Reif, *Parallel tree contraction and its application*, in Proceedings of 26th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, 1985, pp. 478–489.
[26] ———, *Parallel tree contraction, part 2: Further applications*, SIAM J. Comput., 20 (1991), pp. 1128–1147.
[27] G. L. Miller and S.-H. Teng, *Dynamic parallel complexity of computational circuits*, in Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing, ACM Press, 1987, pp. 254–263.
[28] ———, *Tree-based parallel algorithm design*, Algorithmica, 19 (1997), pp. 369–389.
[29] ———, *The dynamic parallel complexity of computational circuits*, SIAM J. Comput., 28 (1999), pp. 1664–1688.
[30] J. H. Reif, ed., *Synthesis of Parallel Algorithms*, Morgan Kaufmann Publishers, February 1993.
[31] S. Sato and K. Matsuzaki, *A generic implementation of tree skeletons*, International Journal of Parallel Programming, 44 (2016), pp. 686–707.
[32] F. E. Sevilgen, S. Aluru, and N. Futamura, *Parallel algorithms for tree accumulations*, J. Parallel Dist. Com., 65 (2005), pp. 85–93.
[33] D. B. Skillicorn, *Foundations of Parallel Programming*, vol. 6 of Cambridge International Series on Parallel Computation, Cambridge University Press, 1994.
[34] ———, *Parallel implementation of tree skeletons*, J. Parallel Dist. Com., 39 (1996), pp. 115–125.
[35] ———, *A parallel tree difference algorithm*, Inform. Process. Lett., 60 (1996), pp. 231–235.
[36] ———, *Structured parallel computation in structured documents*, J. Univars. Comput. Sci., 3 (1997), pp. 42–68.
[37] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl, *Questions and Answers about BSP*, Scientific Programming, 6 (1997), pp. 249–274.
[38] U. Vishkin, *A no-busy-wait balanced tree parallel algorithmic paradigm*, in Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2000), ACM Press, 2000, pp. 147–155.

Edited by: Frédéric Loulergue
Received: September 17, 2016
Accepted: January 17, 2017

DOI 10.12694/scpe.v18i1.1232
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 18, Number 1, pp. 35–49. http://www.scpe.org

TELOS: AN APPROACH FOR DISTRIBUTED GRAPH PROCESSING BASED ON OVERLAY COMPOSITION

PATRIZIO DAZZI*, EMANUELE CARLINI†, ALESSANDRO LULLI‡, AND LAURA RICCI§

Abstract. The recent years have been characterised by the production of a huge amount of data. As matter of fact, human being, overall, is generating an unprecedented amount of data flowing in multiple and heterogeneous sources, ranging from scientific devices, social network, business transactions, etc. Data that is usually represented as a graph, which, due to its size, is often infeasible to process on a single machine. The direct consequence is the need for exploiting parallel and distributed computing frameworks to face this problem. This paper proposes Telos, an high-level approach for large graphs processing. Our approach takes inspiration from overlay networks, that is a widely adopted approach for information dissemination, aggregation and computing orchestration in highly distributed systems. Telos consists of a programming framework supporting the definition of computations on graphs as a composition of overlays, each devoted to a specific aim. The framework is implemented on the top of Apache Spark. A set of experimental results is presented to give a clear evidence of the effectiveness of our approach.

Key words: Programming Models, Distributed Architectures, Self-Organisation

AMS subject classifications. 68M14, 68N19, 68W15

1. Introduction. Our world contains an unimaginably vast amount of digital information which is getting ever vaster rapidly. In fact, the amount of data produced every day by human beings is far beyond what has ever been experienced before. According to some statistics, the 90 percent of all the data in the world available in May 2013 has been generated over 2012 and 2013. As a further example, in 2012 were created, every day, 2.5 exabytes (2.5×10^{18}) of data [33] which comes from multiple and heterogeneous sources, ranging from scientific devices to business transactions. A significant part of this data is gathered and then modelled as a graph, such as social network graphs, road networks and biological graphs.

A careful analysis and exploitation of the information expressed by this data makes it possible to perform tasks that previously would be impossible to do: to detect business trends, to prevent diseases, to combat crime and so on. If properly managed, data can be used to unlock new sources of economic value, provide fresh insights into science and hold governments to account. However, the vastity of this data makes infeasible its processing by exploiting the computational and memory capacity of a single machine. Indeed, a viable solution to tackle its analysis relies on the exploitation of parallel and distributed computing solutions.

Many proposal for the parallel and distributed processing of large graphs have been designed so far. However, most of the methodologies currently adopted fall in two main categories. On the one hand, the exploitation of low-level techniques such as send/receive message passing or, equivalently, unstructured shared memory mechanisms. This is the way traditionally adopted to process large dataset, usually with parallel machines or clusters. Unfortunately, this approach leads to few issues. Low-level solutions are complex, error-prone, hard to maintain and their tailored nature hinder their portability.

On the other hand, a wide use of the MapReduce paradigm [19] can be observed, which has been inspired by the well-known map and reduce paradigms that is part of the algorithmic skeletons, which across the years have been provided by a number of different frameworks [37, 32, 18, 4, 2, 3, 13]. MapReduce-based solutions often exploit the MapReduce paradigm in a "off-label" fashion, i.e., in ways and contexts that are pretty different from the ones it has been conceived to be used in. Some of the most notable implementations of such paradigm are frequently used with algorithms which could be more fruitfully implemented with different parallel programming paradigms or different ways to orchestrate their computation. This is especially true when dealing with large graphs [29]. In spite of this, some MapReduce based frameworks achieved a wide diffusion due to their ease of use, detailed documentation and very active communities of users.

*Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", National Research Council of Italy, Pisa, Italy, (patrizio.dazzi@isti.cnr.it).
†Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", National Research Council of Italy, Pisa, Italy, (emanuele.carlini@isti.cnr.it).
‡CS Department, University of Pisa, Pisa, Italy, (lulli@di.unipi.it).
§CS Department, University of Pisa, Pisa, Italy, (ricci@di.unipi.it).

Recently, a few solutions have been proposed to let the MapReduce frameworks more suitable for performing analysis on large graphs in a more natural way. Some of them have been inspired by the BSP bridging model [44] that resulted in a good approach to implement certain algorithms whose MapReduce implementation would be either too complex or counter-intuitive. Malewicz *et al.* presented this approach when proposing the Pregel framework [29]. It provides the possibility to describe graph processing applications from the viewpoint of a graph vertex, which processes independently and can only access its local state and is aware only of its neighbourhood.

In this paper we propose Telos[1], an approach to process graphs by leveraging the vertex-centric approach to provide a solution that focuses on the orchestration of nodes and their interactions. Our approach takes inspiration by the similarities existing between large graphs and massively distributed architectures, e.g., P2P systems. In fact, many of these solutions orchestrate the computation and spread the information exploiting *overlay networks*. Essentially, an overlay is a virtual network, built upon the existing physical network, connecting nodes by means of logical links established according to a well-defined goal. The building blocks of overlays match the key elements of graphs. Vertices can be seen as networked resources and edges as links.

We believe that by means of these similarities overlay-based solutions can be fruitfully ported and exploited for graph processing. We already presented a first analysis about the applicability of Telos to solve graph partitioning [8]; we also gave a brief description of its conceptual architecture [26] and its programming model [9]. This paper gives a consolidated presentation of Telos. Along with a more detailed description of the approach, we present an advanced application scenario showing the advantages deriving from the exploitation of our proposed approach. More in details, the contributions of this paper are the following:

- the definition of a *high-level* programming approach based on the composition of overlays, targeting computations on large graphs;
- the presentation of *our framework* built on top of Apache Spark [50] providing both a high level API to define custom layers and some built-in layers. This gives the possibility to dynamically define graph topologies different from the original one, so enabling each vertex to choose the most promising neighbours to speed-up the convergence of the distributed algorithm;
- an extensive proof-of-concept demonstrator to assess the feasibility of Telos;
- a case study where we show how to port a popular graph partitioning algorithm in Telos. In addition, we enhanced such algorithm by means of the Telos' layered architecture showing through an experimental testbed the improvements on the quality of the obtained results.

The remainder of this paper is organised as follows. In Section 2 we discuss the current related work on distributed graph processing framework, with a focus on those able to evolve the graph during the computation. Section 3 presents the main design choices and the structure of Telos as well as its programming model and architecture. We evaluate our approach in Section 4 by sketching the Telos implementation of an overlay building protocol and a balanced graph partitioning case study. Finally, Section 5 reports some final remarks and further ideas for future work.

2. Related Work. Comprehensive surveys of widely adopted distributed programming models targeting graphs, and of the derived frameworks is presented in several survey contributions, such as by McCune *et al.* [34], Kalavri *et al.* [21], Doekemeijer *et al.* [16]. Most of the current frameworks provide a set of basic features for distributed graph processing, while several approaches have been developed with the goal of extending and/or optimizing the basic frameworks. In this section we focus on the approaches that are closely related to our work.

Among the several existing distributed programming frameworks focusing on graphs, a notable amount are characterised by the *vertex-centric* programming model. However, not all of them allow graph mutation during the execution, which is a core feature in our proposed approach.

Pregel [29] is the Google framework for distributed graph processing, and the one that de-facto defined the vertex-centric model. Pregel has been inspired by the BSP model [44], in which computations are organised in sequential supersteps; in Pregel, like in BSP, during each superstep each vertex executes a specific function. Vertices communicate in a message-passing model, by sending and receiving messages from/to other vertices.

[1] https://github.com/hpclab/telos

Apache Giraph [12] is another vertex-centric framework initially developed by Yahoo!. Conversely from Pregel, Giraph is open-source and runs over Hadoop.

More recent solutions are aimed at optimising the basic vertex-centric model provided by Pregel. GPS [41] provides a few interesting optimisations specifically targeted to vertex-centric graph processing. These solutions include an adaptive rescheduling of the computation, essentially aimed at performing the computation sequentially on the master node when the number of active nodes is below a given threshold. In such a case the exploitation of parallelism is no longer justified but just became an overhead. Another optimisation consists in merging several vertices together to form supervertices to reduce the communication cost. Both of these optimisations focus on reducing the completion time and the communication volume by instrumenting the support with automated recognition features, activated to speed-up the computation.

Another interesting approach, which goes beyond classical vertex-centric solutions, has been proposed by Tian *et al.* [43] and it is called as *partition-centric*. Their idea is to shift the reasoning from the single vertex perspective to the viewpoint of a set of vertices aggregated into a partition, in order to reduce the inter-vertex communication and to accelerate the convergence of vertex-centric programs. By means of such a change of perspective they have been able, on selected problems, to yield a notable speeds-up in the computation. The partition-centric model has been further specialized in other works, such as Simmhan *et al.* [42] and Yan *et al.* [48]. These proposals and our approach are based on the idea of pushing, by design, a shift in the paradigm that allows the implementation of more efficient solutions.

Distributed Graph Partitioning. Graph partitioning is an NP-Complete problem well known in graph theory. Distributed graph partitioning assumes the impossibility to access the entire graph at once, with the graph distributed among various workers nodes. In the context of a MapReduce computation, several approaches have considered the problem of distributed graph partitioning.

JA-BE-JA [40] is a distributed approach based on iterations of local nodes swapping to obtain a partition of the graph. Such an approach has been originally designed for extremely distributed graphs, in which a node can be associated with a vertex of the graph, but can be ported to MapReduce platforms with some tweaking [8]. Sheep [30] consists in a distributed graph partitioning algorithm that reduces the graph to an elimination tree to minimizes the communication volume among the workers of the MapReduce computation. Spinner [31] employs an heuristics based on the label propagation algorithm for distributed graph partitioning. A penalty function is applied to discourage the movements of nodes toward larger partitions. Similarly, Meyerhenke et al. [35] employs a size-constrained label propagation targeting heavily clustered and hierarchical complex networks, such as social networks and web graphs.

3. From vertices to overlays. The vertex-centric approach, also known as Think Like A Vertex (TLAV), is currently being exploited in several applications aimed at computing big data analysis involving large graphs, and widely used by both academia and industry. Many popular implementations of the TLAV model [29, 12] are based on a synchronized execution model according to the well-known BSP approach [44]. The core of BSP computations consists of three main pillars:

- *Concurrent computation*: Every participating computing entity (a vertex in this case) may perform local computations, i.e., computing by making use of values stored in the local memory/storage.
- *Barrier synchronisation*: After the termination of its local computation, every vertex is blocked until all the other vertices composing the graph have reached the same stage.
- *Communication*: All the vertices exchange data between themselves, in a point-to-point fashion, after they reach the synchronization barrier. Each message prepared during superstep S is received by the recipient at superstep $S + 1$.

From an operative viewpoint, during a superstep, each vertex receives messages sent in the previous iteration and executes a user-defined function that can modify its own state. Once the computation of such a function has terminated, it is possible for the computing entity to send messages to other entities.

To orchestrate the computation, the BSP model relies on a synchronisation barriers occurring at the end of every superstep. The same occurs in TLAV models, even if the vertex centric abstraction could, in principle, be realized with other computation models that do not rely on a strong synchronisation. Indeed, the programming approach that characterises the vertex-centric model, in which programmers are in charge of coding the behaviour of each single element, which globally cooperates to contribute to the whole computation, is neither

new nor exclusive of this kind of solutions. For instance, it is also the approach adopted in actor-based models.

The correspondence with the superstep defined by the BSP model is realized as the following. The computation is described by the method *compute()* described in Section 3.1.1. Regarding the communication, the messages are generated by the compute method. A message can be inter-vertex, intra-vertex or extra-protocol as defined in Section 3.1.2. The actual communication is materialized in the reduce phase. The synchronisation is enforced by the Spark framework, which assures that an iteration begins only when all nodes have executed the reduce of previous iteration.

In some ways, programming according to the vertex-centric model also recalls the definition of epidemic (or gossip) computing [5, 36, 15, 9, 17]. A widely used approach in massively distributed systems leading nodes of a network to work independently but sharing a common, global, aim. Indeed, according to a common strategy followed by many existing gossip protocol, nodes build logic network overlays to exploit for information exchange. An overlay consists of a logical communication topology, built over an underlying network, which is maintained and used by nodes. Across the years, several overlay-based protocols have been proposed for data sharing, diffusion, aggregation and for computation orchestration, as well. Usually, these solutions are adopted in highly distributed systems, e.g., massively multiplayer games [10], distributed replica management [22], distributed resource discovery [14, 7], etc. Most of these overlay-oriented solutions are layered [24] and exploit the different overlays to achieve very different tasks. Overlay-based algorithms are characterised by several different interesting features:

- *Local knowledge*: algorithms for overlays are based on local knowledge. During the computation each node relies only on its own data and the information received from its neighbours, resulting in a reduced access to the network and consequently improving the scalability of the approach.
- *Multiple views*: multi-layer overlays have been a successful trend [20]. These approaches build a stack of overlays, each organised accordingly to a specific goal. By means of their combination more complex organisations can be achieved, fostering an efficient and highly targeted dissemination of information.
- *Approximate solutions*: since overlays are usually based on an approximated knowledge on the graph, several algorithms designed for overlays are conceived to deal with approximated data. Usually, solutions are achieved by these approaches in an incremental and iterative way, allowing to achieve in a flexible way approximated solutions.
- *Asynchronous approach*: differently from vertex-centric algorithm, gossip-based approaches operate in an unreliable, not synchronized environment, in which messages can be received later than expected and their content be not consistent. Due to this, gossip protocols are able to work under the assumption of approximate and incremental results.

Our approach is aimed at introducing techniques for large graph processing borrowed from solutions targeting massively distributed systems, for instance solutions defined within the scope of Peer-to-peer computing area, into large graph processing framework. We believe that this approach could be useful to support large graph processing by means of the composition of solutions based on multiple-layer graphs, eventually combined to define more effective graph processing algorithms.

3.1. Telos. Telos builds upon the overlay-based approach to provide a tool aimed at large graph processing whose programming model is organised according to the vertex-centric approach for graph processing.

A set of protocols, one for each network overlay, is associated with each vertex. For each overlay, each vertex maintains a local context and a neighbourhood. The former represents the "state" of the vertex w.r.t. a given overlay, the latter represents the set of vertices that are exchanging messages with such a vertex by means of that overlay, according to the associated protocol. Both the context and the neighbourhood can change during the computation and across the supersteps, given the possibility of building evolving "graph overlays".

Telos brings to the vertex-centric model many of the typical advantages of a layered architecture:

- *modularity*: protocols can be composed and additional functions can be obtained by stacking different layers;
- *isolation*: changes occurring in a overlay, by means of a protocol, do not affect the other protocols;
- *reusability*: protocols can be reused for many and possibly different computations.

Telos supports programmers in combining different protocols. Each protocol is managed independently and all the communications and the organisation of the records are all in charge of Telos.

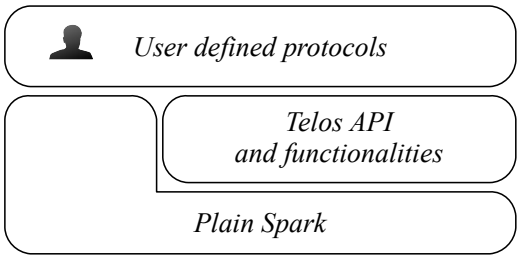


FIG. 3.1. *Telos integration with Spark*

TABLE 3.1
The Protocol Interface

function	**description**
getStartStep() : Int getStepDelay() : Int	defines the first step on which `compute()` is called defines the delay in terms of steps between two successive calls of `compute()`
compute(ctx:Context, mList:List[Message]) : (Context, List[Message])	contains the business code of a protocol, e.g., message handling, data precessing, etc.

In its current implementation, the Telos framework is built on top of Spark [50]. One of the key aspects of this tool are the Resilient Distributed Dataset (RDD) [49], on top of which are natively defined collective operations such as map, reduce and join. By relying on these operations, Telos provides its own TLAV abstraction. All the tasks for managing RDDs are transparent to the programmers and managed by Telos.

Figure 3.1 shows the integration of the Telos framework with Spark. The framework coordinates the protocols and masquerades the underlying support to ease the application development. The framework handles all the burden required to create the initial set of vertices and messages and provides to each vertex the context it requires for the computation. Communications are completely hidden to the programmers as well. Telos handles data dispatch to the target vertices.

3.1.1. Protocols. Protocols are first-class entities in Telos. They are the abstractions that orchestrate the computation and organise the topologies of each overlay. Protocols manage the context and the neighborhood of all the vertices. In fact, each protocol is in charge of:

- modifying the state associated to each vertex;
- defining a representation describing the state of the vertices, which is eventually sent as messages to the other vertices;
- defining custom messages aimed at supporting the overall orchestration of the nodes;

Table 3.1 describes the protocol interface, i.e., the main methods provided by the interface of a protocol (for the full table please refer to our former papers focusing on the description of such an interface [26, 9]). In a nutshell, the Protocol interface abstracts the structure of the computation running on each vertex. Basically, the core logic of a Protocol is contained in the `compute()` method. Once called, say at superstep S, the `compute()` method can access the whole state, associated to the vertex, and to the messages received by such vertex during the step S - 1. The output of the `compute()` method is a new vertex context and a set of messages to be dispatched to the target vertices.

The termination of a Protocol is coordinated by a "halt" vote. At the very end of its computation each vertex votes to halt, and when all vertices voted to halt, the computation terminates.

The frequency at which a protocol is activated (w.r.t. the supersteps) is regulated by `getStartStep()` and `getStepDelay()` methods. These methods are useful when a computation involves different protocols characterised by different convergence time, i.e., the amount of steps it needs to converge to a useful result. By means of these methods it is possible to regulate the activations of the protocols with respect to the amount of

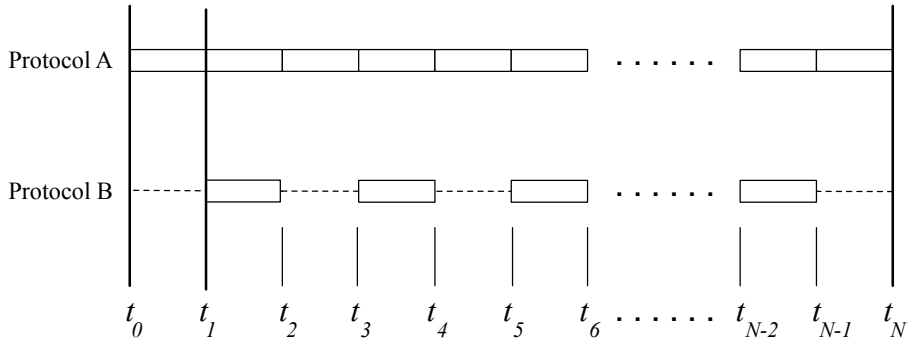


FIG. 3.2. *The effect of* `getStartStep()` *and* `getStepDelay()` *methods*

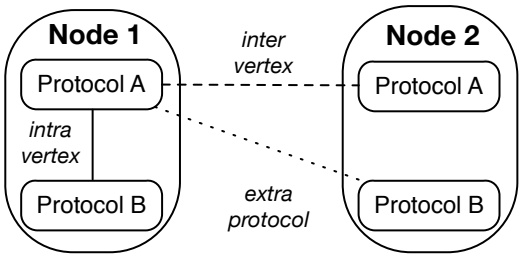


FIG. 3.3. *Telos interactions*

elapsed supersteps.

Figure 3.2 graphically depicts the behaviour of these methods. In the figure, *Protocol A* is activated at each superstep, i.e., if invoked, its implementation of `getStepDelay()` method, will return 1, whereas the very same call on *Protocol B* will return 2. In a pretty similar fashion the method `getStartStep()` drives the *first* activation of a protocol. Going back to the aforementioned figure, when called on *Protocol A* it will return 0, whereas it will return 1 if called on *Protocol B*.

3.1.2. Interactions. Telos enables three different types of interactions that involve the vertices belonging to the graph.

- *intra-vertex*: the internal access made by a given vertex, when executing a certain protocol, to the context associated at a different protocol.
- *inter-vertex*: the remote access made by a given vertex, when executing a certain protocol, to the context of another vertex, associated to the same protocol.
- *extra-protocol*: the remote access made by a given vertex, when executing a certain protocol, to the context of another vertex, associated to a different protocol.

Figure 3.3 shows an example of the aforementioned interactions. In the example, the Protocols A and B of Node 1 have an inter-vertex interaction. Protocol A of Node 1 and the same protocol on Node 2 have a inter-vertex interaction. Finally, Protocols A of Node 1 and protocol B of Node 2 have a inter-vertex interaction.

To summarise, the computation of TELOS is defined as the execution of a sequence of supersteps. At each superstep, the messages from the previous superstep are collected, a subset of the protocols are executed and both inter-vertex and extra protocol messages are sent, afterward the synchronisation barrier is executed.

3.1.3. Built-in Protocols. To evaluate the effectiveness of porting approaches from massively distributed system into the distributed large graph analysis we developed two protocols with Telos, which we briefly present in this section.

- *Random protocol.* The aim of this protocol is to provide to each vertex with a random vertex identifier upon request. The vertex identifier must be taken uniformly and randomly in the space of identifiers of graph vertices. It is implemented according to a random peer sampling approach [45]. From an

operative viewpoint, the random peer sampling protocol requires that each vertex periodically shares information about its neighbourhood with a random vertex, that will do the same. At the end, each vertex remains with a neighbourhood composed by k vertices, with k size of the neighbourhood.

- *Ranking-function protocol.* Highly distributed solutions exploit ranking functions to create and manage overlays built according to a specific topology [20, 46]. The ranking function determines the similarities between two vertices and the topology is built accordingly. In Telos we implemented a generic ranking protocol able to take as input the ranking function to realise the consequent topology.

4. Evaluation. To assess the evaluation of our approach we conducted several experiments. First, in Section 4.1 we provide a proof-of-concept implementation that validates our framework by implementing a stack composed by the two protocols, instantiating the random and the ranking-function protocols discussed above. Then, in Section 4.3 we evaluate the scalability provided by our implementation of Telos.

All the experiments have been conducted on a cluster running Ubuntu Linux 12.04 consisting of 4 nodes (1 master and 3 slaves), each equipped with 16 Gbytes of RAM and 8-cores. The nodes are interconnected via a 1 Gbit Ethernet network.

4.1. A Proof of Concept Implementation. The aim of this experiment is to verify the correctness of Telos when dealing with a layered composition of protocols. We built a two layered structure composed of a random protocol layer and a ranking protocol layer. We generated a set of points in 3D space and we assigned to each node a point. The points are generated in order to form a torus. Then, the neighborhood of each node is randomly initialized and the execution started. By using the ranking protocol, we iteratively connect nodes that are close to each other considering the euclidean distance. At the end of the execution, each node is connected to the most similar nodes according to the euclidean distance forming a torus-shaped graph.

We tested the implementation on a graph made of $20K$ vertices. The results in Figure 4.1 show the evolution of the graph in different iterations. Note, we plot the graphs using Gephi[6] and the default layout for placing the nodes. Due to this, although the point assigned to a vertex does not change it may happen that the same point is in different positions in the figures. The topology recalls the shape of a torus already at super-step 10. At super-step 20 no edges connect "distant" vertices in the torus. This experiment shows that, if properly instrumented, Telos can correctly manage multiple layers to organize network overlays according to user-defined ranking functions.

In the following, to have a deeper understanding of Telos, we present a pseudo-code implementation of the protocols of this experiment. The pseudo-codes are inspired by the original Scala[38] implementation. Code Fragment 1 presents the implementation of the random protocol. The idea is to randomly generate a number upon request on the method `getVertexId()`. The generated numbers must be in the range $[startRange, endRange]$. Where, in general, *startRange* is equal to 0 and *endRange* is equal to the number of nodes in the graph. The main issue is that each machine executing the protocol must be able to call such method. To do this, in the initialization (`init` method) we create two broadcast variables initialized with the values representing the required range. The RandomContext contains the logic to generate a random number and it is initialized with the two broadcast variables. In this experiment we use the convention to set the ids of the nodes in the range $[1, 20000]$ without missing values. Finally, the random protocol is initialized with such values and the ranking protocol makes use of the method `getVertexId()` to select a node identifier in order to communicate with a different and random node in each iteration.

Instead, code Fragment 2 presents the pseudo-code of the implementation of the ranking protocol. The implementations of `beforeSuperstep`, `afterSuperstep`, `init` and `createInitMessage` are not given because not required for this kind of Protocol. Specifically, the RankingProtocol extends the Protocol interface described in Table 3.1. We instruct the framework to start this protocol at the beginning of the computation with `getStartStep` equals to 1 (Line 3) and to run in each step of the computation with `getStepDelay` (Line 4) equals to 1. The `createContext` method (Line 6) is called at the beginning of the computation to initialize the state on each vertex. The data returned can be any custom type to allow the definition of custom data structures to record all the data needed by the protocols.

The ranking protocol uses the custom defined *RankingContext* to save the coordinate position of a point represented by the vertex and the neighbourhood of the vertex. Recall that Telos calls, for the active protocols, in each super-step the `compute()` method for each active vertex (Line 19) providing both a vertex context *ctx*

Code Fragment 1: `RandomProtocol extends Protocol`

```
class RandomProtocol extends Protocol
{
   def getStartStep() = return 1
   def getStepDelay() = return 1

   def init(sc:SparkContext) =  {
       startRangeBroadcast = sc.broadcast(startRange)
       endRangeBroadcast = sc.broadcast(endRange)
   }

   def compute[RandomContext](ctx:RandomContext,
        mList:List[Message]):(RandomContext, Message) = {
       return (ctx, List())
   }

   def createContext[RandomContext](row:Array[String]):
       RandomContext = {
        return new RandomContext(startRangeBroadcast,
                                 endRangeBroadcast)
   }
}

class RandomContext(val startRngBrd: Broadcast[Long],
          val endRngBrd:Broadcast[Long]) extends VertexContext {
   def getVertexId(): Long = {
       return (random.nextDouble() * (endRngBrd.value -
              startRngBrd.value)).toLong+startRngBrd.value
   }
}
```

and the messages received by the vertex *mList*. In this case, the vertex makes an *intra-vertex* communication to access the context of a different protocol on the same vertex and retrieve the *RandomContext*. Such context is used to choose randomly a counterpart to communicate with and exchange information. In addition, for each message received the vertex prepares a response message. In the last part of the method the vertex orders all the data received and keeps the vertices that are closer to the assigned 3D point. The value k is a custom parameter and in this experiment it is initialized to 5. The value returned by the `compute` method is the new vertex context and a list of messages to be dispatched by Telos to the specified vertices.

4.2. Case Study: Balanced Graph Partitioning. JA-BE-JA [40] is a distributed algorithm for computing the balanced k-way graph partitioning problem and it works as follows. Initially, it creates k logical partitions on the graph by randomly assigning colours to each vertex, with the number of colours equals to k. Then, each vertex attempts to swap its own colour with another vertex according to the most dominant colour among its neighbours, by: (i) selecting another vertex either from its neighbourhood or from a random sample, and (ii) considering the "utility" of performing the colour swapping operation. If the colour swapping would decrease the number of edges between vertices characterised by a different colour, then the two vertices swap their colour; otherwise, they keep their own colours.

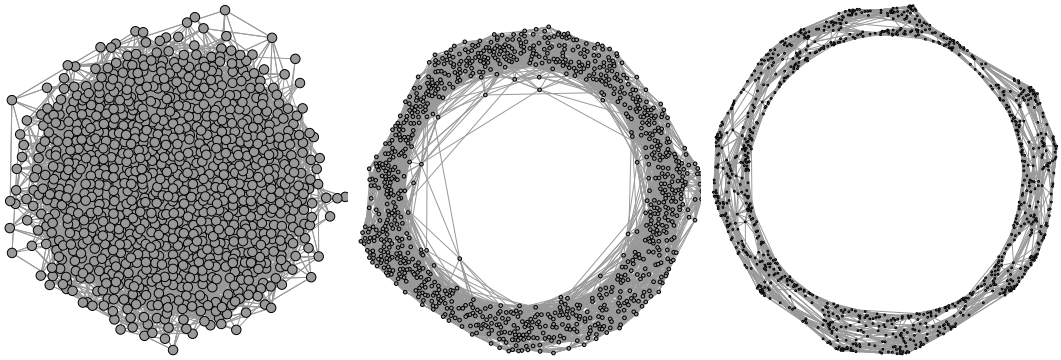
In a previous work [8] we presented an implementation of JA-BE-JA in Apache Spark outlining the adaptations that have been required in order to efficiently adapt the algorithm to match a BSP-like structure. In its original formulation, JA-BE-JA assumes that each vertex has complete access to the context of its neighbours and also their neighbourhood. To enforce this assumption, we initially introduced specific messages to retrieve

Code Fragment 2: `RankingProtocol extends Protocol`

```
1  class RankingFunction extends Protocol
2  {
3     def getStartStep() = return 1
4     def getStepDelay() = return 1
5
6     def createContext[RankingContext](row: Array[String]) :
7     RankingContext = {
8         var context = new RankingContext
9         context.x = row(0)
10        context.y = row(1)
11
12        for (i <- 2 until row.length)
13        {
14            context.neighbour.add(row(i))
15        }
16        return context
17    }
18
19    def compute[RankingContext](ctx:RankingContext,
20                mList:List[Message]):(RankingContext,Message) = {
21        var message = List()
22        val randomContext = ctx.accessProtocol("RandomProtocol")
23        val rcpt = randomContext.getVertexId()
24        val buffer = ordByEucDis(rcpt, ctx.neighbour)
25        message = message ++ new Message(rcpt, getTopK(buffer))
26
27        var newData = List()
28        foreach(msg : mList)
29        {
30            newData = newData ++ msg.content
31            val buffer =
32               ordByEucDis(msg.sender,ctx.neighbour++msg.content)
33
34            message =
35               message ++ new Message(msg.sender,getTopK(buffer))
36        }
37
38        val myBuf = ordByEucDis(ctx.self,ctx.neighbour++newData)
39        return (new RankingContext(ctx.self, myBuf),message)
40    }
41 }
```

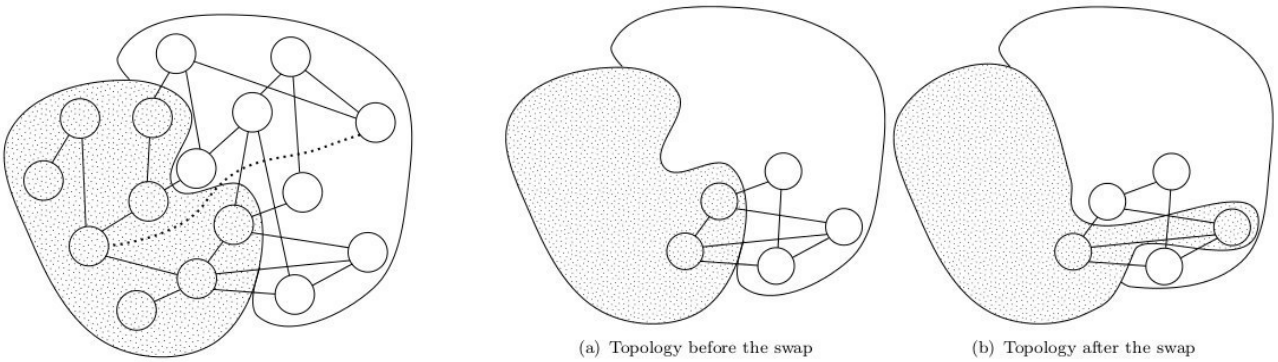
the neighbourhood of any vertex. However, we noticed that forcing a strong consistency of such information slows down the performance too much. As a consequence, we provided an alternative implementation (called GP-SPARK) that introduces a degree of approximation to accelerate the computation, in which vertices piggy-back their neighbourhood information in other messages. This mechanism causes the vertices to apply the local heuristics on possibly stale data, but increases the performance of the original approach providing a comparable quality of results with respect to the original version.

The original GP-SPARK works with a two layered architecture composed as follows: (i) the *colour swapping*



(a) Superstep 0 (b) Superstep 10 (c) Superstep 20

FIG. 4.1. *Evolution of the torus overlay at different Supersteps*



(a) Topology before the swap (b) Topology after the swap

FIG. 4.2. *Swap in the stain metaphor of the JA-BE-JA topology*

protocol, that attempts the colour swapping targeting either a vertex from the local neighbourhood or from the random sampling, and (ii) the random sampling protocol that provides some random vertices to the colour swapping protocol. Here, we present GP-TELOS, an improved version of GP-SPARK that introduces a new layer with the *border ranking protocol* aimed to boost the quality of results. The objective is to give to JA-BE-JA the possibility to select from a better set of vertices when attempting colour swapping.

Recall, the layered approach exploits orchestration between several protocols relying over the intra-communication facility. GP-TELOS adds a new layer that interacts with the colour swapping one. First of all, the colour swapping's layer is initialized with the input graph topology. Then, we have a random layer to provide to each vertex with some long range links that are created through the retrieve of a small sample of random nodes for each vertex. This layer is refreshed with new vertices in each iteration.

As shown in Figure 4.2.a, the colour swapping layer could be thought as a set of stains which interact with each other *moving* like fluids. In this metaphor, the long range links fold the stains in a N dimensional space, where N is the number of links maintained in the random layer. Such links allow the interaction between areas of the stains that are not topologically near each other in the color swapping layer. The JA-BE-JA algorithm orchestrates those stains in order to minimize the edge-cut. Whenever a swap happens is like a stain flows rolling to the neighbouring stain, as shown in Figure 4.2.b.

In GP-TELOS we introduce an additional layer of the *ranking* kind. This relies over the definition of a ranking function. The idea is to abstract the property of *borderness* of each vertex in the color swapping layer. Whenever 2 vertices are on the borders the swap could take place. Figuratively the vertices cross the boundaries of the stain.

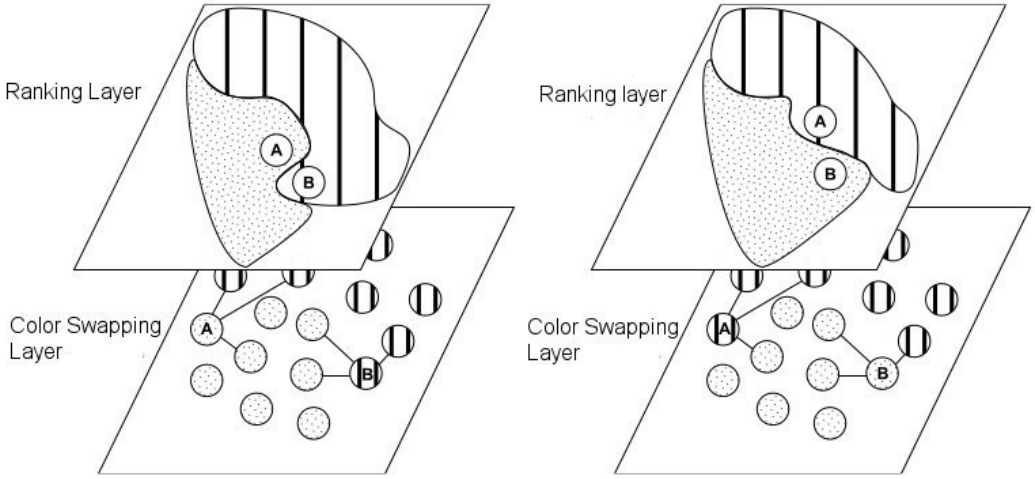


FIG. 4.3. *Example of an exchange colour using the ranking layer over the real JA-BE-JA coloured graph.*

TABLE 4.1
Edge-cut value for GP-TELOS *and* GP-SPARK *with the three datasets*

K	3elt		Vibrobox		Facebook	
	GP-TELOS	GP-SPARK	GP-TELOS	GP-SPARK	GP-TELOS	GP-SPARK
2	750	**1,433 (+91%)**	14,812	22,244 (+50%)	75,690	80,971 (+7%)
4	1,810	2,903 (+60%)	30,432	40,358 (+33%)	147,991	157,282 (+6%)
8	3,048	4,473 (+47%)	43,728	56,954 (+30%)	256,902	**245,682 (-4%)**
16	4,191	6,344 (+51%)	54,339	75,051 (+38%)	348,494	353,061 (+1%)
32	5,241	8,491 (+62%)	67,787	95,858 (+41%)	415,315	457,257 (+10%)
64	6,419	10,622 (+65%)	88,953	116,149 (+31%)	520,391	552,714 (+6%)

The *ranking function* between two nodes u and v is defined as follow:

$$H(z) = \left| \frac{|\{q|q \in neighbor(z) \wedge color(q) \neq color(z)\}|}{|neighbour(z)|} \right| \tag{4.1}$$

$$rankingFunction(u,v) = \begin{cases} +\infty, & \text{if } color(u) = color(v) \\ |H(u) - H(v)|, & \text{otherwise} \end{cases} \tag{4.2}$$

Note, two peers u and v are more similar as the $rankingFunction(u,v) \to 0$. The function ranks similar two peers u and v that have the same amount of neighbours of different colour and u and v are coloured differently. As a consequence, peers that are in the middle of a partition are similar to the ones that are in the middle of another partition (i.e. a node u has all the neighbours of its very same color, $H(u) = 1$) and, more useful for our purposes, vertices which are on the borders of a partition are similar to others that are also on the same kind of border. For instance, a blue vertex having 5 neighbours with 3 coloured red would be ranked high from a red vertex having 5 neighbours with 3 coloured blue. As a result, in a figurative way as shown in Figure 4.3, the stains flow between each other like a liquid. In this way, the ranking layer keeps in the neighbourhood of each vertex other vertices having high similarity according to *rankingFunction*. The aim is to retrieve a better solution starting from the actual topology to another more accurate for the purpose. Moreover, such an abstraction allows to find better candidates to exchange the colour with and to filter and organize the vertices among their *borderness* property.

We compared the performance of GP-TELOS and GP-SPARK by means of the following metrics: (i) *edge-cut*: the number of edges that cross the boundaries of each subgraph. This metrics gives an estimation about

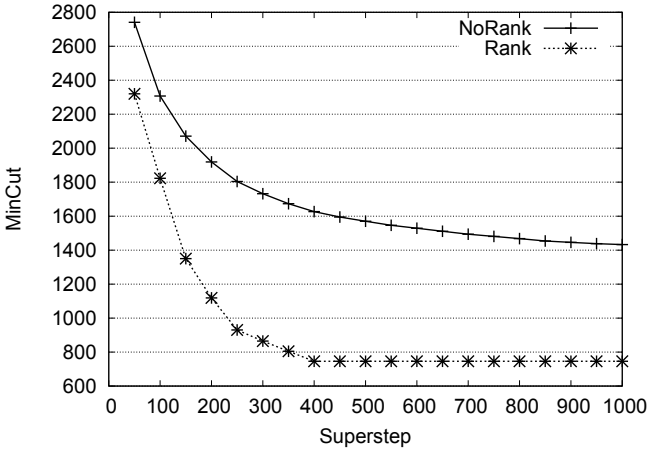


FIG. 4.4. *Convergence of edge-cut over supersteps*

the quality of the cut, with lower values corresponding to a better cut, and (ii) *convergence*: the number of supersteps required to achieve a substantially definitive edge-cut result. Our aim is to show that Telos does not affect the performance in term of supersteps to find a solution with respect to the GP-SPARK implementation.

The experiments have been conducted on two datasets taken from the Walshaw archive [47] (3elt and vibrobox) and from the Facebook social network[2]. Figure 4.4 shows the convergence time in term of supersteps for the 3elt dataset with $K = 2$. Results with other datasets and different values of K are not included due to space constraints but they exhibit similar trends. The results show evidence that convergence is similar between GP-SPARK and GP-TELOS, as in both cases they achieve an almost-definitive edge-cut around the 400th superstep. In particular, after the 400th superstep GP-TELOS is stable, whereas GP-SPARK is converging but it is improving the result in every step marginally. Also, GP-TELOS yields a much better quality of results, achieving half the edge-cut of GP-SPARK.

Table 4.1 presents the edge-cut obtained by GP-TELOS and GP-SPARK, averaging the results of 5 runs. We executed multiple runs by varying the number of the graph partitions with the values $K = \{2, 4, 8, 16, 32, 64\}$. It can be noticed that GP-TELOS obtains a better edge-cut in all the datasets and in all the configurations but with the Facebook dataset and 8 partitions. However also in this configuration GP-TELOS provides an edge-cut similar (just 4% less) to the one in the GP-SPARK version. Overall, the GP-TELOS version provides a value between the 47% and the 91% better in the 3elt dataset and always better that the 30% in the Vibrobox dataset. These results suggest that the new layer helps improving the results, and a layered vertex-centric approach can be used to carry out graph processing computations.

4.3. Scalability. The aim of this experiment is to give a preliminary evaluation of Telos framework in managing large input graphs varying the number of cores involved in the computation. To this end, we built two Erdos-Renyi random graphs (1M vertices 5M edges the first, 500K vertices 1.5M edges the second) generated with the Snap library [23]. It is worth to point out that the current implementation of Telos is prototypical, thus our interest is more related to the viability of the approach than to the pure performance. To give this evaluation, we run the torus overlay experiment described in the previous section on both graphs, measuring the convergence time when a different amount of computing cores is used. We performed our experiments using both these graphs, varying the number of cores in the following range: $\{8, 16, 24\}$. For the sake of the presentation, values are normalised, independently for each graph, in the range [1 ; 100], with 100 being the highest execution time. In addition, each value is the average of 3 independent runs. The results are presented in Figure 4.5. With the graph composed of 500K vertices, our approach achieves only a little scalability and no benefit is obtained when the total amount of computing elements is greater than 16. Instead, when considering the larger graph our approach is able to achieve a decent scalability using up to 24 cores.

[2] http://socialnetworks.mpi-sws.org/

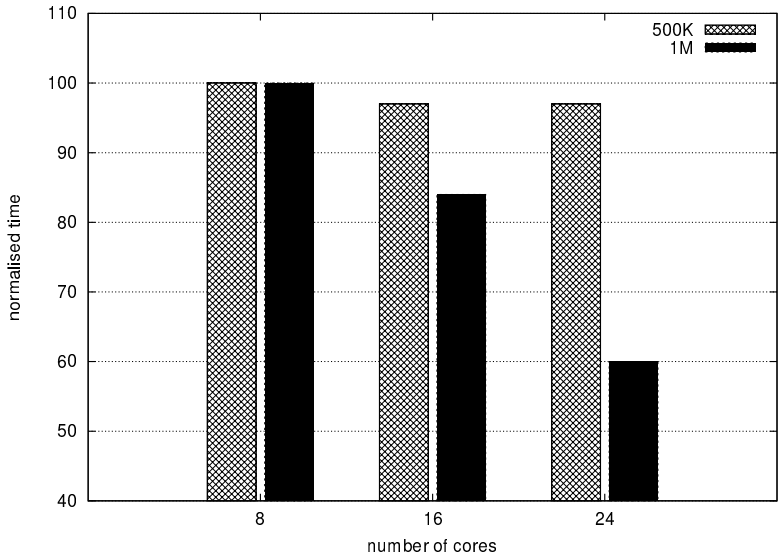


FIG. 4.5. *Scalability as a function of the number of cores*

5. Conclusion. This paper presents the approach adopted by Telos for programming of computations on large graphs. The aim of Telos is to propose an alternative to plain vertex-centric computations. The idea of Telos is based on defining overlay networks collectively built and managed by the nodes of the graph. Each overlay is specialised for a peculiar aim. The overlay-based approach takes inspiration from approaches that have proved to be robust and efficient in massively distributed systems that, somehow, recalls the structure of large graphs. We provided a detailed description of the concepts at the basis of Telos and its architecture, as well. We presented the Telos API, which has been designed to ease the programming effort required to program applications dealing with large graphs. In this way programmers can focus on the algorithm logic and to leave all the other burdens to the framework. We conducted an experimental analysis to validate the feasibility of the approach and showed its scalability with respect to the computational resources exploited. The experiments demonstrated that dynamic topologies can be effectively exploited during the computation. We believe that the ability of supporting multiple dynamic layers can be useful in many contexts, as for example in Peer-to-Peer applications [10, 39, 11] or for classical graph analysis problems, such as connected components [28, 25] and centrality measures [27]. As a future work we plan to conduct a comprehensive analysis of the performance of the framework and a comparison with other approaches targeting large graph computation. We also plan to implement Telos on top of different distributed frameworks (e.g., FastFlow [3], Akka [1]) to assess the performance of our proposed approach regardless the performances provided by Apache Spark.

REFERENCES

[1] Akka framework. `http://www.akka.io/`.
[2] M. ALDINUCCI, M. DANELUTTO, AND P. DAZZI. Muskel: an expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4), 2007.
[3] M. ALDINUCCI, M. DANELUTTO, P. KILPATRICK, AND M. TORQUATI. Fastflow: high-level and efficient streaming on multi-core.(a fastflow short tutorial). *Programming multi-core and many-core computing systems, parallel and distributed computing*, 2011.
[4] B. BACCI, M. DANELUTTO, S. ORLANDO, S. PELAGATTI, AND M. VANNESCHI. P3l: A structured high-level parallel language, and its structured support. *Concurrency: practice and experience*, 7(3):225–255, 1995.
[5] R. BARAGLIA, P. DAZZI, M. MORDACCHINI, L. RICCI, AND L. ALESSI. Group: A gossip based building community protocol. In *Smart Spaces and Next Generation Wired/Wireless Networking*, pages 496–507. Springer Berlin Heidelberg, 2011.
[6] M. BASTIAN, S. HEYMANN, M. JACOMY, ET AL. Gephi: an open source software for exploring and manipulating networks. *ICWSM*, 8:361–362, 2009.
[7] E. CARLINI, M. COPPOLA, P. DAZZI, D. LAFORENZA, S. MARTINELLI, AND L. RICCI. Service and resource discovery supports over p2p overlays. In *Ultra Modern Telecommunications & Workshops, 2009. ICUMT'09. International Conference on*,

pages 1–8. IEEE, 2009.
[8] E. Carlini, P. Dazzi, A. Esposito, A. Lulli, and L. Ricci. Balanced graph partitioning with apache spark. In *Euro-Par 2014: Parallel Processing Workshops*, pages 129–140. Springer, 2014.
[9] E. Carlini, P. Dazzi, A. Lulli, and L. Ricci. Distributed graph processing: an approach based on overlay composition. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1912–1917. ACM, 2016.
[10] E. Carlini, P. Dazzi, M. Mordacchini, and L. Ricci. Toward community-driven interest management for distributed virtual environment. In *Euro-Par 2013: Parallel Processing Workshops*, pages 363–373. Springer, 2014.
[11] E. Carlini, A. Lulli, and L. Ricci. dragon: Multidimensional range queries on distributed aggregation trees. *Future Generation Computer Systems*, 55:101–115, 2016.
[12] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan. One trillion edges: graph processing at facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.
[13] M. D. P. Dazzi. A java/jini framework supporting stream parallel computations. 2005.
[14] P. Dazzi, P. Felber, L. Leonini, M. Mordacchini, R. Perego, M. Rajman, and É. Rivière. Peer-to-peer clustering of web-browsing users. *Proc. LSDS-IR*, pages 71–78, 2009.
[15] P. Dazzi, M. Mordacchini, and F. Baglini. Experiences with complex user profiles for approximate p2p community matching. In *Computer and Information Technology (CIT), 2011 IEEE 11th International Conference on*, pages 53–58. IEEE, 2011.
[16] N. Doekemeijer and A. L. Varbanescu. A survey of parallel graph processing frameworks. *Delft University of Technology*, 2014.
[17] Z. J. Haas, J. Y. Halpern, and L. Li. Gossip-based ad hoc routing. *IEEE/ACM Trans. Netw.*, 14(3):479–491, June 2006.
[18] N. Javed and F. Loulergue. Osl: Optimized bulk synchronous parallel skeletons on distributed arrays. In Y. Dou, R. Gruber, and J. Joller, editors, *Advanced Parallel Processing Technologies*, volume 5737 of *Lecture Notes in Computer Science*, pages 436–451. Springer Berlin Heidelberg, 2009.
[19] D. Jeffrey and G. Sanjay. Mapreduce: Simplified data processing on large clusters. *Commununication ACM*, 1, 2008.
[20] M. Jelasity, A. Montresor, and O. Babaoglu. T-man: Gossip-based fast overlay topology construction. *Computer networks*, 53(13):2321–2339, 2009.
[21] V. Kalavri, V. Vlassov, and S. Haridi. High-level programming abstractions for distributed graph processing. *arXiv preprint arXiv:1607.02646*, 2016.
[22] H. Kavalionak and A. Montresor. P2p and cloud: a marriage of convenience for replica management. In *Self-Organizing Systems*, pages 60–71. Springer, 2012.
[23] J. Leskovec and R. Sosič. SNAP: A general purpose network analysis and graph mining library in C++. `http://snap.stanford.edu/snap`, June 2014.
[24] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, S. Lim, et al. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, 7(1-4):72–93, 2005.
[25] A. Lulli, E. Carlini, P. Dazzi, C. Lucchese, and L. Ricci. Fast connected components computation in large graphs by vertex pruning. *IEEE Transactions on Parallel & Distributed Systems*, to appear.
[26] A. Lulli, P. Dazzi, L. Ricci, and E. Carlini. A multi-layer framework for graph processing via overlay composition. In *European Conference on Parallel Processing*, pages 515–527. Springer, 2015.
[27] A. Lulli, L. Ricci, E. Carlini, and P. Dazzi. Distributed current flow betweenness centrality. In *2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems*, pages 71–80. IEEE, 2015.
[28] A. Lulli, L. Ricci, E. Carlini, P. Dazzi, and C. Lucchese. Cracker: Crumbling large graphs into connected components. In *2015 IEEE Symposium on Computers and Communication (ISCC)*, pages 574–581. IEEE, 2015.
[29] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.
[30] D. Margo and M. Seltzer. A scalable distributed graph partitioner. *Proceedings of the VLDB Endowment*, 8(12):1478–1489, 2015.
[31] C. Martella, D. Logothetis, A. Loukas, and G. Siganos. Spinner: Scalable graph partitioning in the cloud. *arXiv preprint arXiv:1404.3861*, 2014.
[32] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A library of constructive skeletons for sequential style of parallel programming. In *Proc. of the 1st International Conference on Scalable Information Systems*, InfoScale '06, New York, NY, USA, 2006. ACM.
[33] A. McAfee, E. Brynjolfsson, T. H. Davenport, D. Patil, and D. Barton. Big data. *The management revolution. Harvard Bus Rev*, 90(10):61–67, 2012.
[34] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. *ACM Computing Survey*, in press, 2014.
[35] H. Meyerhenke, P. Sanders, and C. Schulz. Parallel graph partitioning for complex networks. In *Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International*, pages 1055–1064. IEEE, 2015.
[36] M. Mordacchini, P. Dazzi, G. Tolomei, R. Baraglia, F. Silvestri, and S. Orlando. Challenges in designing an interest-based distributed aggregation of users in p2p systems. In *Ultra Modern Telecommunications & Workshops, 2009. ICUMT'09. International Conference on*, pages 1–8. IEEE, 2009.
[37] U. Müller-Funk, U. Thonemann, and G. Vossen. The münster skeleton library muesli–a comprehensive overview. 2009.
[38] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. The scala language specification, 2004.
[39] A. H. Payberah, H. Kavalionak, A. Montresor, J. Dowling, and S. Haridi. Lightweight gossip-based distribution

estimation. In *Communications (ICC), 2013 IEEE International Conference on*, pages 3439–3443. IEEE, 2013.

[40] F. Rahimian, A. H. Payberah, S. Girdzijauskas, M. Jelasity, and S. Haridi. Ja-be-ja: A distributed algorithm for balanced graph partitioning. In *Self-Adaptive and Self-Organizing Systems (SASO), 2013 IEEE 7th International Conference on*, pages 51–60. IEEE, 2013.

[41] S. Salihoglu and J. Widom. Optimizing graph algorithms on pregel-like systems. *PVLDB*, 7(7):577–588, 2014.

[42] Y. Simmhan, A. Kumbhare, C. Wickramaarachchi, S. Nagarkar, S. Ravi, C. Raghavendra, and V. Prasanna. Goffish: A sub-graph centric framework for large-scale graph analytics. In *European Conference on Parallel Processing*, pages 451–462. Springer, 2014.

[43] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 7(3):193–204, 2013.

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

[45] S. Voulgaris, D. Gavidia, and M. Van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.

[46] S. Voulgaris and M. van Steen. Vicinity: A pinch of randomness brings out the structure. In *Middleware 2013*, pages 21–40. Springer, 2013.

[47] C. Walshaw. The graph partitioning archive, 2002.

[48] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14):1981–1992, 2014.

[49] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[50] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

Edited by: Frédéric Loulergue
Received: September 17, 2016
Accepted: March 11, 2017

DOI 10.12694/scpe.v18i1.1233
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 18, Number 1, pp. 51–66. http://www.scpe.org

SEQUENCE SIMILARITY PARALLELIZATION OVER HETEROGENEOUS COMPUTER CLUSTERS USING DATA PARALLEL PROGRAMMING MODEL

MAJID HAJIBABA*, SAED GORGIN*, AND MOHSEN SHARIFI†

Abstract. Sequence similarity, as a special case of data intensive applications, is one of the neediest applications for parallelization. Clustered commodity computers as a cost-effective platform for distributed and parallel processing, can be leveraged to parallelize sequence similarity. However, manually designing and developing parallel programs on commodity computers is a time-consuming, complex and error-prone process. In this paper, we present a sequence similarity parallelization technique using the Apache Storm as a stream processing framework with a data parallel programming model. Storm automatically parallelizes computations via a special user-defined topology that is represented as a directed acyclic graph. The proposed technique collects streams of data from a disk and sends them sequence by sequence to clustered computers for parallel processing. We also present a dispatching policy for balancing the cluster workload and managing the cluster heterogeneity to achieve more than 99 percent parallelism. An alignment-free method, known as n-gram modeling, is used to calculate similarities between the sequences. To show the cost-performance superiority of our method on clustered commodity computers over serial processing in powerful computers, we simply use UniProtKB/SwissProt dataset for evaluation of the performance of sequence similarity as an interesting large-scale Bioinformatics application.

Key words: commodity cluster, parallelization, sequence similarity, Apache Storm

AMS subject classifications. 68W10, 68R10

1. Introduction.

As a result of increases in the size of available genome sequence data, the processing and storage of such data have become one of the major challenges in modern genomic research. Bioinformatics applications such as finding genes in DNA sequences, or aligning similar proteins, all need complex computations and vast amounts of processing power beside massive memory capacity. Therefore, there is high demand for faster algorithms and more powerful computers to reduce the execution time of these applications.

For Bioinformatics applications, the primary form of information are raw DNA and protein sequences. Therefore, one of the first steps toward obtaining information about a new biological sequence is to compare it with a set of known sequences in existing sequence databases [1]. The results of comparisons often suggest functional, structural, or evolutionary analogies between sequences. For a given sequence, searching in sequences of protein databases is one of the most fundamental and important tools for predicting the structural and functional properties of uncharacterized proteins.

The similarity between two sequences (gene or protein) indicates that they may be derived from the same ancestral sequence by an evolution. This evolutionary similarity is called homology and similar sequences are known as homologous [2].

The most common metric showing the similarity between two instances is their distance. Computing distances between sequences in large datasets is a computationally intensive task [1]. To handle the complexity of this task,we can take advantage of parallelization techniques. Pairwise calculation provides a natural target for parallelization because all elements of the distance matrix are independent [1].

Using network-attached workstations as a platform for distributed and parallel computing is becoming more and more popular. This is due to the fact that todays desktop machines are extremely powerful as well as affordable. In contrast, dedicated supercomputers are often very expensive, so most research groups have only limited access to such machines. Commodity computers may be slower than commercial supercomputers, but they are readily accessible and usable for many tasks. Small networks of commodity computers can be used for coarse-grain parallelization of database searches. As a result, in response to the rapid increase in the amount of data generated by the new technologies, such as next-generation sequencing (NGS), researchers can use parallelization strategies and distributed systems on commodity clusters to accelerate the search in such datasets. However, the users of Bioinformatics tools are typically unaware of the advantages of emerging technologies in distributed computing. This is due to the difficulty of converting sequential jobs to distributed

*Department of Electrical Engineering and Information Technology, Iranian Research Organization for Science and Technology, Tehran, Iran (hajibaba.m@irost.org, gorgin@irost.ir).

†School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran (msharifi@iust.ac.ir).

ones and the challenge of understanding distributed technologies well enough to create and manage a distributed environment [3]. Therefore, it can be useful to show how to parallelize Bioinformatics applications that can be followed by Bioinformaticians.

A practical study on using parallel computing to speed up the Bioinformatics problems has different aspects. A rich and complex study of parallel computing techniques and technologies for solving Bioinformatics problems has been explored in [1]. In this paper, we use sequence similarity as a case study of Bioinformatics problems for applying parallelization strategy. But, we focus on alignment-free sequence similarity methods. So, parallelization strategy for alignment-based methods such as BLAST [4, 5] is out of scope of this work. On the other hand, we try to find a software parallel solution to sequence comparison, and hardware parallel solutions are not in our scope. Various parallel architectures for sequence similarity search are compared in [6]. Moreover, some researchers in this scope have used parallel technologies such as GPU to speed up the sequence analysis algorithms [7, 8, 9]. In this paper, we concentrate on having applications that are run both in parallel and in a distributed manner and ignore solutions that only consider running applications in parallel like GPU solutions. We use commodity clustered computers to distribute sequence searches much the same as other researches that have parallelized sequence searches on workstation clusters, such as the work in [10] that uses an alignment-based method. There are also other works that emphasis on parallelization of sequence similarity search [11, 12, 13, 14], but few works use streaming to overcome big data challenges [15, 16, 17]. For example, [15] has proposed a stream-based approach for NGS read alignment based on the IBM InfoSphere Streams computing platform deployed on Amazon EC2.

We also use stream processing for a data and compute-intensive scientific problem (i.e. DNA sequence analysis). But, as a software solution, our strategy is based on a high-level parallel programming paradigm in an open source stream processing framework named Apache Storm. In order to get an efficient use of Storm on a heterogeneous commodity cluster, we have developed a scheduler to balance the workload on processing nodes. The used similarity method, in our work, is based on n-gram that is derived from statistical language modeling that uses the Markov chains representation together with cross-entropy from information theory to get a measure of similarity between n-grams [18]. We use stream processing to overcome computation challenges as well as memory challenges in this type of sequence similarity search method [19, 20].

More precisely, our contributions are as follows:

- We use protein-sequence streaming as a data partitioning strategy and utilize stream processing for coarse-grain search in a large database. Using protein-sequence streaming instead of database partitioning, we can reduce the cost of storage in some solutions such as in cloud computing.
- We build a task dispatching strategy (i.e., a weighted round-robin scheduler) to provide a balanced workload among heterogeneous clustered computers. In this strategy, we dispatch tasks with corresponding data to processing elements. So, each element takes proper sized subsets of the dataset, according to its computing power.
- We deploy Apache Storm for parallelization of a word-based sequence similarity search method in a semi-automatic programmer directed manner.

The rest of this paper is organized as follows. Section 2 outlines the scope of the work, the method used for sequence similarity and the parallel framework used in this paper. Section 3 describes the parallelization concerns of our work and used strategy to overcome these concerns in a real heterogeneous distributed environment. Section 4 presents how we have evaluated the performance and effectiveness of our proposed strategy. Section 5 concludes the paper.

2. Background.

2.1. Sequence comparison. Sequence comparison is used to detect the similarity of proteins for explaining phylogenetic relations between them. Existing sequence comparison methods can be classified into two main categories:

1. Alignment-based methods.
2. Alignment-free methods.

Alignment-based methods [21, 22] use dynamic programming to find the optimal alignment. These methods assign scores to different possible alignments and select the alignment with the highest score. The similarity score between two sequences in alignment-based methods is measured by an alignment algorithm (e.g., BLAST

[23] or FASTA [24]). However, the alignment-based methods have computational limitations on large biological databases and suffer from speed limitation [19, 25].

Alignment-free methods [26, 27] have been developed to alleviate the limitations of alignment-based methods [20]. The similarity score of two sequences in an alignment-free method is measured by various distance metrics, such as Euclidean distance or Kullback-Leibler divergence [28].

The alignment-free methods can be subdivided into three different classes: gene contents, data compression, and word composition [19]. The latter that has received considerable attention counts the number of words shared between a subject and a query sequence. Basically, in word-based methods, an optimal word length k is chosen, and then the similarity of any two sequences is assessed by comparing the frequencies of all subsequences of k adjacent letters (also called n-gram, k-mer, k-tuple or k-word) in sequences. High computing time and memory usage are disadvantages of all these three methods [19, 20]. Nonetheless, word-based methods have received credit in high-throughput classification of genome sequences and are deployed for diverse objectives in genomics such as sequence clustering and word similarity searches [29].

Some of the word-based methods [30, 31], determine the similarity between sequences by using a Markov model to estimate the probability that a sequence is relevant to a given query with respect to their word frequencies. In this paper, we deploy a similar method to calculate similarities between sequences, which is described in Section 2.2.

2.2. Similarity method. Amino acids consist of 20 different symbols that can be considered as the alphabet of a hypothetical language. Protein sequences can also be treated as texts written in this language. With this analogy, we can apply statistical language techniques in biological sequence analysis [18].

The unique order of amino acids in sequences affects the similarity of proteins. These orders can be obtained by n-gram based modeling techniques and used in similarity methods with cross-entropy related measures [18]. In these techniques, a Markov chain is built for n-gram models. For example, in Drosophila gene eyeless sequence [32], amino acids in the range of 30 to 40 are $EAVEASTASH$. Available tokens of this gene in 2-gram model are continuous base pairs with length 2, which are $\{EA, AV, VE, EA, AS, ST, TA, AS, SH\}$, while in the 3-gram model, are continuous base pairs with length 3, which are $\{EAV, AVE, VEA, EAS, AST, STA, TAS, ASH\}$.

The Shannon entropy [33] can be used to show how a sequence is fitted by an n-gram model, which is estimated by equation 2.1 [34].

$$H(x) = -\sum_{w^*} P(w_i^n)\ logP(w_{i+n-1}|w_i^{n-1}) \tag{2.1}$$

In equation 2.1, w_i^n corresponds to $\{w_i, w_{i+1}, \ldots, w_{i+n-1}\}$. The summation is over all feasible combinations of successive w with size n, i.e. $w = \{\{w_1, w_2, \ldots, w_n\}, \{w_2, w_3, \ldots, w_{n+1}\}, \ldots\}$. $P(w_i^n)$ is a joint probability that can be broken down using the Chain Rule to yield equation 2.2.

$$P(w_i^n) = P(w_i)\ P(w_{i+1}|w_i)\ P(w_{i+2}|w_i^2)\ \ldots\ P(w_{i+n-1}|w_i^{n-1}) = \prod_{k=i}^{i+n-1} P(w_k|w_i^{k-i}) \tag{2.2}$$

In equation 2.2, $P(w_{i+n-1}|w_i^{n-1})$ is the conditional probability of the nth word of a n-gram with respect to the previous $n-1$ words. Using the maximum likelihood estimation (MLE) [33], this term can be estimated by using the n-gram frequencies:

$$P(w_{i+n-1}|w_i^{n-1}) = \frac{C(w_i^{n-1}\ w_{i+n-1})}{C(w_i^{n-1})} = \frac{C(w_i^n)}{C(w_i^{n-1})} \tag{2.3}$$

In equation 2.3, $C(w_i^n)$ is the count of w_i^n. For example, in the aforementioned Drosophila gene eyeless sequence [32], the probability of bigram AS is:

$$P(A|S) = \frac{C(AS)}{C(A)} = \frac{2}{3} = 0.66$$

and its bigram model is illustrated in Figure 2.1, after calculating MLE for each token.

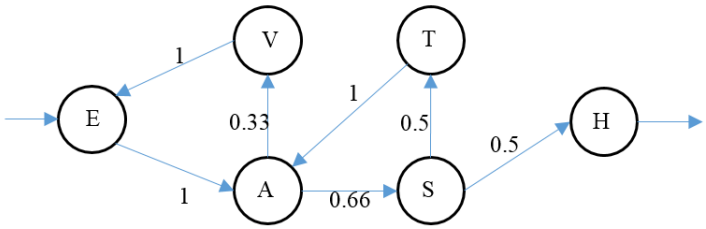


FIG. 2.1. *The bigram model of the sequence of EAVEASTASH*

If equation 2.1 is applied to two different sequences X and Y, the outcomes show which sequence is better fitted by the model, but cannot be used for their direct comparison. Thus, the similarity between two sequences must be represented as equation 2.4.

$$H(x,y) = -\sum_{w^*} P_X(w_i^n)\ logP_Y(w_{i+n-1}|w_i^{n-1}) \tag{2.4}$$

In equation 2.4, $P_X(w_i^n)$ relates to the subject sequence X and $P_Y(w_{i+n-1}|w_i^{n-1})$ relates to the query sequence Y whose model has to be estimated. Variable w_i^n runs over all the words (or n-grams) of the query protein sequence.

Now, if we let S_r be a subject sequence and $S = S_1, S_2, \ldots, S_N$ be the target protein database, then we can compute the similarity between them by using equation 2.4. The first step is the computation of reference score for the subject protein by calculating $H(S_r, S_r)$. Next, each sequence in the target database, $S_i (i = 1, \ldots, N)$ is given as the model sequence for calculating $H(S_r, S_i)$. After these calculations, we can have N dissimilarities that are the absolute differences against the reference score $D(S_r, S_i) = |H(S_r, S_i) - H(S_r, S_r)|$, for all $i = 1, \ldots, N$. Finally, by ordering these N dissimilarity measures, we can simply find the most similar sequences with the lowest distance to the subject sequence.

For protein coding genes, a tuple size of 3 can be a good choice [35] with less sensitivity and computation, but for more accuracy in this paper, we have chosen $n = 4$.

2.3. Apache Storm. Apache Storm is a real-time stream processing framework built by Twitter that is available as an Apache project [36]. After the batch processing became popular, it was realized that in many cases, the turnaround time was unacceptable. Given the needs for low-latency asynchronous stream processing solutions, Storm has tried to address these needs.

Storm is a scalable, fault-tolerant, distributed and real-time platform that provides distributed stream processing in a cluster. The Storm programming paradigm consists of Streams, Spouts, Bolts, and Topology. A Stream is a continuous unlimited sequence of data that can be processed in a parallel and distributed fashion. A Spout acts as a source that Streams come from and a Bolt takes the input Streams, processes them and produces new output Streams. The Spouts and Bolts are organized in a directed acyclic graph (DAG) called a Topology (Figure 2.2). An application developer that uses Storm can define an ideal Topology to describe communications between processing elements in his/her application and declare data dispatching mechanisms between processing elements. In Storm, data placement and distribution are provided by this user-defined Topology. Bolts and Spots in this topology are connected together with stream grouping. Stream grouping defines how Streams should be distributed among the Bolt's tasks. For example, in shuffle grouping, tuples are randomly spread over the Bolt's tasks such that each Bolt gets an equal number of tuples or in all grouping, the Stream is replicated across all tasks of the Bolt. Storm does not use intermediate queues to pass Streams between tasks. Instead, Streams are passed directly between tasks using batch messages at the network level to achieve a good balance between latency and throughput. Storm employs a master-worker computational model wherein the master is the main process that distributes tasks among the workers. Each worker process runs a Java Virtual Machine (JVM), on which it runs one or more executors. An executor is a thread in the system that is generated by a worker process and runs within the workers JVM. Executors are made of one or more tasks. The actual work for a Bolt or a Spout is done with the task. A worker process executes a subsection of a Topology on its own JVM.

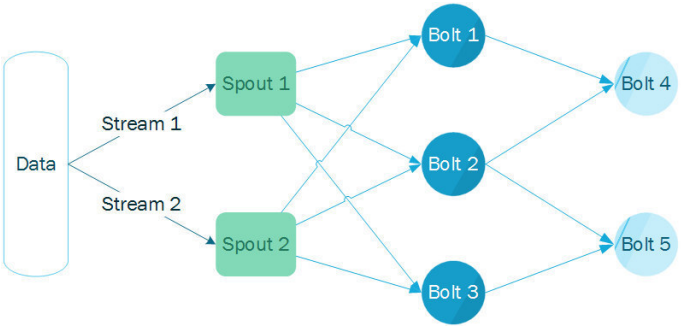


FIG. 2.2. *An example of Storm topology (components with the same color have the same work)*

```
Config conf = new Config();
conf.setNumWorkers(3); // use three worker instances

// define new topology
TopologyBuilder topologyBuilder = new TopologyBuilder();

// create Spout tasks using an object of user defined class of GreenSpout
// with two executors (threads) that should be assigned to execute this bolt
topologyBuilder.setSpout("green-spout", new GreenSpout(), 2);

// create Bolt tasks using an object user defined class of NavyBlueBolt
// with three tasks that should be assigned to execute this bolt in one executor (thread)
topologyBuilder.setBolt("navy-blue-bolt", new NavyBlueBolt(), 1).setNumTasks(3)
        .allGrouping("green-spout");

topologyBuilder.setBolt("blue-bolt", new BlueBolt(), 2).shuffleGrouping("navy-blue-bolt");

StormSubmitter.submitTopology("mytopology", conf, topologyBuilder.createTopology());
```

FIG. 2.3. *An example program in Storm*

The Storm resource manager divides nodes into slots and for each worker node, the user configures how many slots exist on that node. The Storm default scheduler uses a simple round-robin strategy [37]. When a job is submitted to the master, the scheduler counts all available slots on the worker nodes and assigns worker instances to nodes one by one in a round-robin manner. It deploys workers so that each node in a topology has almost an equal number of worker instances. The scheduler gets a job and decomposes it into tasks based on the topology, which can be delivered to slots.

Here are some reasons why Storm has been chosen as our choice for stream processing. Firstly, it is one of the leading open source stream processing tools that guarantees scalable, reliable and fault-tolerant processing of data. Secondly, compared with other leading open source tools such as Apache Spark, Storm processes the incoming streams in real time without any intentional latency. Thirdly, Storm has an easy programming paradigm (in contrast to map-reduce in Spark) and is usable with many programming languages (supports both JVM and non-JVM languages) without requiring any additional new concepts (such as Resilient Distributed Dataset (RDD) in Spark). Figure 2.3 shows a simple example Java code in Storm for the topology represented by Figure 2.2.

At last, but not least, Storm can be deployed on clustered commodity computers with low memory (1GB) and old architecture, while Spark needs high memory capacity for in-memory processing.

3. Distributed computation and parallelization. As mentioned in Section 1, Bioinformatics is confronted with increasingly large datasets. Processing of these datasets takes unacceptably long time on a single computer. Therefore, distributed computing on multiple computers is becoming an attractive approach to achieve shorter execution times [3]. In this section, we describe how to distribute sequence comparison jobs using a stream processing model on commodity computers to process large data in parallel.

Sequence comparison tasks can run on parallel processors using either coarse-grain or fine-grain methods [6]. In a coarse-grain method, sequences are partitioned equally among the processing elements that are appropriate for searches in a large database. So equal sized subsets of the database are assigned to processing elements, one subset to one processing element. Then, each element performs sequence analyses independently on its subset of the database. If there is a small number of sequences, coarse-grain parallelism may not be a good method. In these cases, a fine-grain method can be used to get a better speedup [6]. In our work, in order to evaluate the search in a huge database on a commodity cluster, we use coarse-grain parallelization to minimize the interactions between parallel processes.

Manually developing parallel programs is a time consuming, complex and error-prone process. Up to now, various tools have been available to assist programmers in converting serial programs into parallel ones. Designing and developing parallel programs by these tools requires the use of a parallel programming model such as a message passing or a data-parallel model. In distributed-memory parallel computation, a data-parallel model does not need to pass messages and thus makes the programming easier. A well-known representation of the data-parallel model is a Directed Acyclic Graph (DAG) in which nodes represent application tasks and directed arcs represent inter-task dependencies. One such tool that offers semi-automatic parallelization and follows the data-parallel model in a DAG is the Apache Storm that was described in Section 2.3.

The Storm programming model provides a distributed model for processing of streams on processing elements, wherein each element processes the input stream. Since batch processing is a specialization of the stream processing [38], we can use stream processing frameworks to do batch processing works. In this sense, we can aggregate the streams arriving in a time period and pass the aggregate to a processing element. So, Storm assumes the incoming streams as a batch and a set of batch tasks can be executed in parallel in a stream processing framework. In this manner, each task takes a chain of inputs, processes them using a user-defined function, and produces a result.

For sequence similarity, in this paper, a simple topology is used in Storm. In this topology, a Spout receives data and constructs a sequence from character streams to prepare the data elements in protein-sequence streaming. The Spout sends the result to main processing elements (i.e. Bolts) to compare them with the subject sequence. It sends sequences asynchronously to Bolts via a queue channel, so Bolts have their input data ready for execution. If the queue does not get empty during execution, with a sufficient number of Bolts, we can fully utilize the CPU cycles of the nodes. The subject sequence can be retrieved from a distributed coordination system such as Apache Zookeeper [39]. Sequences are sent from Spout to Bolts with a statically balanced Weighted Round Robin method that is described later (Algorithm 2). This architecture is illustrated in Figure 3.1.

For all schedulers in a distributed system, fairness is a critical feature to be considered. The round robin strategy as the default Storm scheduler in a shared heterogeneous cluster, which has multiple nodes with different hardware configurations, may not be fair. In default scheduling, although the processing is done in parallel, a job waits for its slowest worker to finish.

To balance the workload, we must partition the database into a number of portions according to the number and the power of processors allocated. To do this, our scheduling algorithm gets benchmarking data (as described in Algorithm 1) and then dispatches data elements according to their class weights to balance the workload of the cluster (as described in Algorithm 2).

By using the weights that are taken from benchmarks, the scheduling described in Algorithm 2 minimizes the imbalance ratio in comparison to the default scheduler; the imbalance ratio is the ratio of largest and smallest load on a processing instance, that we measure it in the next section.

In the next section, we compare the performances of the default scheduling and our proposed scheduling, as well as the cost-adjusted-performance of the clustering.

4. Experiments and results. There are various ways to measure the performance of parallel codes. To be rigorous, the speedup obtained by a parallel implementation should be compared with the best available sequential code for that problem. Also of importance is the concept of efficiency, defined as the ratio of speedup to the number of processors, and the computation to communication ratio. But the speedup is not quite calculable due to heterogeneity of clustered computers. Also, sequence-search algorithms can be measured along many dimensions such as execution time (or speed), millions of cell updates per second (MCUPS), deployment cost,

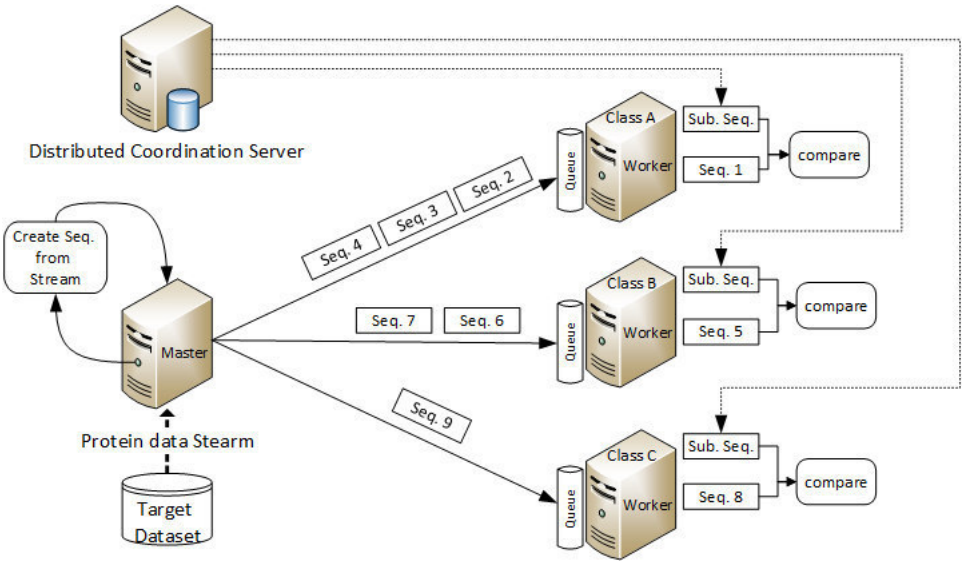


FIG. 3.1. *Sequence similarity distributed computing system overview*

Algorithm 1 Prepare information for choosing tasks

Input: $Benchmark$ is information receive from benchmarking phase
Output: $WorkersMeta$ is a List of workers' metadata

1: Info {
   id, ▷ id of a worker that is running on a node
   $weight$, ▷ class weight of the worker
   $counter$ ▷ the count of tasks must execute on the worker
   };
2: $cluster \leftarrow$ fetch running workers
3: **for each** $worker \in cluster$ **do**
4: $id \leftarrow$ identification number of the worker
5: $weight \leftarrow$ derive the weight of node from $Benchmark$ using GCD
6: $counter \leftarrow weight * factor$ ▷ $factor$ used to round the weight to a proper counter (normally is 10)
7: $tmpInf \leftarrow (Info)[Id, weight, counter]$
8: $WorkersMeta.add(tmpInf)$
9: **end for**
10: **return** $WorkersMeta$

and sensitivity [40]. In this section we measure the imbalance load ratio and the speedup of our solution. At last, we investigate the cost-adjusted-performance that is the product of execution time multiplied by the deployment cost.

For evaluation, the proposed method has been applied to publicly available protein database SWISS-PROT [41] as the target database to compare with the genome HIV-gp120 as a query sequence [42]. The input data (target and query) are all FASTA formatted sequence files. Characteristics of the target dataset and the query sequence are shown in Tables 4.1 and 4.2 respectively.

The studied similarity method has been benchmarked on several testbed systems in an effort to characterize their performance. We first present the performance on a commodity cluster with default Storm scheduler, demonstrating a poor performance caused by heterogeneity. Our comparative study of the performance of our proposed scheduling shows significant improvement in execution time.

Algorithm 2 Choose worker for processing the sequence

Input: $Task$ is a task for comparing two sequence,
$WorkersMeta$ is a List of workers' metadata
Output: $worker$ is the id of a worker to running the $task$

```
Let n be the size of WorkersMeta List
Let workerInf[1...n] be an Array of Info initiated from WorkersMeta
worker ← −1
Label : WorkerSelection
i ← 0
while i < n do
    counter ← workerInf[i].counter
    if counter > 0 then
        worker ← workerInf[i].id
        counter ← counter − 1
        break
    end if
    i ← i + 1
end while
if worker = −1 then
    for all worker ∈ workerInf do
        worker.counter ← worker.weight ∗ factor
    end for
    goto WorkerSelection
end if
return worker                    ▷Send Task to Worker
```

TABLE 4.1
SWISS-PROT dataset characteristics

Characteristic	value
Name	SWISS-PROT
Number of sequences	460,903
Shortest sequence	6
Longest sequence	35,213
Average nucleotides per sequence	374
Total number of nucleotides	172,370,171

Our test cluster had 10 compute nodes, each with 1GB memory and Intel processors with different architectures. Specifications of nodes deployed in the cluster, the execution time for each node to compare subject sequence with previously known target sequences in the dataset, and the number of each node are stated in Table 4.3. Compute nodes ran Apache Storm 0.9.3 under Ubuntu-12.04 to execute 1 or 2 workers per node according to the number of cores.

To evaluate our cluster, lets consider a cluster wherein we are given m workers for scheduling that are indexed by the set $W = w_1, .., w_m$. There are additional given n tasks for scheduling that are indexed by the set $T = t_1, .., t_n$. Let T_i be the set of tasks scheduled on w_i. Then, the load of the worker i (that is w_i) is defined by equation 4.1.

$$\ell_i = \sum_{t_i \in T} C_{i,j} \tag{4.1}$$

In equation 4.1, task t_j takes $C_(i, j)$ units of time if it is scheduled on w_i. The maximum load of scheduling

TABLE 4.2
SHIV-gp120 characteristics

Characteristic	value
Name	HIV1 HXB2 GP120
Length	481
Residues	31-511

TABLE 4.3
Specifications of the cluster nodes

CPU Model	# of Cores	Price[1]	Single Thread Execution Time	# of Nodes
Intel Pentium Processor E2180 (1M Cache, 2.00GHz, FSB 800MHz)	2	$0.5	1h 15m	1
Intel Pentium Processor E2160 (1M Cache, 1.80 GHz, FSB 800MHz)	2	$0.01	2h 10m	2
Intel Pentium 4 Processor HT (3.20GHz, 1M Cache, FSB 800MHz)	1	$2.2	1h 50m	3
Intel Pentium 4 Processor (2.40GHz, 512K Cache, FSB 533MHz)	1	$1.99	3h 30m	3
Intel Celeron D Processor 336 (256K Cache, 2.80GHz, FSB 533MHz)	1	$2.95	4h 20m	1

is called the makespan of the schedule and is given by equation 4.2.

$$\ell_{max} = \max_{i \in \{1,\ldots,m\}} \ell_i \tag{4.2}$$

In our test case, lets assume that T is the set of sequence comparison tasks that is submitted to the master. Each task is composed of two subtasks t_a and t_b. t_a is the task of constructing a sequence from Stream and t_b is the comparison of the sequence with the subject sequence. Since t_a is a lightweight task with respect to t_b, i.e. $t_b \gg t_a$, and t_a always runs on a fixed node (Spout node), we can ignore it from calculations. In addition, if we assume all sequences have the same length, so all tasks would be identical. In our dataset, the difference between sequence lengths is not significant, so for simplicity in calculations, we can assume that all tasks are identical. Sequence comparison tasks have also no data dependencies and are independent of each other in a non-preemptive manner.

Our cluster of computers was composed of 7 one core PCs and 3 double core PCs. To achieve a high level of parallelism, we used a fixed number of workers on each node according to its number of cores. Since we set cores per node as computing resources, we had 13 workers. One worker was used as Spout to collect and dispatch data sequentially. Therefore, twelve ($m = 12$) workers $W = w_1, w_2, \ldots, w_12$ were left for sequence comparison in parallel. Scheduling was carried out at worker level and each worker used First-In, First-Out (FIFO) method for performing the sequence comparison tasks.

By benchmarking, we estimated the time taken by each worker on a node to perform each task. The nodes in the cluster were classified by system benchmarking before submitting a topology. In the benchmarking phase, we ran several sequence comparisons on each node to get a task execution time. A task execution time was measured based on the number of sequences processed in a time period. The benchmarking results are stated in Table 4.4.

In this work, the workers are classified in p classes as $G = \{w_{g_1}, w_{g_2}, \ldots, w_{g_p}\}$ wherein g_i is an index for class i. So, the *makespan* will be $\ell_{max} = \max_{i \in \{1,\ldots,p\}} \ell_{g_i}$. Wherein $\ell_{g_i} = \sum_{t_j \in T_{g_i}} C_{g_i,j}$ denotes the load of

[1]These prices are taken from online marketplaces such as Amazon and eBay

TABLE 4.4
Execution time and performance on each type of node in the cluster testbed

Class (p)	**# of Slots**	**Job Completion Time**	**Sequences**	**Task execution time per worker (core)**	**Performance** ($\frac{sequence}{second}$)
1	2	75m = 4500s	460,903	0.0097	102
2	3	110m = 6600s	460,903	0.0143	70
3	4	130m = 7800s	460,903	0.0169	59
4	3	210m = 12600s	460,903	0.0273	36
5	1	260m = 15600s	460,903	0.0338	29

w_{g_i}, in which, task t_j takes $C_{g_i,j}$ units of time if scheduled on a worker with class i, and T_{g_i} denotes the set of tasks scheduled on w_{g_i}.

The Storm default scheduler uses a simple round-robin strategy that grants the same number of tasks ($\frac{sizeofT}{m}$) to each worker. In our test case, each task is a sequence comparison from the dataset with the subject sequence that is 460903 tasks of sequence comparison. Therefore, the *makespan* of the default scheduler at Storm is the following:

$$\ell_{max} = \sum_{t_j \in T_{g_5}} C_{g_5,j} = \frac{460903}{12} \times C_{g_5,j} = 38409 \times 0.0338 \approx 1,298$$

in which,

$$T_{g_5} = \{t_k, t_{k+12}, t_{k+(2\times12)}, \ldots, t_{k+(38408\times12)}\} \mid 1 \leq k \leq 12, C_{g_5,j} \approx 0.0338$$

To efficiently use a parallel computer system, a balanced workload among the processors is required. To compare the effectiveness of balancing the workload, the percentage of load imbalance (PLIB) [43] is defined by equation 4.3.

$$PLIB = \frac{\ell_{max} - \ell_{min}}{\ell_{max}} \times 100 \tag{4.3}$$

$PLIB$ is the percentage of the overall processing time that the first finished processor must wait for the last processor to finish. This number also indicates the degree of parallelism. For example, if $PLIB$ is less than one, we achieve over a 99 percent degree of parallelism. Therefore, a computational method with a lower $PLIB$ is more efficient than another one with a higher $PLIB$. The workload is perfectly balanced if $PLIB$ is equal to zero [43].

For default scheduler, $PLIB$ is obtained as follows:

$$\ell_{min} = \sum_{t_j \in T_{g_1}} C_{g_1,j} = \frac{460903}{12} \times C_{g_1,j} = 38409 \times 0.0097 \approx 373$$

$$PLIB = \frac{\ell_{max} - \ell_{min}}{\ell_{max}} \times 100 = \frac{1298 - 373}{1298} \times 100 \approx 71$$

Therefore, in this case, we have less than 29 percent parallelism. In this work, the goal is to minimize the *makespan*, i.e. the completion time of the last tasks in the schedule. For this goal, we have to achieve ideal workload balance, where $PLIB = 0$. In our proposed scheduling method, each worker gets tasks according to its computation power (or weight). This means that $\ell_1 \approx \ell_2 \approx \ldots \approx \ell_{12}$.

To obtain the weight of each worker from the benchmark we calculate the greatest common divisor (GCD) of task execution times on nodes and set weights accordingly. Assuming that the GCD of task execution times

is d, then we can set the weight of workers as in equation 4.4.

$$Weight_{g_i} = \frac{\sum_{k=1}^{p} \frac{\text{execution time of } w_{g_k}}{d}}{\frac{\text{execution time of } w_{g_i}}{d}} \tag{4.4}$$

In our example $d = 13$ and we have

$$\sum_{k=1}^{5} \frac{\text{execution time of } w_{g_k}}{d} = \frac{\text{execution time of } w_{g_1}}{d} + \frac{\text{execution time of } w_{g_2}}{d} + \ldots + \frac{\text{execution time of } w_{g_5}}{d} = 7.5 + 11 + 13 + 21 + 26 = 78.5$$

So,

$$Weight_{g_1} = 10.4, Weight_{g_2} = 7.1, Weight_{g_3} = 6, Weight_{g_4} = 3.7, Weight_{g_5} = 3$$

In our test case, Storm selects one slot of a node in class 3 as master process. So we have

$$T_{g_i} = \frac{460903}{10.4 \times 2 + 7.1 \times 3 + 6 \times 3 + 3.7 \times 3 + 3} = \frac{460903}{74.2}$$

$$\ell_{g_1} = \sum_{t_j \in T_{g_1}} C_{g_1,j} = 6212 \times 10.4 \times 0.0097 \approx 627$$

$$\ell_{g_2} = \sum_{t_j \in T_{g_2}} C_{g_2,j} = 6212 \times 7.1 \times 0.0143 \approx 631$$

$$\ell_{g_3} = \sum_{t_j \in T_{g_3}} C_{g_3,j} = 6212 \times 6 \times 0.0169 \approx 630$$

$$\ell_{g_4} = \sum_{t_j \in T_{g_4}} C_{g_4,j} = 6212 \times 3.7 \times 0.0273 \approx 627$$

$$\ell_{g_5} = \sum_{t_j \in T_{g_5}} C_{g_5,j} = 6212 \times 3 \times 0.0338 \approx 630$$

$$\ell_{max} = \max \ell_{g_i} \approx 631$$

Thus, we expect that our scheduler should perform twice better than the default scheduler (based on their ℓ_{max}, i.e. 631 compared to 1298), which is also shown by our experimental results in Figure 4.1.

Figure 4.1 shows the turnaround times under the default scheduler and our proposed scheduling method according to the class of nodes in the cluster; the turnaround time includes time to load workers, submission time of tasks to workers, scheduling time, synchronization time, inter-node communication time, and delays in acknowledging.

Experimental results in Figure 4.1 show that our proposed method produces a smaller $PLIB$ than the default scheduler.

$$PLIB = \frac{\ell_{max} - \ell_{min}}{\ell_{max}} \times 100 = \frac{631 - 627}{631} \times 100 \approx 0.6$$

Our scheduling method also achieved more than 99 percent degree of parallelism.

Program speedup is another important measure of the performance of a parallel program. By conventional definition of the speedup, if machine M_1 can solve problem P in time T_1 and machine M_2 can solve the same problem in time $T2$, the speedup of machine M_1 over machine M_2 on problem P is represented by equation 4.5.

$$S_P = \frac{T_1}{T_2} \tag{4.5}$$

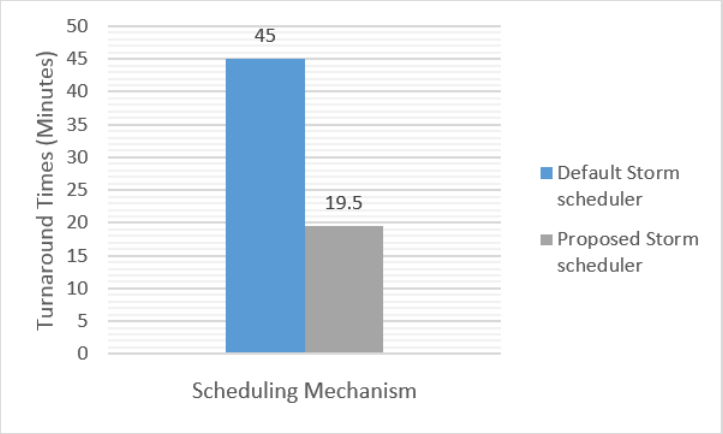


FIG. 4.1. *Comparison of turnaround times for each class of physical nodes and the whole cluster*

If machine M_2 is composed of m identical instances of the base processor of M_1, then we would expect m processors to solve the solution m times faster than a single processor. But this expectation is used in homogeneous clusters. For simple calculation of speedup in heterogeneous clusters, based on [44], if our cluster is a set $H = \{M_1, M_2, \ldots, M_m\}$ of m distinct machines, the speedup of a heterogeneous cluster C on problem P is defined by equation 4.6.

$$S_P = \frac{\min\{T_P^{M_1}, T_P^{M_2}, \ldots, T_P^{M_m}\}}{T_P^C} \tag{4.6}$$

In equation 4.6, $T_P^{M_k}$ is the execution time on machine M_k to solve problem P. In other words, the fastest machine is selected for comparison to the cluster. For our cluster with $m = 13$ based on equation 4.6 the speedup is:

$$S_P = \frac{\min\{4500, 6600, 7800, 12600, 15600\}}{1170} = \frac{4500}{1170} = 3.8$$

But this is not a good measure of speedup for our heterogeneous system due to the difference in nodes performance (slowest machine is 3.5 times slower than the fastest machine). An extension of the relative speedup for heterogeneous systems has been presented in [45], which is proved to be an appropriate tool for the evaluation of the speedup of parallel discrete event simulation in heterogeneous execution environments. So, accordingly, the relative speedup in this paper is calculated by equation 4.7.

$$r_P = \frac{N_e}{T_P^C \times P_c} \tag{4.7}$$

In equation 4.7, N_e is the number of events in the system. The cumulative performance of the heterogeneous system can this be defined by equation 4.8.

$$P_c = \sum_{i=1}^{NT} P_i \times N_i \tag{4.8}$$

Let us denote the number of CPU types in a heterogeneous system by NT, the number and the performance of CPU cores available from type i by N_i and P_i, respectively. Since our problem is near to discrete event simulation, the relative speedup or efficiency of our cluster will be:

$$P_c = 2 \times 102 + 3 \times 70 + 4 \times 59 + 3 \times 36 + 1 \times 29 = 787$$

TABLE 4.5
Cost and performance of the proposed method with other systems

System	# of Cores	Cost	Performance (turnaround time)
Intel Core i7-3770 Processor (8M Cache, up to 3.90GHz)	4	$344.99	26.5m
Intel Pentium G620 (3M Cache, 2.60GHz)	2	$78.12	40m
INTEL Xeon E5430 (12M Cache, 2.66GHz)	4	$40.0	48m
Proposed technique over given cluster	13	$16.04	19.5m

$$r_P = \frac{460903}{787 \times 1170} \approx \%50$$

This value is not bad at all for such heterogeneous clusters [45] with low network bandwidths, especially when the speedup is limited by the serial section (i.e. Spout) of the program.

Scalability becomes the basic need for distributed systems. Because of some problems such as heterogeneity of our cluster and lack of more nodes, we cannot practically test the scalability of our solution. However, we try to measure the scalability of our solutions compared to other similar works that scale well.

A similar work to ours is a streaming approach in a distributed architecture for scaling up natural language processing methods [46] which also leverages Storm. This work describes a series of experiments carried out with the goal of analyzing the scaling capabilities of the language processing. It shows that the use of Storm is effective and efficient for scalable distributed language processing across multiple machines when processing texts on a medium and large scale. Likewise, as mentioned earlier, the used similarity method in this work is a form of natural language processing [31] and implemented using Storm in a heterogeneous distributed environment. So, we can conclude our solution is scalable in nature and is efficient in the same environment.

Storm was originally aimed for processing twitter streams at scale [36]. The authors in [47] propose an event detection approach using Apache Storm to detect events within social streams like Twitter that can scale to thousands of posts every second. It is really a big challenge to analyze the bulk amount of tweets to get relevant and different patterns of information on a timely manner. We observed that the sequence similarity algorithm is an embarrassingly parallel problem that is highly parallel and can be deployed on a distributed system. We can consider sequences as events and similarity of a sequence with the reference as a rule to detect events. It has one Spout that reads events from document (dataset for our case) and a pipeline topology like ours. So, through experimentation on a large Twitter dataset by mentioned paper, we can conclude that our approach can scale to the equivalent size of data.

As a last example, the authors in [48], have addressed the problem of data stream clustering by developing scalable distributed algorithms that run on the Apache Storm. They demonstrate experimentally that they are able to gain close to linear scalability up to the number of physical machines. Likewise, our used similarity method was applied to the clustering sequences, based on Markov chains representation known as n-gram in statistical language modeling [18]. We used this sequence clustering strategy in a stream processing manner on Storm. Therefore, with an optimistic attitude, we can conclude that our solution is scalable at the same level as this work. However, there are some other works that can be used to inference the scalability of our solution [49, 50, 51].

In order to evaluate the performance-per-cost, we just use CPU costs as an approximation of system cost. In our test case, the relation between computer system prices and CPU prices is perfect. The modern and more expensive CPUs need modern and more expensive motherboards, RAMs and also need more power consumption. So we ignore these costs and simply compare CPU costs. The cost of the total cluster and other systems based on the lowest prices (obtained from retail marketplaces) are shown in Table 4.5.

The "cost-adjusted-performance" of the mentioned cluster running Storm with proposed methods, using the multiplication of the runtime and the cost of the system, is compared with other systems in Figure 4.2. The

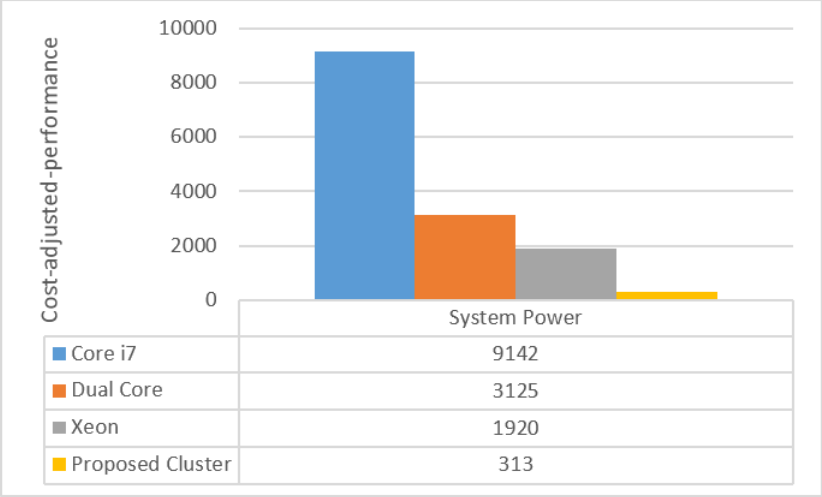


FIG. 4.2. *Comparison of Cost-adjusted-performance of our testbed with proposed method and other types of testbeds*

results show that a commodity cluster with our proposed scheduling method has a better performance-per-cost than other systems.

Although the proposed cluster has more cores, but it demonstrates the lowest cost-adjusted-performance (or highest performance-per-cost) compared to other single systems. The relative importance of equipment cost makes traditional server solutions less appealing for solving our problem because they increase performance, but decrease the performance-per-cost.

5. Conclusions and future works. Clustered commodity computers have already experienced enormous attention and used as a cost-effective platform for distributed and parallel processing of data/compute-intensive applications, compared to more expensive dedicated supercomputers. In Bioinformatics, searching in a database is the most widely used technique and is one of the most common targets to use parallelization in Bioinformatics. In this work, for parallelize sequence similarity methods, we use Storm as a stream processing framework. The Storm framework automatically provides some specifications for parallel computing, such as the scheduling, synchronization, and inter-machine communication. But other specifications, especially partitioning the input data and balancing the workload should be explicitly programmed using a given strategy such as the strategy we have proposed in this paper. Streaming can be a good choice instead of database partitioning when the cost of storage and/or bandwidth is important.

In a perfect world of parallelization, for n processors, the runtime should be n times faster than doing the same work on one processor. However, in a real world, the performance of a parallelization is decreased due to the interaction between processes and synchronization. Moreover, the heterogeneity affects the workload balancing; dividing the jobs equally across machines in a heterogeneous cluster leads to load imbalance. So, efficient scheduling of applications based on the characteristics of computing nodes can yield significant speedups. In this paper, we tried to alleviate these obstacles by running coarse-grain CPU-intensive tasks on loosely-coupled workers with the same workload according to their power.

In the first step to cut down parallelization costs, we diminish the process interactions via reducing the number of tasks. In this work, we try to minimize process interaction by creating coarsely-grained tasks. In the second step after minimizing interactions with proper granularity, communications will be addressed. In a distributed memory fashion, inter-process communications is performed by sending messages over the network. This message-passing paradigm forces programmers to deal with issues such as the location of the data, the method of the communication, and the communication latencies. However, by using a data-parallel model as an explicit programming paradigm for distributed-memory parallel computation that is followed by Storm, the programmer is free from details of message passing. On the other hand, in heterogeneous environments, the speedup suffers from unpredictable behavior that is due to diversities in the performance of CPUs. Getting

a good performance in a heterogeneous distributed system by distributing tasks and dispatching data in such manner that heterogeneity does not affect the speedup is a challenging problem. In this paper, we use an approach that distributes tasks and balances the workload among nodes and provides twofold speedup on a heterogeneous distributed environment.

It should be noted that parallelization strategies presented here would also benefit other commonly used Bioinformatics tools. The alignment-based methods such as BLAST can also be parallelized using our proposed method by streaming the query sequences. Consequently, other algorithms in computational biology may take advantage of performance improvement by this method on cluster based implementations. By running parallel applications on large commodity clusters with tens of nodes, researchers can cheaply perform analyses with acceptable execution time. We are currently studying the potential parallelism of the algorithm, especially in balancing the workload and also the programming paradigms of parallel systems such as MapReduce. In our future work we are going to consider sequence lengths in calculating loads on each node and use a dynamic workload balancer to avoid benchmarking.

REFERENCES

[1] A. Y. Zomaya, *Parallel Computing for Bioinformatics and Computational Biology*, WileySeries on Parallel and Distributed Computing, Wiley-Interscience, 2005.

[2] G. A. Petsko and D. Ringe, *From Sequence to Function: Case Studies in Structural and Functional Genomics*, Primers in biology, New Science Press Sunderland, MA,London, 2004.

[3] A. Matsunaga, M. Tsugawa, and J. Fortes, *Cloudblast: Combining mapreduce and vir-tualization on distributed resources for bioinformatics applications*, in 2008 IEEE FourthInternational Conference on eScience, Dec 2008, pp. 222-229.

[4] A. E. Darling, L. Carey, and W. Feng, *The Design, Implementation, and Evaluation of mpiBLAST*, in ClusterWorld Conference, San Jose, California, June 2003.

[5] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova, *Efficient data access for parallel blast*, in 19th IEEE International Parallel and Distributed Processing Symposium, April 2005, pp. 72b72b.

[6] R. Hughey, *Parallel hardware for sequence comparison and alignment*, CABIOS, 12(1996), pp. 473-479.

[7] E. F. d. O. Sandes, G. Miranda, A. C. M. A. d. Melo, X. Martorell, and E. Ayguad, *Cudalign 3.0: Parallel biological sequence comparison in large gpu clusters*, in 2014 14thIEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, May 2014,pp. 160-169.

[8] M. C. Schatz, C. Trapnell, A. L. Delcher, and A. Varshney, *High-throughput sequence alignment using graphics processing units*, BMC Bioinformatics, 8(2007), p. 474.

[9] H. Jiang and N. Ganesan, *CUDAMPF: a multi-tiered parallel framework for ac-celerating protein sequence search in HMMER on cuda-enabled GPU*, BMC Bioinformatics,17 (2016), p. 106.

[10] R. C. Braun, K. T. Pedretti, T. L. Casavant, T. E. Scheetz, C. L. Birkett, and C. A.Roberts, *Parallelization of local blast service on workstation clusters*, Future Gener. Comput. Syst., 17 (2001), pp. 745-754.

[11] A. S. Deshpande, D. S. Richards, and W. R. Pearson, *A platform for biological sequence comparison on parallel computers*, Computer Applications in the Biosciences, 7(1991), pp. 237-247.

[12] T. Rognes, *Paralign: a parallel sequence alignment algorithm for rapid and sensitive database searches*, Nucleic Acids Research, 29 (2001), p. 1647.

[13] X. l. Yang, Y. l. Liu, C. f. Yuan, and Y. h. Huang, *Parallelization of blast with mapreduce for long sequence alignment*, in 2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming, Dec 2011, pp. 241–246.

[14] M. C. Schatz, *Cloudburst: highly sensitive read mapping with mapreduce*, Bioinformatics, 25 (2009), p. 1363.

[15] R. Kienzler, R. Bruggmann, A. Ranganathan, and N. Tatbul, *Large-Scale DNA Sequence Analysis in the Cloud: A Stream-Based Approach*, Springer Berlin, Berlin, Heidelberg, 2012, pp. 467–476.

[16] B. Wolf, P. Kuonen, and T. Dandekar, *Multilevel parallelism in sequence alignment using a streaming approach*, Nesus 2015 workshop, (2015).

[17] S. A. Issa, R. Kienzler, M. El-Kalioby, P. J. Tonellato, D. Wall, R. Bruggmann, and M. Abouelhoda, *Streaming support for data intensive cloud-based sequence analysis*, BioMed Research International, (2013), p. 16.

[18] A. B. Marta and N. Robu, *A study of sequence clustering on proteins primary structure usinga statistical method*, Acta Polytechnica Hungarica, 3 (2006), pp. 1727.

[19] G. Lu, S. Zhang, and X. Fang, *An improved string composition method for sequence comparison*, BMC Bioinformatics, 9 (2008), p. S15.

[20] C. Yu, R. L. He, and S. S.-T. Yau, *Protein sequence comparison basedon k-string dictionary*, Gene, 529 (2013), pp. 250–256.

[21] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, *Gapped blast and psi-blast: a new generation of protein database search programs*, Nucleic Acids Res, 25 (1997), pp. 3389–3402.

[22] D. H. Huson and C. Xie, *A poor mans blastx high-throughput metagenomic protein database search using pauda*, Bioinformatics, 30 (2013), p. 38.

[23] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, *Basic local alignment search tool*, Journal of Molecular Biology, 215 (1990), pp. 403–410.

[24] W. R. Pearson and D. J. Lipman, *Improved tools for biological sequence comparison*, in Proceedings of the National Academy of Sciences of the United States of America, 85 (1988), pp. 2444–2448.
[25] C. Yu, M. Deng, and S. S.-T. Yau, *DNA sequence comparison by a novel probabilistic method*, Information Sciences, 181 (2011), pp. 1484–1492.
[26] K. Song, J. Ren, Z. Zhai, X. Liu, M. Deng, and F. Sun, *Alignment-free sequence comparison based on next-generation sequencing reads*, Journal of Computational Biology, 20 (2013),pp. 64–79.
[27] C. A. Leimeister, M. Boden, S. Horwege, S. Lindner, and B. Morgenstern, *Fast alignment-free sequence comparison using spaced-word frequencies*, Bioinformatics, 30 (2014), pp. 1991–1999.
[28] M. N. Davies, A. D. Secker, A. A. Freitas, J. Timmis, E. Clark, and D. R. Flower, *Alignment-independent techniques for protein classification*, Current Proteomics, 5 (2008), pp. 217–223.
[29] T. Z. DeSantis, K. Keller, U. Karaoz, A. V. Alekseyenko, N. N. Singh, E. L. Brodie, Z. Pei, G. L. Andersen, and N. Larsen, *Simrank: Rapid and sensitive general-purpose k-mer search tool*, BMC Ecology, 11 (2011),p. 11.
[30] L. Fu, B. Niu, Z. Zhu, S. Wu, and W. Li, *Cd-hit: accelerated for clustering the next-generation sequencing data*, Bioinformatics, 28 (2012), p. 3150.
[31] A. B. Marta, I. Pitas, and K. Lyroudia, *Statistical Method of Context Evaluation for Biological Sequence Similarity*, Springer US, Boston, MA, 2006, pp. 99–108.
[32] *UniProtKB, Eyeless protein*, www.uniprot.org/uniprot/O96791.
[33] C. D. Manning and H. Schutze, *Foundations of Statistical Natural Language Processing*, MIT Press, Cambridge, MA, USA, 1999.
[34] D. H. V. Uytsely and D. V. Compernolle, *Entropy-based context selection in variable-length n-gram language models*, in IEEE BENELUX Signal Processing Symposium, 1988.
[35] K. Yang and L. Zhang, *Performance comparison of gene family clustering methods with expert curated gene family data set in Arabidopsis thaliana*, Planta, 228 (2008), pp. 439–447
[36] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, *Storm@twitter*, in Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD14, New York, NY, USA, 2014, ACM, pp. 147–156.
[37] M. Rychly, P. koda, and P. mr, *Scheduling decisions in stream processing on heterogeneous clusters*, in 2014 Eighth International Conference on Complex, Intelligent and Software Intensive Systems, July 2014, pp. 614–619.
[38] J. Kreps, *I Heart Logs: Event Data, Stream Processing, and Data Integration*, OReilly Media,2014.
[39] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, *Zookeeper: Wait-free coordination for internet-scale systems*, in Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference, USENIXATC10, Berkeley, CA, USA, 2010, USENIX Association, pp. 11–11.
[40] A. M. Aji and W. Feng, *Optimizing performance, cost, and sensitivity in pairwise sequence search on a cluster of PlayStations*, in 8th IEEE International Conference on BioInformatics and BioEngineering, BIBE, Athens, 2008.
[41] E. Boutet, D. Lieberherr, M. Tognolli, M. Schneider and A. Bairoch, *UniProtKB/Swiss-Prot*, Methods in Molecular Biology, 406 (2007), pp. 89–112.
[42] *gp120 - ENV Glycoprotein 120*, http://www.bioafrica.net/proteomics/ENV-GP120prot.html.
[43] T. K. Yap, O. Frieder and R. L. Martino *Parallel Computation in Biological Sequence Analysis*, IEEE Transactions on Parallel and Distributed Systems, 9(1998), pp. 283–294.
[44] V. Donaldson, F. Berman and R. Paturi, *Program Speedup in a Heterogeneous Computing Network*, Journal of Parallel and Distributed Computing, 21(1994), pp. 316–322.
[45] G. Lencse and I. Derka, *Testing the Speedup of Parallel Discrete Event Simulation in Heterogeneous Execution Environments*, in Proc. ISC'2013, 11th Annu. Industrial Simulation Conf., Ghent, Belgium, 2013.
[46] R. Agerri, X. Artola, Z. B. G. Rigau and A. Soroa, *Big data for Natural Language Processing: A streaming approach*, Knowledge-Based Systems, 79(2015), pp. 36–42.
[47] R. McCreadie, C. Macdonald, I. Ounis, M. Osborne and S. Petrovic, *Scalable Distributed Event Detection for Twitter*, in IEEE International Conference on Big Data, Silicon Valley, CA, USA, 2013.
[48] P. Karunaratne, S. Karunasekera and A. Harwood, *Distributed stream clustering using micro-clusters on Apache Storm*, Journal of Parallel and Distributed Computing, 2016.
[49] X. Gao, E. Ferrara and J. Qiu, *Parallel clustering of high-dimensional social media data streams*, 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, 2015.
[50] W. Inoubli, S. Aridhi, H. Mezni and A. Jung, *Big Data Frameworks: A Comparative Study*, CoRR, abs/1610.09962(2016).
[51] M. A. Lopez, A. Lobato and O. C. M. B. Duarte, *A performance comparison of Open-Source stream processing platforms*, in IEEE Global Communications Conference (Globecom), Washington, USA, 2016.

Edited by: Frédéric Loulergue
Received: August 29, 2016
Accepted: March 6, 2017

DOI 10.12694/scpe.v18i1.1234
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 18, Number 1, pp. 67–89. http://www.scpe.org

PVL: PARALLELIZATION AND VECTORIZATION OF AFFINE PERFECTLY NESTED-LOOPS CONSIDERING DATA LOCALITY ON SHORT-VECTOR MULTICORE PROCESSORS USING INTRINSIC VECTORIZATION

YOUSEF SEYFARI, SHAHRIAR LOTFI, AND JABER KARIMPOUR*

Abstract. There is an urgent need for high-performance computations. Cores and Single Instruction Multiple Data (SIMD) units are important resources of modern architectures to speed up the execution of programs. Also, the importance of the data locality cannot be neglected in computations. Using cores, SIMD units, and data locality simultaneously is critical to gain peak performance of the architecture. But, there are a few research efforts trying to consider these three resources at the same time. There is a challenge in choosing loops which could be whether run on SIMD units or cores for vectorization and parallelization, respectively. This paper proposes an approach, named PVL, for parallelization and vectorization of nested loops considering data locality based on the polyhedral model on short-vector multicore processors. More precisely, PVL, tries to satisfy dependences in the middle levels of nested loop (the levels between outermost and innermost levels) while trying to move dependence-free loops to the outermost and innermost position in order to parallelize and vectorize them, respectively. The experimental results show that the proposed approach, PVL, is significantly effective compared to the other approaches.

Key words: short-vector multicore processors, parallelization, vectorization, perfectly nested-loops

AMS subject classifications. 65Y05

1. Introduction.

1.1. Motivation. Since the computers have become publicly available, the demand for more increasingly speed has not vanished. However, it becomes more difficult to speed up the processors by increasing the frequency. Two major problems with speeding up the processors are overheating and power consumption [1, 2]. One of the proposed solutions to these barriers is multicore architecture. Nowadays, there is a growing trend in processor industry towards multicore processors. This trend has involved two different areas: (1) ordinary users and (2) supercomputers. Presently, typical users can get better response time in CPU-intensive applications due to improved multitasking with the help of multicore processors. The appearance of multicore architecture also has brought the supercomputing area into a new era. In fact, multicore processors have already been widely used in parallel computing. Lei et al. [2], in 2007 reported that more than 20% of supercomputers processors are multicore processors, but in the last report of Top500 supercomputer list published in November 2016, there is no supercomputer with single core processor. In other words, 100% of processors of supercomputers are multicore processors [3]. These facts accentuate the importance of effective use of multiple cores.

Besides executing different instructions on multiple processing units simultaneously, namely parallelism, an alternative approach has been proposed which employs certain instructions called vector instructions [4]. Vector instructions would initiate an element-wise operation on two vector quantities in vector registers which could be loaded from memory in a single operation; these machines are called vector machines [5]. Since the late 90's, specialized processing units called Single Instruction Multiple Data (SIMD) extensions have been included in the instruction set architecture (ISA) of processors to exploit a similar kind of data parallelism as vector machines [6]. However, since the width of these vector instructions is relatively small, they are also called short-vectors [7]. But due to the growing demand for speed, the width of vector-SIMD instruction sets is constantly increasing (for example, 128 bits in SSE [8], 256 bits in AVX [9] and 512 bits in LRBni [10]). According to Cebrian et al. [11], in addition to the performance, vectorizing using short-vectors has also a great impact on energy efficiency. These highlights the significance of the effective use of SIMD vectors. The problem gets more complicated when both parallelization and vectorization are considered together.

1.2. The problem. Today, computations from complex scientific computations such as signal and image processing demand high-performance computing to be solved faster. These problems are characterized by long running computations with spending most of their running time in nested loops. Due to the long consumed

*Department of Computer Science, Faculty of Mathematical Sciences, University of Tabriz, Tabriz, Iran (seyfari@tabrizu.ac.ir, shahriar_lotfi@tabrizu.ac.ir, karimpour@tabrizu.ac.ir).

time in the loops, they are privileged candidates for parallel execution on both kinds of mentioned processing resources: (1) multiple cores and (2) vector-SIMD. In order to fulfill the computing needs of scientific computations, it is essential to use both multiple cores and vector-SIMD effectively.

Listing 1
The general form of the perfectly nested loops

```
for(i_1 = L_1; i_1 <= U_1; i_1+ = T_1){
   for(i_2 = L_2; i_2 <= U_2; i_2+ = T_2){
      ...
         for(i_n = L_n; i_n <= U_n; i_n+ = T_n){
S_1:       A[f_1(i_1, i_2, ..., i_n), f_2(i_1, i_2, ..., i_n), ..., f_m(i_1, i_2, ..., i_n)] = ...
S_2:       ... = A[g_1(i_1, i_2, ..., i_n), g_2(i_1, i_2, ..., i_n), ..., g_m(i_1, i_2, ..., i_n)]
         }
      ...
   }
}
```

Listing 1, illustrates the general form of an n-dimensional perfectly nested loop with loop indices $i_1, i_2, ..., i_n$ from the outermost to the innermost loop respectively. Each loop with index k, in the loop nest has a lower bound L_k, an upper bound U_k and a loop step of T_k. The body of the innermost loop in perfect loop nest contains some statements that usually have some arithmetic calculation on arrays. The subscript of arrays is an affine function (f, g) of loop indices [5].

In this paper, we consider the problem of parallelization and vectorization of perfectly nested loops with dependences on modern architecture with cores and short-SIMD. With respect to the different sources for parallel execution including multiple cores and vector-SIMD, there are two kinds of loops: (a) loops that do not carry any dependences, that is, parallel loops which can be executed without any problem in different cores, (b) loops that do carry dependences, that is, this loops have to be executed serially. Among two different approaches for parallel execution namely cores and vector-SIMD, later one requires more constraints than cores. In addition to not having dependence carried by the loop, in order to vectorize them effectively without any overhead it is better the data to be contiguous in the memory. So, a few factors must be considered to run a nested loop on modern architectures with two different sources for parallel execution including multiple cores and vector-SIMD:

1. Multiple cores are used for coarse-grained parallelism, so having parallel loop at outermost level of nested loop is required to get better performance. In case the parallel loop is not in the outermost position, we have to move candidate parallel loop to the outermost level if possible;
2. In order to use vector-SIMD to execute different iterations of a loop in parallel, it is better the data accessed by that loop be contiguous in the memory; in arrays, if it is row-major in memory, the contiguous data is the data in the last dimension of the array, that is, rightmost subscript in the arrays. Else, the contiguous data is the ones in the first dimension of the array, that is, leftmost subscript in the arrays (so contiguous data can be packed in vector registers and can be operated on by vector instructions);
3. Loops have to be dependence-free since our focus is on loops for executing in multiple cores or vector-SIMD.

A number of transformations have to be applied to the original loop nests to change the execution order such that the transformed loop nest can be parallelizable and vectorizable simultaneously. For our purpose, we have used loop strip-mining and loop interchange transformations. However, the application of a transformation on a loop nest is legal if it preserves the original data dependences in the loop nest. Polyhedral model is a powerful abstraction model that can be used to model loop nests, so transformations on each loop nest can be easily expressed. Also, the legality of transformation can be verified easily [12]. We have used Integer Linear Programming (ILP) with an objective function and a built-in legality checking constraints to obtain transformation. With this transformation, we try to move the best candidate vectorizable loop to the innermost position and the best candidate parallelizable loop to the outermost position. However, there are some loop nests that do not have one of these parallelizable/vectorizable candidate loops simultaneously. In this case, we

perform a loop strip-mining transformation before using ILP. For further description see Sec. 3. After having a parallel loop at the outermost position and vector loop at the innermost position, we generate intrinsic vector codes for the innermost vector loop and its statements.

1.3. Contributions. The main contributions of this paper are as follows:

- We propose a unified approach for parallelizing and vectorizing perfectly nested loops simultaneously with respect to the data locality.
- We use ILP with an objective function to get a proper transformation in order to parallelize, vectorize, and improve data locality of the given nested loop at the same time.
- To vectorize innermost vectorizable loop, we propose to generate intrinsic vector code from binary expression tree of the statements inside the loop body.

1.4. Paper outline. The remaining sections of this paper are organized as follows: after introduction, Sec. 2 includes mathematical basic concepts and a brief review of some related works. Section 3 includes the proposed approach. Section 4 includes evaluation and experimental results. Finally, Sec. 5 is dedicated to the conclusions.

2. Polyhedral Model and Related Work. In this section, after introducing required concepts for the polyhedral model, related works are presented.

2.1. Polyhedral Model. An important part of parallelizing/optimizing compilers is the transformation of programs. In this paper, by program transformation we mean loop-restructuring transformation since we focus on the loops. In order to apply transformations on loops easily and effectively, the abstract representation model of the loops is required. Polyhedral model is a powerful and flexible mathematical framework used for representing programs. It is based on a linear algebraic representation of programs [1, 13]. However, not all loops can be represented by polyhedral model except static control parts (SCoPs) [13, 14]. A SCoP is defined as a maximal set of consecutive statements, where loop bounds and conditionals are affine functions of the surrounding loop indices and parameters. The majority of scientific and engineering calculation kernels belongs to this class or can be re-expressed as SCoP [12].

The polyhedral model has four basic components to represent loops: (1) the iteration domain, (2) the access function, (3) dependence polyhedral and (4) the transformation.

The **iteration domain** is used to represent the dynamic instances of each statement in the loop body using a system of affine inequalities. The system of affine inequalities is derived from loop bounds enclosing the statement. Each derived inequality has to be expressed in the $Ax + b \geq 0$ form and iteration domain of a statement is expressed in the $D = \{\overrightarrow{x} \,|\, \overrightarrow{x} \in Z^n, A\overrightarrow{x} + \overrightarrow{b} \geq \overrightarrow{0}\}$ form where $\overrightarrow{x} = (x_1, x_2, ..., x_n)^T$ is the iteration vector representing a loop nest with depth n and x_k is the k^{th} loop index in the loop nest. The possible value for each loop index is an integer, so each entry of the iteration vector is the member of Z.

For instance, in Listing 2 , the iteration vector is $\overrightarrow{x} = (i, j)^T$ and the iteration domain of statement S is $D_S = \{(i, j)|0 \leq i \leq N, 0 \leq j \leq M\}$ and its matrix form is as (2.1).

Listing 2
The convolution kernel

```
for(i = 0; i <= N; i++){
    for(j = 0; j <= M; j++){
S:      y[i] = y[i] + b[j] × x[i + j]
    }
}
```

$$D_S = \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{bmatrix} 0 \\ N \\ 0 \\ M \end{bmatrix} \geq \overrightarrow{0} \tag{2.1}$$

The **access function** is used to represent an access from each statement within loop nest to the data space.

There is an instance of memory reference for each point in the iteration domain of some statement like S. Each instance of memory reference accesses a cell of array A that is shown by access function $F_A^S(\overrightarrow{x}_S)$.

The **dependence polyhedral** is used to represent the data dependences in the loop nest. There is a data dependence between statements S_1 and S_2 if and only if both statements refer to the same memory location (that is the same cell of array) and at least one of them is a write, the statement that executes first is the source and the statement that executes later is the sink. The dependence polyhedral is essential in parallelizing/optimizing loops since they determine the transformations that preserve the meaning of the program and it models the dependences in the program using a set of equalities and inequalities. For example dependence polyhedral for Listing 2 is $D_{S,S} = \{(i,j,i',j') | 0 \leq i \leq N, 0 \leq j \leq M, 0 \leq i' \leq N, 0 \leq j' \leq M, i = i'\}$ and its matrix form is as (2.2).

$$D_{S,S} = \left[\begin{array}{cccc} 1 & 0 & 1 & 0 \\ \hline 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & -1 \end{array}\right] \begin{pmatrix} i \\ j \\ i' \\ j' \end{pmatrix} + \left[\begin{array}{c} 0 \\ \hline 0 \\ N \\ 0 \\ M \\ 0 \\ N \\ 0 \\ M \end{array}\right] \begin{array}{c} = 0 \\ \hline \geq \overrightarrow{0} \end{array} \tag{2.2}$$

The **transformations** in the polyhedral model are used to change the iteration order of original loop nests. A program transformation in the polyhedral model is represented by a function that determines the logical order of statement instances based on surrounding loop indices of the corresponding statement, called scheduling function [14, 15].

Loop transformations have been applied to improve instruction scheduling, register use and cache locality and also to expose parallelism [16]. There are a lot of transformations such as loop reversal, loop interchange, loop distribution, strip-mining and so on. Because some loop transformations only reorder the iterations of a loop nest these are categorized into one group named Iteration-reordering transformations such as loop interchange and strip-mining; in addition to iteration-reordering some other loop transformations also reorder statements, statement instances and/or the operations within a statement such as loop distribution [16].

Some loop transformations can be represented using a scheduling matrix Θ. For example in loop iterators $\begin{pmatrix} i \\ j \end{pmatrix}$ of a 2-d loop nest, the loop interchange transformation can be shown by $\Theta \begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$ and loop reversal transformation can be represented by $\Theta \begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$. Because the final aim of transformation stage in this work is to expose hidden parallelism as a coarse-grained parallelization as well as a fine-grain vectorization, the most useful transformations are the strip-mining and the loop interchange; however loop strip-mining cannot be represented as a matrix.

Loop strip-mining is a transformation that can be applied to single loops and it has been applied successfully in kernels to run on vector machine platforms [16, 17] and since in short-vector multicore processors architecture there are also vector registers with the short length, loop strip-mining can be also applied. Listing 3 shows the loop strip-mined version of Listing 2. P in Listing 3 is the number of available cores.

Listing 3
The strip-mined convolution kernel

```
k=ceil(N/P);
for(I = 0; I <= N; I+ = k){
    for(i = I; i <= min(I + k − 1, N); i + +){
        for(j = 0; j <= M; j + +){
S:          y[i] = y[i] + b[j] × x[i + j];
        }
    }
}
```

2.2. Legality of Transformations in Polyhedral Model. In order to parallelize/vectorize loops during the process of loop transformation, the semantic of the program should not change, that is, transformed program should generate the same outputs with the same order as the original program; otherwise, the transformation is not legal. In order to have a legal transformation, it must satisfy (preserve) dependences in the program. There are four kinds of data dependences: input-dependence, true-dependence, anti-dependence and output-dependence [5]. However, violating the input-dependence does not affect the meaning of a program.

Given data dependence e between statements S and T, dependence e is said to be satisfied if for all pairs of dependent instances $\langle \overrightarrow{X}_S, \overrightarrow{X}_T \rangle$ the source is executed before the sink. In other words, the schedule of the source instance is smaller than the schedule of the sink instance, i.e., if Θ represents the scheduling of statements, for all pairs of dependent instances: $\Theta(\overrightarrow{X}_S) < \Theta(\overrightarrow{X}_T)$ and for statements the schedule of source statement, S, is smaller than the schedule of sink statement T: $\Theta(S) < \Theta(T)$. Applying a transformation Θ on a program is legal if and only if (2.3) holds.

$$\forall e \in E \forall \langle \overrightarrow{X}_S, \overrightarrow{X}_T \rangle \in D_{S,T} \Theta_T(\overrightarrow{X}_T) - \Theta_S(\overrightarrow{X}_S) \geq 0 \tag{2.3}$$

where $D_{S,T}$ is dependence polyhedral of statements S and T.

2.3. Related Work. Parallelization and vectorization are two important techniques to execute different iterations of loops simultaneously. However, most of the previous works have considered only one of them; while in order to get better performance, parallelization and vectorization have to be considered together since modern processors support both types of parallelization. Generally, works in loop optimization can be categorized into three different fields: improving data locality with loop transformations, loop tiling, and data transformations.

Data locality is a very important factor in improving the performance of the programs and especially loops [18, 19]. One of the important methods to deal with data locality is loop transformation such as loop fusion and loop distribution and others. Naci in 2007 [20] tried to improve data locality of loops with loop transformations. Ozturk in 2011 [21] presented a method based on constraint satisfaction problem (CSP) to handle data locality and parallelizing of loop nests. Parsa et al. [22] proposed a method to expose coarse-grain parallelism by reusing data to execute loops on multicore processors. They tried to find a scheduling function for nested loops using a polyhedral model that makes iterations of outer loops independent and consequently parallel. To find such a transformation (scheduling function), they tried to satisfy dependences in inner loops. Doing so, the reuse distance of the data is reduced. It is notable to mention that making outer loops dependence-free and moving dependencies to inner loops has achieved just by the only main idea: to satisfy dependencies at inner loops. To use vector-SIMD units of processors, the innermost loops have to be dependence-free, that is, either the innermost loop does not carry any dependences or if it carries some dependences, then another candidate dependence-free loop should be moved to the innermost position if it is legal. However with their idea, they make the innermost loops to carry dependencies and as a result, they miss vectorization chance. Not only they have not considered the important technique of vectorization, but also their idea inhibits any next vectorization.

Tiling is a vital technique to gain both data locality and coarse-grained parallelization of loop nests [23, 24]. Parsa et al. [25] presented a genetic algorithm (GA) to tile the iteration space of the nested loops with more than three dimensions. They tiled the iteration space in a way that each individual tile can be executed on different processors simultaneously. Since the size and the shape of tiles are uniform, they just consider a tile

in iteration space as a chromosome in GA. This method has some advantages including having no restrictions on loop dimensions and also after tiling, nested-loops can be executed in parallel. However, vectorization is not considered in this work. Krishnamoorthy et al. in 2007 [26] presented a method to tile loops for eliminating pipelined start and drained finish of the tiles. With this method in addition to improving data locality and gaining coarse-grained parallelization, they addressed the problem of unbalanced execution of tiles in multi-core systems. Bondhugula et al. in 2008 [27] proposed an iteration space tiling method based on polyhedral to execute on multicore processors. In this work, they presented a cost model to obtain parallelization and data locality at the same time. The aims of this tiling are minimization of communications and increasing data reuse in each processor. The advantage of this work is that it can be used on loops with any dimensions and the disadvantages of this work are the delays of pipelined start-up and drain of tiles and also vectorization has not been considered in this work. Bandishti et al. in 2012 [28] presented a new tiling method. This method was replaced traditional parallelopiped tiling in Pluto compiler.

Data layout transformation is another technique that has been using for preparing loops both to enable vectorization and also for improving data locality in the new architectures [29, 30]. Lu in 2009 [31] presented a data layout transformation method for optimizing data locality in multi-processor architecture of Non-Uniform Cache Architecture (NUMA). Jang et al. in 2010 [32] presented a mathematical model for acquiring access patterns of data and then computing the most proper data transformation in order to vectorize loops.

A number of the works have addressed the problem of parallelization and vectorization of the programs, however, they either focused on a very special kind of CPU like iPSC-VX or they tried to parallelize or vectorize a special programs only. These works are reviewed in the following section.

Krechel et al. [33] investigated parallelization and vectorization strategies for the solution of large tridiagonal linear systems on MIMID computers which have vector processors. In their experiments, they try to minimize communication on message-based computing systems and maximize the parallelism. In particular, they presented a distribution of large tridiagonal systems across different processes on iPSC2-VX architecture. Aykanat et al. [34] implemented Conjugate Gradient algorithm for exploiting both parallelization and vectorization on iPSC-VX/d2 architectures with 2-dimensional hypercube and used it to solve large sparse linear systems of equation. They achieve efficient parallelization by reducing communications and overlapping computations on the vector processor with internode communication. Crawford et al. [35] proposed a vectorization and parallelization approach for the application of the adiabatically-adjusting principal axes hyperspherical(APH) Hamiltonian used in time-dependent quantum reactive scattering calculations. They separate Hamiltonians for APH coordinates into its constituent parts and vectorize these to speedup computations required in matrix multiplications and then parallelize.

3. The Proposed Approach. In this section, we propose an approach for parallelizing and vectorizing perfectly nested loops with dependences on modern architecture with cores and short-SIMD.

There are two different cases in the original loop nest that have been dealt with in the proposed method. In the first case, the original loop nest has more than one dependence-free loop one of whose has contiguous data in the memory and in the second case the original loop nest has just one dependence-free loop whose data is contiguous in the memory. In the first case, employing loop strip-mining is not required and it is sufficient to use loop interchange transformation to move parallel/vector loop to proper positions.

Listing 4
The summation kernel, C=A+B

```
for(j = 0; j <= N; j++){
    for(i = 0; i <= M; i++){
S:      C[i,j] = A[i,j] + B[i,j]
    }
}
```

For example the loop nest in Listing 4 has two dependence-free loops but because the arrays are stored in row-major, in order to enable vectorization, loop with j index has to be moved to the innermost position (Listing 5).

LISTING 5
The loop interchanged summation kernel, C=A+B

```
for(i = 0; i <= M; i++){
    for(j = 0; j <= N; j++){
S:      C[i,j] = A[i,j] + B[i,j]
    }
}
```

However, in the second case where there is just one dependence free loop, the usage of loop strip-mining is unavoidable to have both parallel and vector loops. For example in order to parallelize and vectorize the loop nest in Listing 2, since there is just one dependence-free loop, using loop strip mine with step sizes k on loop i results two dependence-free loops I, and i (Listing 3) that loop i can be vectorized. In order to vectorize loop i, it has to be moved to the innermost position by a proper transformation.

In addition to the cores and short-vectors, the data locality has a significant effect on the execution time of the programs. So, besides parallelization and vectorization, we try to find a transformation that has improved data locality according to the storage layout of the array whether it is row-major or column-major. In this order, we have used an ILP-based method to obtain proper transformation.

3.1. ILP formulation. In this section, a new ILP based method, PVL, is proposed to obtain proper loop interchange transformation that enables coarse-grained parallelism and fine-grained vectorization. Generally, when formulating a problem in ILP, there is an objective function we want to maximize/minimize and some constraints with respect to which the objective function has to be satisfied.

It has been shown [5] that cores are best suited to coarse-grained computations in new short-vector multicore architectures and the computations of outer loops are more coarse-grained than inner loops in loop nests. Hence, in the loop nest, it is better to move the parallel loop to the outermost position- that is, the outer the loop, the more coarse-grained the computations. On the other hand, parallelism is fine-grained in the short-vectors. Loops with contiguous data which are in the innermost position can be executed in short-vectors.

3.1.1. Formulating Objective Function. Loop interchange transformation is a square matrix $n \times n$ which n is the dimension of the original loop nest and it can be represented as a matrix in which entries belong to $\{0, 1\}$, such that there is just a single one in each row as well as each column; the original scheduling of strip-mined convolution is as (3.1) where the iteration order of strip-mined convolution would not change by this transformation.

$$\Theta = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.1}$$

Having just a single one in each row in loop interchange transformation, it can be formulated as a constraint in ILP (3.2).

$$\sum_{i=1}^{|dimensions|} c_i = 1 \tag{3.2}$$

To obtain proper transformation using proposed ILP based method, transformation Θ will be produced row by row. For a three-dimensional loop, each row of Θ in general, is denoted by (c_i, c_j, c_k) such that each entry has a value. Hence, in order to get proper values of c_i, c_j, and c_k, the objective function of ILP has to be a function of row's entries such that it satisfies parallelization, vectorization, and data locality in the transformed loop nest. In order to create the objective function, it is required to notice the behavior of the transformation matrix. For a three-dimensional loop nest, given loop interchange transformation Θ is composed of three rows:

$$\Theta = \begin{bmatrix} c_{1i} & c_{1j} & c_{1k} \\ c_{2i} & c_{2j} & c_{2k} \\ c_{3i} & c_{3j} & c_{3k} \end{bmatrix}$$

TABLE 3.1
Creating the objective function for a sample 3-dimensional loop

coefficients	c_i	c_j	c_k
P	$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$
V	$\frac{1}{3}$	$\frac{1}{3}$	3
l_1	$\frac{1}{3}$	$\frac{2}{3}$	$\frac{3}{3}$
l_2	$\frac{1}{3}$	$\frac{2}{3}$	$\frac{3}{3}$
l_3	$\frac{1}{3}$	$\frac{2}{3}$	$\frac{3}{3}$
$L = \sum l_i$	$\frac{3}{3}$	$\frac{6}{3}$	$\frac{9}{3}$
$P + V + L$	$\frac{5}{3}$	$\frac{8}{3}$	$\frac{19}{3}$
$n \times (P + V + L)$	5	8	19
objective function	$min 5c_i + 8c_j + 19c_k$		

In this transformation, each row is dedicated to a loop through loop nest, and each column is representing the position of the corresponding loop (row) in the loop nest, that is, if the first row of Θ is $(0, 1, 0)$, it denotes that the position of the first loop in transformed loop nest would be the second position. As explained so far, with respect to the final aim of parallelization and vectorization at the same time in the loop nest, it is required that first row and last row of the transformation Θ show the parallel and the vector loops, respectively. However, since in the proposed method transformation Θ is obtained row by row from the outermost row to the innermost row, it is clear that the parallel loop has to be selected at first and the vector loop has to be selected at last. In the following transformation Θ, the row denoting parallel loop is highlighted as green (box with dashed line) and the row denoting vector loop is highlighted as blue (box with dotted line).

$$\Theta = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & \cdots & c_{nn} \end{bmatrix}$$

With these explanations, in the objective function of the proposed method the priority of the parallel loop is selected as the highest and the priority of vector loop is selected as the lowest. In the other hand, loop nests may have some loops neither parallelizable nor vectorizable; these loops are the loops with dependence and transformation Θ is legal if it satisfies all dependences in the loop nest. Since the first and last rows represent parallel and vector loops respectively, so dependences have to be satisfied in the middle rows highlighted as red (box with solid line) in the transformation, so the priority of loops with dependence is mid. By middle rows we mean the rows of the transformation Θ that are between the first and the last rows. Because the formulated ILP problem in Sec. 3.1.1 is a minimization problem, the loops with the highest priority (parallel loops) have smallest coefficients and similarly the loops with the lowest priority (vector loops) have the highest coefficient. Hence, the objective function of sample three-dimensional loop nest in Listing 6 is created according to the Table 3.1.

LISTING 6
Sample 3d-loop

```
for(k = 0; k <= 100; k++){
    for(j = 0; j <= 100; j++){
        for(i = 0; i <= 100; i++){
S:          A[i,j,k] = A[i,j,k] + B[i,j,k];
        }
    }
}
```

In this loop nest, there are three parameters to make up an objective function: data locality, parallel, and

vector which are described as follows:

Data locality in this paper means whether the arrays are stored in the memory as row-major or column-major. There is a row (i.e. $l_1, l_2, ...$) in the table for each reference. The final locality, L, is defined as $L = \sum l_i$. Values of each l_i in the table is computed as follows: if arrays are stored as row-major in the memory, $l_{i,j} = \frac{j}{dimension_{loop}}$, where l_{ij} is the value in column c_j of row l_i. If arrays are stored as column-major in the memory, $l_{i,j} = \frac{dimension_{loop}-j+1}{dimension_{loop}}$. In this example, it is assumed that the arrays are stored as row-major. So, the first row of data locality, l_1, is for the first reference in the statement within loop nest i.e. $A[i, j, k]$ (left-hand-side). The values of this row are $\frac{1}{3}, \frac{2}{3}$, and $\frac{3}{3}$. In this example $dimension_{loop} = 3$ and because the loop index i is the first subscript in the reference, so its value is $\frac{1}{3}$, and so on.

Parallel parameter denotes the loops which its iterations can be executed in parallel. Values of this row in the table belong to $\{n, \frac{1}{n}\}$ where n is the number of dimensions. The presence of n means that the iterations of the corresponding loop can not be executed in parallel, and the presence of $\frac{1}{n}$ means that the iterations of the corresponding loop can be executed in parallel. In Listing 6, iterations of all loops can be executed in parallel, so the values of all entries of this row are $\frac{1}{n}$.

Vector parameter denotes the loops with contiguous data which do not carry any dependence. Values of this row in the table belong to $\{n, \frac{1}{n}\}$, where n is the number of dimensions. The presence of n means that the corresponding loop can be vectorized and the presence of $\frac{1}{n}$ means that the corresponding loop is not vectorizable. In this example, the loop with index k can be vectorized. So in the table, vector parameter of this loop is $\frac{1}{3}$ (here $n = 3$) and other loops are 3.

Finally, the parameters of Table 3.1 is used as $n \times (P + V + L)$ to obtain the objective function, where n is the number of dimensions and P, V, and L are parallel, vector and locality parameters, respectively.

3.1.2. Automatically Getting Objective Function from Data Dependences. In order to automatically obtain objective function from data dependences, Algorithm 1 is proposed. As explained in the Sec. 3.1.1, the loops that could be executed in parallel will be run in the cores, because the formulated ILP problem in the previous section is minimization, in step 1 the algorithm gives the lowest coefficient $\frac{1}{n}$ for all of them, denoting that one of these loops should be scheduled first. The loops that could be vectorized will be run in the short-vectors within each core, so the algorithm gives the highest value n for them, denoting that one of these loops should be scheduled last, where n is the number of dimensions. The arrays storage layout is used in the algorithm to have better data locality. In step 2, the objective function is get by $P + V + L$ and since all terms of objective function have denominator n, the final objective function is get by $n \times (P + V + L)$ which omits denominators.

3.2. Formulating Constraints. In the following subsections, constraints of the problem are formulated for ILP

3.2.1. Legality of Transformation. One of the benefits of using polyhedral model is that legality of the obtaining transformation can be verified with a built-in set of constraints obtained from Rel. 2.3 [36]. To create the set of semantic-preserving constraints using Rel. 2.3, for each dependence that is for every dependent statements S and T, the obtained legality constraint from (2.3) is (3.3) and in the case of non-uniform dependences, the obtained nonlinear constraint is converted to linear using Farkas lemma [1, 22, 37].

$$\Theta_T(\overrightarrow{X}_T) - \Theta_S(\overrightarrow{X}_S) \geq 0 \tag{3.3}$$

3.2.2. Linearly independence rows. After obtaining the transformation Θ, it is required to be applied on the original polyhedron $A\overrightarrow{x} + \overrightarrow{b} \geq \overrightarrow{0}$ defining the original loop nest. Scattering function leading to the target index $\overrightarrow{y} = \Theta\overrightarrow{x}$, the polyhedron of the transformed loop nest is defined by (3.4) [12, 15].

$$(A\Theta^{-1})\overrightarrow{y} + \overrightarrow{b} \geq \overrightarrow{0} \tag{3.4}$$

The transformation Θ is used as inverted (Θ^{-1}) in the polyhedron of the transformed iteration space, relation (3.4), so it is required to be invertible. On the other hand, transformation Θ is obtained row by row, so

Algorithm 1 Automatically obtain objective function for ILP

Input: Data Dependence Matrix
Output: Objective Function

Step1: //for each loop in the loop nest fill the parallel and vector parameters of
//Objective Table (OT), Table 3.1 in the paper
n=number of dimensions
for loop index i=1 **to** number of dimensions **do**
  //parallel
  if (loop L_i carries any dependences) **then**
    $OT['parallel', i] = n$;
  else
    $OT['parallel', i] = \frac{1}{n}$;
  end if

  //vector
  if (loop L_i does not carry any dependences AND Loop index of L_i has contiguous data in the memory) **then**
    $OT['vector', i] = n$;
  else
    $OT['vector', i] = \frac{1}{n}$;
  end if
end for

//locality
for each reference within loop nest **do**
  for loop index i=1 **to** number of dimensions **do**
    if arrays are stored as row-major in the memory **then**
      $OT['locality', i] += \frac{j}{dimension_{loop}}$ if this loop index, i, is the jth subscript of the array
    else
      $OT['locality', i] += \frac{dimension_{data} - j + 1}{dimension_{loop}}$ if this loop index, i, is the jth subscript of the array
    end if
  end for
end for

Step 2:
for loop index i=1 **to** number of dimensions **do**
  for each of the parameters (parallel, vector, and locality) **do**
    coefficient of the loop L_i in the objective function, $coefficient_i =$
    $n \times (OT['parallel', i] + OT['vector', i] + OT['locality', i])$;
  end for
end for

in order to Θ be invertible, its rows have to be linearly independent. To this order, in the process of obtaining Θ, after obtaining each row a new constraint is added to the set of the constraints of ILP formulation of new row by (3.5) that ensures the solutions are linearly independent [1, 38, 39].

$$\Theta_S^{\perp} = I - \Theta_S^T(\Theta_S\Theta_S^T)^{-1}\Theta_S \tag{3.5}$$

$$\forall i, \Theta_S^{i\perp}\theta_S^* \geq 0 \wedge \sum_i \Theta_S^{i\perp}\theta_S^* \geq 1 \tag{3.6}$$

In (3.5) and (3.6), Θ_S is the obtained transformation for statement S so far, Θ_S^T is the transposition of Θ_S, I is the identity matrix and θ_S^* is the next row to be found for statement S.

3.3. The Proposed Algorithm. Algorithm 2 is proposed to obtain the transformation used to parallelize and vectorize nested loops simultaneously. In the first step, if there is not enough dependence-free loop for both parallelization and vectorization, loop strip-mining transformation will be applied to have enough dependence-free loops for both parallelization and vectorization. The constraints of ILP will be built in the steps 2 and 3. In step 2, the loop interchange constraint will be built e.g. for a three-dimensional loop the loop interchange constraint is as Rel. 3.2. In step 3, legality constraints will be built according to the data dependences of the given loop nest using Rel. 3.3. The transformation matrix Θ (T in the algorithm) will be found in step 4. Step 4 is used in this algorithm to obtain objective function for each row in the transformation matrix Θ. In this step, after finding each row of the transformation matrix, the linearly independence constraint is added to the constraints set using Rels. 3.5 and 3.6.

Algorithm 2 PVL algorithm

Input: Data Dependence Graph (DDG), Dependence polyhedron ($D_{S,T}$) for each edge in DDG
Output: Transformation matrix T

Step 1:
if (there is not enough dependence-free loop) **then**
  Apply loop strip-mining
end if

Step 2: //build loop interchange constraint
Add $\sum_{i=|dimensions|} c_i = 1$ to constraints-set

Step 3: //build legality constraints
for (each dependence e in DDG) **do**
  Build legality constraints LC_e and add to constraints-set
  if (e is nonuniform) **then**
    Apply Farkas lemma and get linearized constraints
  end if
end for

Step 4: //obtain transformation matrix T using ILP formulation
for (each dimensions of original loop nest) **do**
  //objective function:
  obj_fun = Get new objective function using Algorithm 1 (input:D)
  r = ILP-SOLVE(obj_fun, constraints-set)
  Add row r to T
  Eliminate satisfied dependences from D
  Add linearly independent constraints to constraint-set
end for

3.4. Intrinsic Vector Code Generation. After applying the obtained transformation to the original loop nest. OpenMP is used to parallelize the outermost parallel loop and the following intrinsic vectorization approach is used for vectorizing the innermost vector loop. In this approach after determining loop index of the innermost loop, a binary expression tree is created for each statement within the body of the innermost loop. The binary expression tree is traversed as post order and for each leaf of the tree, an intrinsic load code and for each non-leaf node of the tree corresponding intrinsic vector operation are generated. In order to vectorize inner most vectorizable loop nest i, we want all operations as SIMD. Consider statement y[i]=y[i]+b[j]×x[i+j],

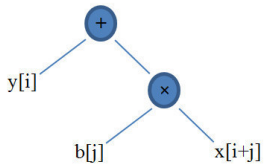


FIG. 3.1. *The binary expression tree of y[i] = y[i]+b[j]×x[i+j]*

two states are possible. In the first state, the subscript of reference in the statement contains loop index i, such as y[i] in this statement. In this state, the intrinsic vector load will load four contiguous elements y[i], y[i+1], y[i+2] and y[i+3] of array y. In the second state, the subscript of reference in the statement does not contain loop index i, such as b[j] in this statement. In this state, the corresponding register vector will load the same element b[j] four times (b[j], b[j], b[j], and b[j]). For example if the statement of the innermost loop is y[i]=y[i]+b[j]×x[i+j] and the loop index of the innermost loop is i, binary expression tree and corresponding intrinsic vector codes are as Fig. 3.1 and Listing 7, respectively. If the current statement within loop nest requires more vector registers than available vector registers in the underlying architecture, then we split that statement into some sub-statements and store result of each sub-statement in a temporary vector register than gather all temporary vector registers in order to make up the result of that statement.

LISTING 7
Intrinsic vector code of Fig. 3.1

```
__m128 y4=_mm_loadu_ps(&y[i]);
__m128 b4=_mm_set1_ps(b[j]);
__m128 x4=_mm_loadu_ps(&x[i+j]);
y4 = _mm_add_ps(_mm_mul_ps(b4, x4), y4);
_mm_storeu_ps(&y[i], y4);
```

3.5. Example. In this section, Algorithm 1 and Algorithm 2 are used step by step in order to obtain a proper transformation for following loop nest, which upper bound of loops, N, is divisible by four:

LISTING 8
Example loop nest

```
for(i = 1; i < N; i++){
    for(j = 1; j < N; j++){
        for(k = 0; k < N; k++){
            for(l = 0; l < N; l++){
S:              A[i,j,k,l] = A[i-1,j-1,k,l] + B[i-1,j-1,k,l];
            }
        }
    }
}
```

The iteration domain polyhedron for statement S within the loop nest is:

$$D_S = \{(i,j,k,l) | 1 \leq i \leq N-1, 1 \leq j \leq N-1, 0 \leq k \leq N-1, 0 \leq l \leq N-1\}$$

The dependence polyhedron for flow dependence (RAW: read after write) from the write at $A[i,j,k,l]$ to the read at $A[i-1,j-1,k,l]$ for any two iterations, $I = (i,j,k,l)$ and $I' = (i',j',k',l')$ is:

$$\begin{aligned} D_{S,S} = \{(i,j,k,l,i',j',k',l') | & 1 \leq i \leq N-1, 1 \leq j \leq N-1, 0 \leq k \leq N-1, 0 \leq l \leq N-1, \\ & 1 \leq i' \leq N-1, 1 \leq j' \leq N-1, 0 \leq k' \leq N-1, 0 \leq l' \leq N-1, \\ & i' = i-1, j' = j-1, k' = k, l' = l\} \end{aligned}$$

TABLE 3.2
Creating the objective function for the first row of the transformation matrix for the example loop nest

coefficients	c_i	c_j	c_k	c_l
P	4	4	$\frac{1}{4}$	$\frac{1}{4}$
V	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	4
l_1	$\frac{1}{4}$	$\frac{2}{4}$	$\frac{3}{4}$	$\frac{4}{4}$
l_2	$\frac{1}{4}$	$\frac{2}{4}$	$\frac{3}{4}$	$\frac{4}{4}$
l_3	$\frac{1}{4}$	$\frac{2}{4}$	$\frac{3}{4}$	$\frac{4}{4}$
$L = \sum l_i$	$\frac{3}{4}$	$\frac{6}{4}$	$\frac{9}{4}$	$\frac{12}{4}$
$P + V + L$	$4 + \frac{1}{4} + \frac{3}{4}$	$4 + \frac{1}{4} + \frac{6}{4}$	$\frac{1}{4} + \frac{1}{4} + \frac{9}{4}$	$\frac{1}{4} + 4 + \frac{12}{4}$
$n \times (P + V + L)$	20	23	11	29
objective function	$\min 20c_i + 23c_j + 11c_k + 29c_l$			

Because this loop nest has two loops with independent iterations, so Step1 of the Algorithm 2 is not required. The loop interchange constraint for this loop nest using Rel. 3.2 is:

$$c_i + c_j + c_k + c_l = 1$$

and the legality constraints for this dependence using Rel. 3.3 is as follows:

$$\begin{aligned}(c_i, c_j, c_k, c_l) \times (i, j, k, l)^T - (c_i, c_j, c_k, c_l) \times (i', j', k', l')^T \geq 0\\ (c_i, c_j, c_k, c_l) \times (i, j, k, l)^T - (c_i, c_j, c_k, c_l) \times (i-1, j-1, k, l)^T \geq 0\\ c_i + c_j \geq 0\end{aligned}$$

Now the constraints of ILP are determined. The objective function of the ILP is created according to the Table 3.2. In this table, because the loops i and j carry dependence, the corresponding values for these loops in row P (parallel parameter) are n and because the iterations of the loops k and l are independent the corresponding values for these loops in row P are $1/n$, which n is number of dimensions ($n = 4$). The values of row V (vector parameter) corresponding with loops i and j are $1/n$ means they are not vectorizable because of the dependence and the value of loop k is $1/n$ denoting that although iterations of this loop are independent but the data corresponding with this loop in the arrays are not contiguous. Hence, the loop k is not preferred to be vectorized. The value of the loop l is n, denotes that this loop is preferred to be vectorized (which n is the number of dimensions, here $n = 4$). The locality parameter is the aggregation of the l_is, the locality status of each reference within the statement of the loop nest. Since in this example data layout is row-major, for each reference the locality is calculated based on $\frac{j}{dimension_{loop}}$, where j denotes that this loop index is jth subscript in the reference from the left. The final objective function is $min 20c_i + 23c_j + 11c_k + 29c_l$. So, the ILP problem for the first row of the transformation matrix is as follows:

min $20c_i + 23c_j + 11c_k + 29c_l$ s.t.
(1) $c_i + c_j + c_k + c_l = 1$;
(2) $c_i + c_j \geq 0$;
(3) $c_i \geq 0$;
(4) $c_j \geq 0$;
(5) $c_k \geq 0$;
(6) $c_l \geq 0$;

The coefficients (c_i, c_j, c_k, c_l) of the first row of the transformation matrix obtained by solving above ILP problem is (0, 0, 1, 0). This row of transformation matrix determines the position of the loop k. Since the RAW dependence is not carried by this loop, after finding the position of this loop, the dependence graph does not change so the objective function for the next row will not change. However, the linear independence constraint will enforce $c_k = 0$ for the next row of the transformation matrix. To obtain the next row of the transformation

TABLE 3.3
Creating the objective function for the third row of the transformation matrix for the example loop nest

coefficients	c_i	c_j	c_k	c_l
P	4	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
V	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	4
l_1	$\frac{1}{4}$	$\frac{2}{4}$	$\frac{3}{4}$	$\frac{4}{4}$
l_2	$\frac{1}{4}$	$\frac{2}{4}$	$\frac{3}{4}$	$\frac{4}{4}$
l_3	$\frac{1}{4}$	$\frac{2}{4}$	$\frac{3}{4}$	$\frac{4}{4}$
$L = \sum l_i$	$\frac{3}{4}$	$\frac{6}{4}$	$\frac{9}{4}$	$\frac{12}{4}$
$P + V + L$	$4 + \frac{1}{4} + \frac{3}{4}$	$\frac{1}{4} + \frac{1}{4} + \frac{6}{4}$	$\frac{1}{4} + \frac{1}{4} + \frac{9}{4}$	$\frac{1}{4} + 4 + \frac{12}{4}$
$n \times (P + V + L)$	20	8	11	29
objective function	$min 20c_i + 8c_j + 11c_k + 29c_l$			

matrix, the linear independence constraint is added to the constraints set according to the Rels. 3.5 and 3.6:

$$\Theta_S^{\perp} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

hence, the linear independence constraint $c_i + c_j + c_l \geq 1$ should be added to the constraints set.

The ILP formulation for obtaining the second row of the transformation matrix is as follows:

min $20c_i + 23c_j + 11c_k + 29c_l$ s.t.
(1) $c_i + c_j + c_k + c_l = 1$;
(2) $c_i + c_j \geq 0$;
(3) $c_i \geq 0$;
(4) $c_j \geq 0$;
(5) $c_k \geq 0$;
(6) $c_l \geq 0$;
(7) $c_i + c_j + c_l \geq 1$;

The solution of the above ILP problem gives (1, 0, 0, 0) for the coefficients (c_i, c_j, c_k, c_l) of the second row of the transformation matrix. This row of transformation matrix determines the position of the loop i. To obtain the next solution for the third row, linear independence constraint $c_j + c_l \geq 1$ should be added to the constraints set. Because the RAW dependence is carried by this loop, after finding the position of this loop, this dependence will be satisfied and therefore will be removed from the dependence graph and constraints set ($c_i + c_j \geq 0$). So, the objective function for the next row will change as Table 3.3. The positions of the loops i and k are determined already. Therefore, because of linear independence constraint the values of coefficients corresponding with the loops i and k in the next rows will be zero. Hence, their columns in the Table 3.3 will not have any effects on the solution of the next row. So, we do not change the values of these columns in Table 3.3.

The ILP formulation for obtaining the third row of the transformation matrix is as follows:

min $20c_i + 8c_j + 11c_k + 29c_l$ s.t.
(1) $c_i + c_j + c_k + c_l = 1$;
(2) $c_i \geq 0$;
(3) $c_j \geq 0$;
(4) $c_k \geq 0$;
(5) $c_l \geq 0$;
(7) $c_j + c_l \geq 1$;

The solution of the above ILP problem gives (0, 1, 0, 0) for the coefficients (c_i, c_j, c_k, c_l) of the third row of

the transformation matrix. This row of transformation matrix determines the position of the loop j. To obtain the next solution for the fourth row, linear independence constraint $c_l \geq 1$ should be added to the constraints set. Because after finding the third row of the transformation matrix the dependence graph does not change, so the objective function for the next row will not change.

The ILP formulation for obtaining the fourth row of the transformation matrix is as follows:

min $20c_i + 8c_j + 11c_k + 29c_l$ s.t.
(1) $c_i + c_j + c_k + c_l = 1$;
(2) $c_i \geq 0$;
(3) $c_j \geq 0$;
(4) $c_k \geq 0$;
(5) $c_l \geq 0$;
(7) $c_l \geq 1$;

The solution of the above ILP problem gives (0, 0, 0, 1) for the coefficients (c_i, c_j, c_k, c_l) of the fourth row of the transformation matrix. Therefore, the final transformation matrix is as follows:

$$\Theta = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Applying the transformation matrix Θ to the original loop nest moves the loop k to the outermost position as a coarse-grained parallel loop and the loop l to the innermost position as a fine-grained vector loop. Listing 9 shows the resultant final loop nest after generating intrinsic vector code for the statement within the loop.

Listing 9
Final output code for example loop nest

```
#pragma omp parallel for schedule(static)
for(k = 0; k < N; k++){
    for(i = 1; i < N; i++){
        for(j = 1; j < N; j++){
            for(l = 0; l < N; l+= 4){
                __m128 a41=_mm_loadu_ps(&A[i−1, j−1, k, l]);
                __m128 b4=_mm_loadu_ps(B[i−1, j−1, k, l]);
                __m128 a42 = _mm_add_ps(a41, b4);
                _mm_storeu_ps(&A[i, j, k, l], a42);
            }
        }
    }
}
```

The final output code of PVL for one-dimensional convolution (Listing 2) is also represented in Listing 10.

4. Evaluation and Experimental Results. The effectiveness of the proposed method was experimentally evaluated using execution on a hardware platform. The effectiveness of the proposed method was first demonstrated using a set containing the benchmarks that usually have used in the field of parallelizing compilers. In order to further study the effectiveness of the proposed method, another set of loop nests containing four synthetic benchmarks are used. Each synthetic benchmark is designed to demonstrate the effectiveness of the proposed method in a special case out of four cases that cover the majority of loop nests in scientific computations. Each benchmark was run 10 times and the average execution time of 10 runs is reported in the tables and figures. We compare the performance of the proposed method with the original programs and also with previous state-of-the-art methods. The execution time test was evaluated using GCC 4.9.2.

Listing 10
Final output code for one-dimensional convolution

```
int k=ceil(n/p);
int m=0;
#pragma omp parallel for schedule(static)
for (int I=0; I<n; I+=k){
    m=min(I+k−1, n);
    for (int j = 0; j<n; j++){
        for(int i=I; i<m; i+=4){
            __m128 y4=_mm_loadu_ps(&y[i]);
            __m128 b4=_mm_set1_ps(b[j]);
            __m128 x4=_mm_loadu_ps(&x[i+j]);
            y4 = _mm_add_ps(_mm_mul_ps(b4, x4), y4);
            _mm_storeu_ps(&y[i], y4);
        }
    }
}
```

4.1. Experimental Setup.

Hardware. The execution time tests were performed on two dual and quad core systems. Dual core system is an Intel Dual Core CPU E2200 with a 2.20GHz frequency, 32KB L1 cache, 1024KB shared L2 cache, 3GBs RAM, and running Linux Ubuntu 12.04 and quad core system is an Intel Core2 Quad CPU Q6600 with a 2.40GHz frequency, 32KB L1 cache, two 4096KB L2 caches, 2GBs RAM and running Linux Ubuntu 12.04. CPU programs were compiled using GCC 4.9.2 with the '-march=native -mtune=native -O3' optimization flags in order to generate the best code for underlying architecture.

Benchmarks. To test the effectiveness of the proposed parallelization and vectorization technique, we use four data-intensive real benchmarks: matrix multiplication, one-dimensional convolution, two-dimensional convolution, and Sobel filter. Matrix multiplication is the core mechanism in linear algebra applications such as image processing, and physical or economic simulations, and improving it can be very useful in real life applications. Both one and two-dimensional convolutions are very regular in Digital Signal Processing (DSP) field and finally, Sobel filter is an application of two-dimensional convolution. The another set contains four synthetic benchmarks covering four cases that are usual in the scientific applications. Table 4.3 demonstrate each case and the synthetic benchmark covering it. This set of benchmarks is listed in Table 4.1. All the benchmarks use single precision floating point operations. Table 4.2 shows problem sizes used in the benchmarks. All array sizes used in the benchmarks were set to be significantly larger than last level cache of the hardware platform.

4.2. Experimental Results.

For the given problem sizes in the Table 4.2, the parallelized and vectorized codes are generated by the PVL method. The experimental evaluations in the follow show that, as theoretically

Table 4.1
The set two of benchmarks consisting four different synthetic benchmarks

`for(j=1; j<n; j++)`
`  for(k=0; k<n; k++)`
`  for(i=0; i<n; i++)`
`  a[i][j][k]=a[i][j-1][k]+1;`
(B1)
	`for(i=0; i<n; i++)`
	`  a[i]=b[i]+c[i];`
	(B2)

`for(i=0; i<n-1; i++)`
`  for(j=0; j<n; j++)`
`  for(k=0; k<n; k++)`
`  a[i+1][j]=a[i][j]+b[k][j];`
(B3)
	`for(i=1; i<n-1; i++)`
	`  for(j=0; j<n; j++)`
	`  a[i][j]=a[i+1][j]+a[i-1][j];`
	(B4)

TABLE 4.2
The Problem sizes used in benchmarks

Real Benchmarks	Problem size	Synthetic Benchmarks	Problem size
Convolution 1d	8192	B1	512
Convolution 2d	2048×2048	B2	16777216
Sobel	4096×4096	B3	1024×1024
Matmul	768×768	B4	8192×8192

TABLE 4.3
Demonstration of the synthetic benchmarks

Benchmark	Dimensions	Has dependence?	Description
B1	3	Yes	A 3-d perfectly nested loop that carries a dependence.
B2	1	No	A 1-d loop nest that does not carry any dependences.
B3	3	Yes	A 3-d perfectly nested loop that carries a dependence.
B4	2	Yes	A 2-d perfectly nested loop that carries a dependence.

expected, when benchmarks are parallelized and vectorized using PVL method, because the PVL attempts to use both cores and SIMD short vectors to execute the benchmark, the execution time of the benchmarks reduce significantly. As a result of using all available resources of the platform, the PVL could approach to the peak performance of the underlying platform. Figures 4.1 and 4.2 show the gained speed up through the proposed approach over the sequential execution of the original programs on both systems. The average speedup of the proposed method on two and four threads (cores) is about 6.56 and 9.17, respectively. As it is clear in the proposed method, the speedup is achieved not only by multiple cores, but (1) through parallelization using multiple cores and (2) vectorization using short-vectors within each core and also (3) trying to optimize data locality to reduce cache miss. Hence, the gained speedup usually is more than number of cores.

Figure 4.3 shows the execution time of the sequential, Pluto [40], Parsa et al. [22] method, and PVL (proposed method) on both sets of benchmarks on two and four cores systems. We compiled the sequential program and output of mentioned three methods using GCC compiler. Figures 4.3 (a) and 4.3 (c) are the execution time of the real benchmarks on two cores and four cores systems, respectively. Figures 4.3 (b) and 4.3 (d) are the execution time of the synthetic benchmarks on two cores and four cores systems, respectively. In all benchmarks, execution time of the PVL (proposed method) is much faster than the others.

4.3. Execution with -O3 flag. The underlying hardware architecture of the proposed method and both compared methods are modern multi-core architectures with given sources for parallel and vector executions: cores and short-vectors. However, both compared methods are giving priority to the cores and postponing vectorization. We claim that in order to have best execution time, both cores and short-vectors have to be given enough importance. So, we try to find loop transformation proper for both cores and short-vectors at the same time. In order to demonstrate our claim, we have compared the proposed method with two main research of the filed in two different scenarios: (1) at first, we have executed the results of the compared methods without any changes on the given hardware. In this scenario both of the compared methods have used only cores for execution of the benchmarks (Fig. 4.1 shows the execution times related to the this case), and (2) in order to also use short-vectors as a postponed transformation (exactly as the original intention of the compared methods), after getting results of compared methods, auto vectorization of the GCC compiler have enabled by using -O3 flag. -O3 flag of GCC not only enables auto vectorization but also it enables the highest level of safe optimizations in the compiler. By this way, both cores and short-vectors are used.

Figure 4.4 shows execution time of the real benchmarks with -O3 flag enabled. As it is obvious, again our proposed method is far superior to sequential, Pluto and Parsa et al. methods which are optimized by -O3 flag. In all four scenarios of both real and synthetic benchmarks, the execution time of the proposed method is less than the other methods. So, the experiments demonstrate that the PVL applies both parallelization and vectorization better than others. Also, experiments support the claim that both cores and short-vectors have

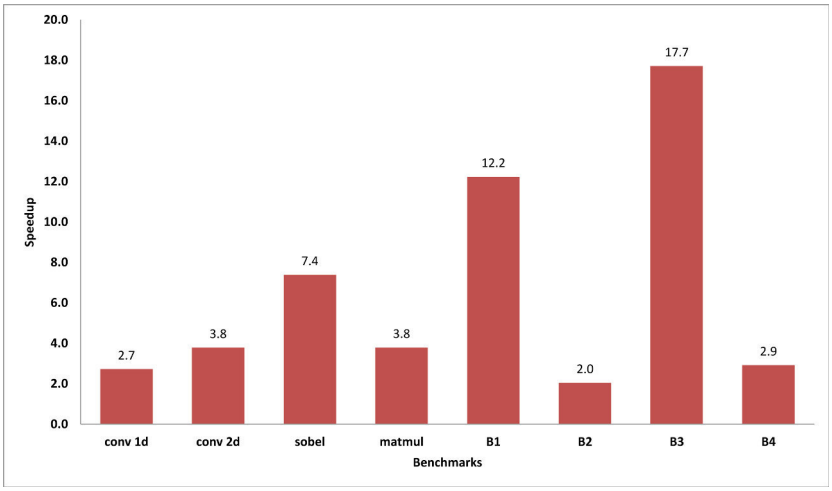


FIG. 4.1. *Speedup of PVL (the proposed method) over sequential programs on Intel Dual Core CPU E2200 @ 2.20GHz CPU.*

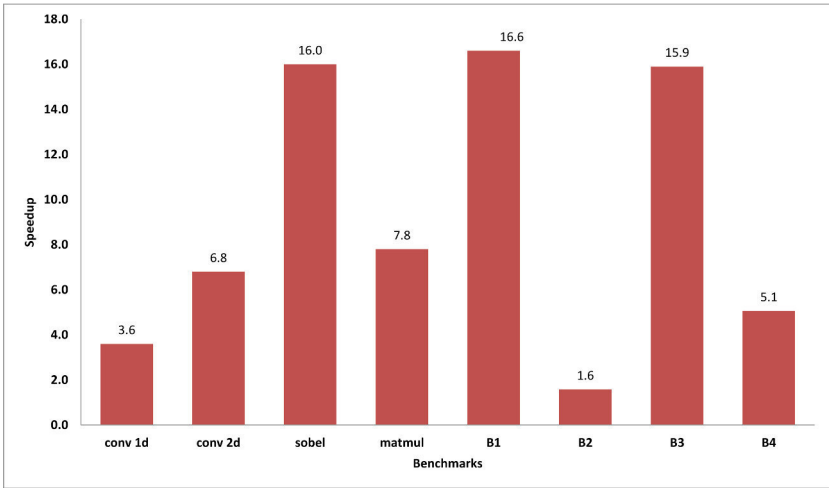


FIG. 4.2. *Speedup of PVL (the proposed method) over sequential programs on Intel Core2 Quad CPU Q6600 @ 2.40GHz CPU.*

to be given enough importance and also transforming schema of the PVL for modern multicore architectures is better that Pluto, and Parsa et al. approach.

Figure 4.5 shows the speedup of the Pluto, Parsa et al. approach, and the PVL over sequential programs on two different systems with/without -O3 flag of the GCC compiler. In both sets of real and synthetic benchmarks, the speedup of the PVL is higher than the speed up of the Pluto, and Parsa et al. method. The average of the achieved speedup for the PVL (proposed method), Pluto, and Parsa et al. method over sequential in two systems with/without -O3 flag is as Table 4.4.

Figure 4.6 shows the improvement in execution time of the PVL (proposed method) over Pluto, and Parsa et al. method. Average improvement percentage in execution time of PVL over Pluto, and Parsa et al. method is as Table 4.5.

5. Conclusion. This paper discusses the idea of parallelizing and vectorizing nested loops on multicore architectures. Because executing loops on cores and on SIMD units demand different necessities, at first, we have described these characteristics and then, our proposed method tries to find proper loop transformation using polyhedral model in order to achieve three objectives: (1) parallelize proper dependences-free loop in

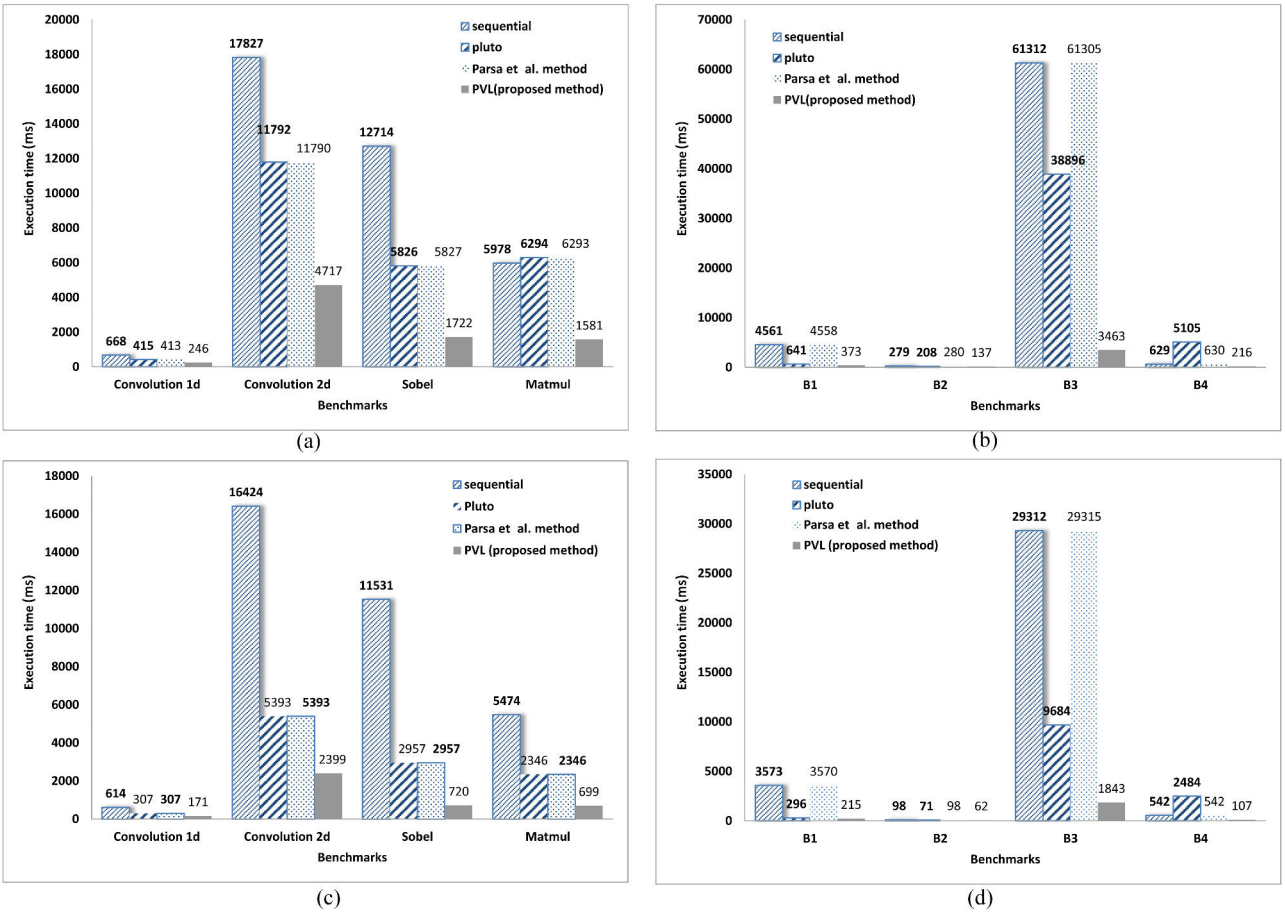


FIG. 4.3. *Execution time of the sequential program, Pluto, the Parsa et al. method and the PVL (proposed method) on Intel Dual Core CPU E2200 @ 2.20GHz and Intel Core2 Quad CPU Q6600 @ 2.40GHz systems. (a) Execution of benchmarks set one on Intel Dual Core CPU E2200 @ 2.20GHz system, (b) Execution of benchmarks set two on Intel Dual Core CPU E2200 @ 2.20GHz system, (c) Execution of benchmarks set one on Intel Core2 Quad CPU Q6600 @ 2.40GHz system, and (d) Execution of benchmarks set two on Intel Core2 Quad CPU Q6600 @ 2.40GHz system*

TABLE 4.4
The average speedup achieved by different methods

	PVL (proposed method)	Pluto	Parsa et al. method
E2200 without -O3	6.7	2.0	1.3
E2200 with -O3	8.2	3.2	1.3
Q6600 without -O3	9.2	3.5	1.9
Q6600 with -O3	8.9	4.6	1.8

TABLE 4.5
The average improvement in execution time over Pluto, and Parsa et al. method

	Improvement over Pluto(%)	Improvement over Parsa et al. method (%)
E2200 without -O3	63.6	68.6
E2200 with -O3	61.3	64.8
Q6600 without -O3	57.8	68.8
Q6600 with -O3	52.3	62.7

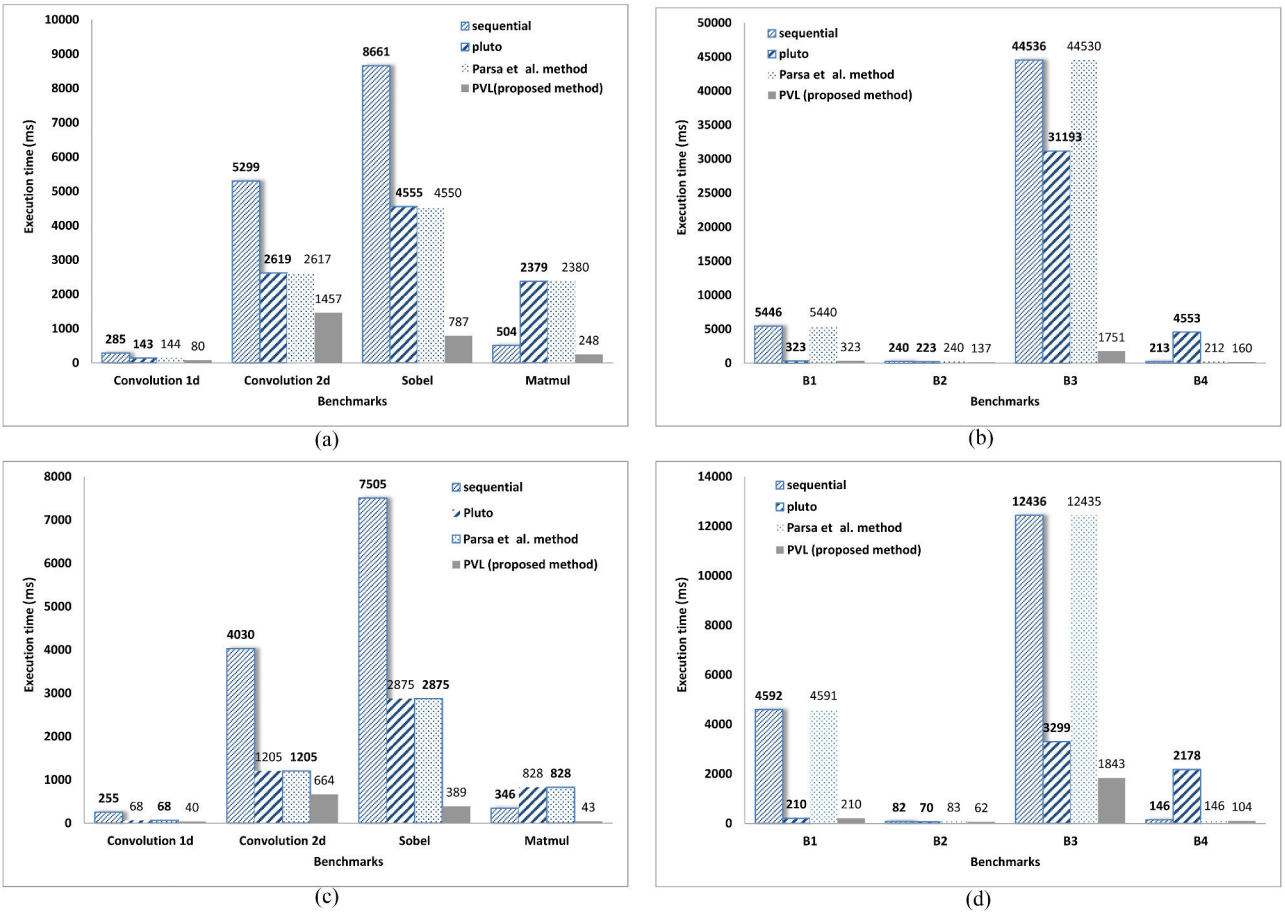


FIG. 4.4. *Execution time of the sequential program, Pluto, the Parsa et al. method, and the PVL (proposed method) on Intel Dual Core CPU E2200 @ 2.20GHz and Intel Core2 Quad CPU Q6600 @ 2.40GHz systems with -O3 flag enabled. (a) Execution of benchmarks set one on Intel Dual Core CPU E2200 @ 2.20GHz system with -O flag enalbed, (b) Execution of benchmarks set two on Intel Dual Core CPU E2200 @ 2.20GHz system with -O3 flag enabled, (c) Execution of benchmarks set one on Intel Core2 Quad CPU Q6600 @ 2.40GHz system with -O3 flag enabled, and (d) Execution of benchmarks set two on Intel Core2 Quad CPU Q6600 @ 2.40GHz system with -O3 flag enabled*

the cores, (2) vectorize dependence-free loop which has contiguous data in the memory in the short-vectors within each core, and (3) improve data locality of the loops. After applying the transformation to the loop nest, intrinsic vector codes are generated to the innermost vectorizable loop using the proposed approach. Since data dependences are unavoidable in programs and they have to be satisfied as long as we want to preserve the semantic of programs, while looking for proper transformation using proposed method, we try to satisfy data dependences in the middle loops, as much as possible.

Finally, to achieve the peak performance of different hardware architectures, concentrating on both cores (outer parallelization) and data locality is not sufficient. Nowadays, widely available vector units are critical to achieving this peak performance in modern architectures. Thus, the vectorization stage should not be left to the post-transformation stage and it should be used simultaneously beside outer parallelization and data locality improvements. Although the result of the proposed method is significantly better than the state-of-the-art compiler Pluto and also the result of Parsa et al. [22], the result can be improved further by using cache configurations of the underlying system, loop tiling, and considering temporal locality.

Acknowledgments. We would thank Nasrin Nasrabadi and Ali A. Noroozi for their precious helps.

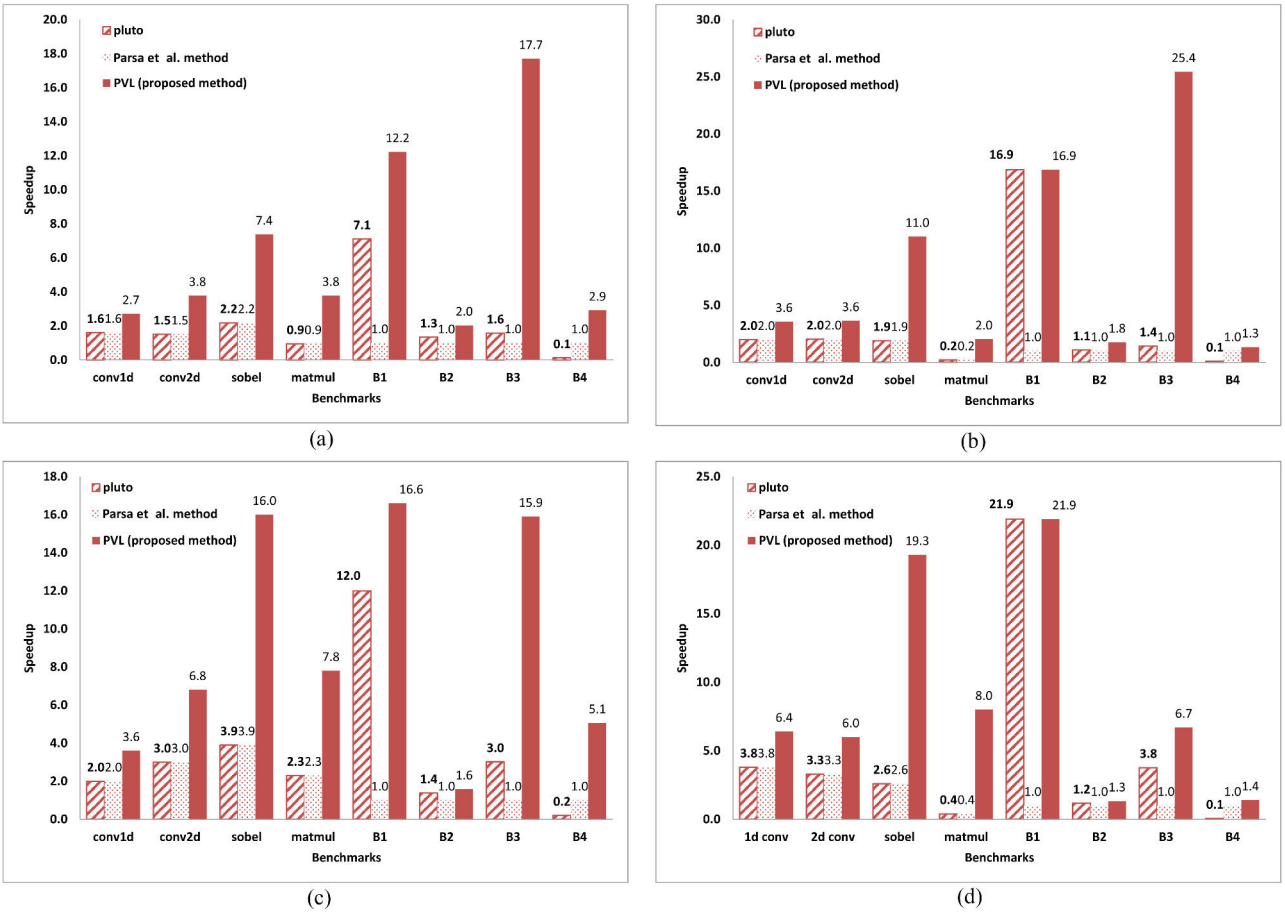


FIG. 4.5. *Speedup of Pluto, the Parsa et al. method and the PVL (proposed method) over sequential program on two core and 4 core systems. (a) Intel Dual Core CPU E2200 @ 2.20GHz system without -O3 flag, (b) Intel Dual Core CPU E2200 @ 2.20GHz system with -O3 flag enabled, (c) Intel Core2 Quad CPU Q6600 @ 2.40GHz system without -O3 flag, and (d) Intel Core2 Quad CPU Q6600 @ 2.40GHz system with -O3 flag enabled*

REFERENCES

[1] U. BONDHUGULA, *Effective automatic parallelization and locality optimization using the polyhedral model*, Dissertation, (2008), The Ohio State University.

[2] L. CHAI, Q. GAO, AND D. K. PANDA, *Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system*, In Seventh IEEE International Symposium on Cluster Computing and the Grid, (2007), pp. 471–478.

[3] *Top 500 SuperComputer Sites*, http://www.top500.org/. Accessed 20 December 2016.

[4] G. E. BLELLOCH, *Vector models for data-parallel computing*, Cambridge: MIT press, (1990).

[5] R. ALLEN AND K. KENNEDY, *Optimizing compilers for modern architectures: a dependence-based approach*, San Francisco: Morgan Kaufmann, (2001).

[6] F. FRANCHETTI, *Performance portable short vector transforms*, Dissertation, (2003), Karlsruhe Institute of Technology.

[7] P. BULIC, AND V. GUSTIN, *An extended ANSI C for processors with a multimedia extension*, International Journal of Parallel Programming, 31 (2003), pp. 107–136.

[8] S. RAMAN, K. PENTKOVSKI, AND J. KESHAVA, *Implementing streaming SIMD extensions on the Pentium III processor*, IEEE micro, 20 (2000), pp. 47–57.

[9] C. LOMONT, *Introduction to intel advanced vector extensions*, Intel White Paper, (2011), pp. 1–21.

[10] K. STOCK, T. HENRETTY, I. MURUGANDI, P. SADAYAPPAN, AND R. HARRISON, *Model-driven simd code generation for a multi-resolution tensor kernel*, In Parallel & Distributed Processing Symposium (IPDPS), IEEE, 51 (2011), pp. 1058–1067.

[11] J. M. CEBRIAN, M. JAHRE, AND L. NATVIG, *ParVec: vectorizing the PARSEC benchmark suite*, Computing Journal, 97 (2015), pp. 1–24.

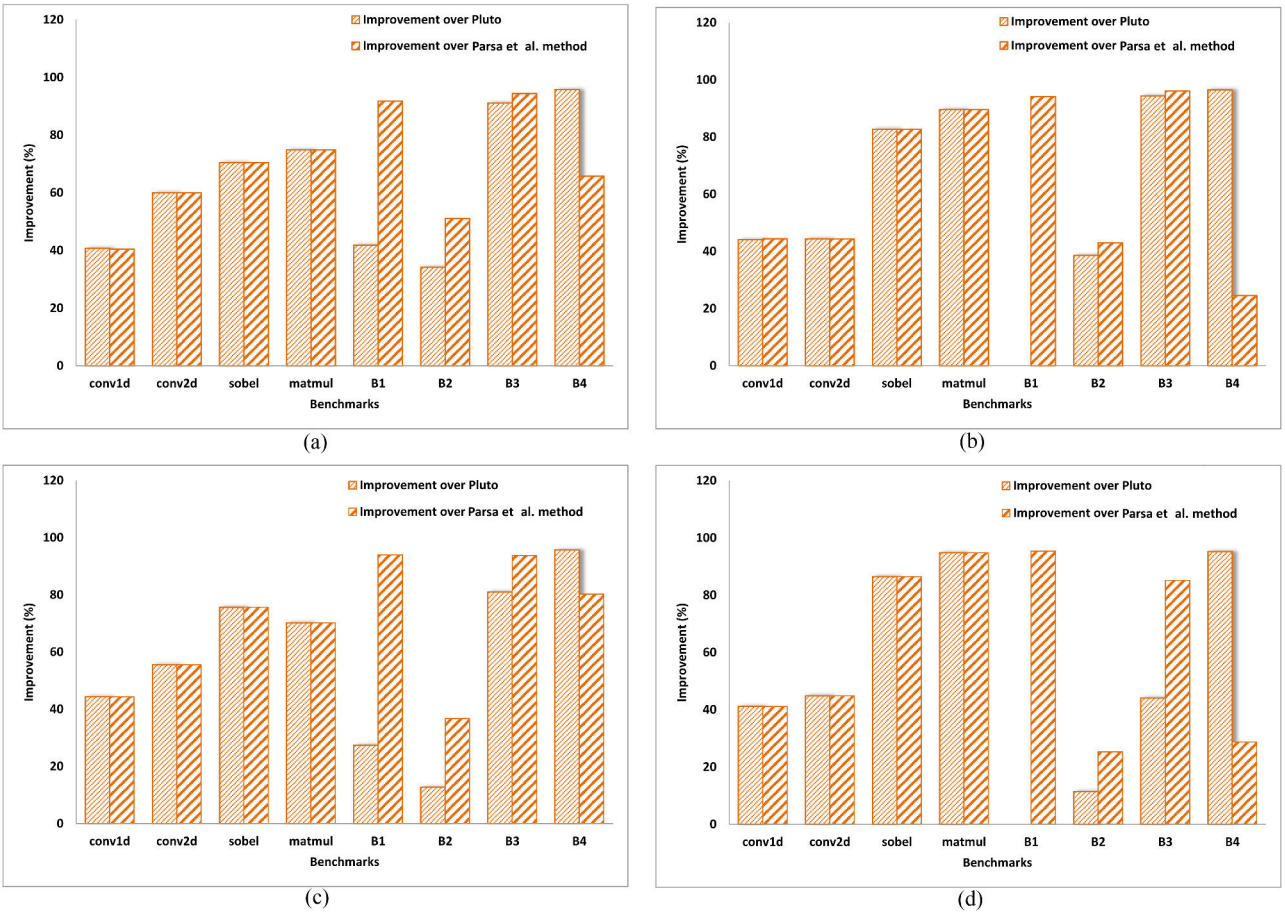


FIG. 4.6. *Improvements in execution time over Pluto, and Parsa et al. method on two core and 4 core systems. (a) Intel Dual Core CPU E2200 @ 2.20GHz system without -O3 flag, (b) Intel Dual Core CPU E2200 @ 2.20GHz system with -O3 flag enabled, (c) Intel Core2 Quad CPU Q6600 @ 2.40GHz system without -O3 flag and (d) Intel Core2 Quad CPU Q6600 @ 2.40GHz system with -O3 flag enabled*

[12] L. N. POUCHET, *When iterative optimization meets the polyhedral model: one-dimensional date*, Dissertation, (2006), University of Paris-Sud XI.
[13] M. W. BENABDERRAHMANE , L. N. POUCHET, A. COHEN, AND C. BASTOUL, *The polyhedral model is more widely applicable than you think*, In Compiler Construction, (2010), pp. 283–303.
[14] S. GIRBAL, N. VASILACHE, C. BASTOUL, A. COHEN, D. PARELLO, M. SIGLER, AND O. TEMAM, *Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies*, International Journal of Parallel Programming, 34 (2006), pp. 261–317.
[15] C. BASTOUL, *Code generation in the polyhedral model is easier than you think*, In Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques, (2004), pp. 7–16.
[16] J. XUE, *Loop tiling for parallelism*, Springer Science & Business Media, (2000).
[17] R. ALLEN AND K. KENNEDY, *Vector register allocation*, IEEE Transactions on Computers, 41 (1992), pp. 1290–1317.
[18] R. XU, S. CHANDRASEKARAN, X. TIAN, AND B. CHAPMAN, *An Analytical Model-Based Auto-tuning Framework for Locality-Aware Loop Scheduling*, In International Conference on High Performance Computing, (2016), pp. 3–20.
[19] D. UNAT, T. NGUYEN, W. FAROOQI, M. N. BASTEM, G. MICHELOGIANNAKIS, AND J. SHALF, *Tida: High-level programming abstractions for data locality management*, In International Conference on High Performance Computing, (2016), pp. 116–135.
[20] S. NACI, *Optimizing Inter-Nest Data Locality Using Loop Splitting and Reordering*, In Parallel and Distributed Processing Symposium, (2007), pp. 1–8.
[21] O. OZTURK, *Data locality and parallelism optimization using a constraint-based approach*, Journal of Parallel and Distributed Computing, 71 (2011), pp. 280–287.
[22] S. PARSA, M. HAMZEI, *Locality-conscious nested-loops parallelization*, ETRI Journal, 36 (2014), pp. 124–133.

[23] W. Bielecki, and M. Palkowski, *Loop Nest Tiling for Image Processing and Communication Applications*, In International Multi-Conference on Advanced Computer Systems, (2016), pp. 305–314.
[24] K. Qiu, Y. Ni, W. Zhang, J. Wang, X. Wu, C. J. Xue, and T. Li, *An adaptive Non-Uniform Loop Tiling for DMA-based bulk data transfers on many-core processor*, In Computer Design (ICCD), (2016), pp. 9–16.
[25] S. Parsa, S. Lotfi, *A new genetic algorithm for loop tiling*, The Journal of Supercomputing, 37 (2006), pp. 249–269.
[26] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, *Effective automatic parallelization of stencil computations*, In ACM Sigplan Notices, 42 (2007), pp. 235–244.
[27] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, *Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model*, In Compiler Construction, (2008), pp. 132–146.
[28] V. Bandishti, I. Pananilath, and U. Bondhugula, *Tiling stencil computations to maximize parallelism*, In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, (2012), pp. 40–51.
[29] M. Baskaran, B. Pradelle, B. Meister, A. Konstantinidis, and R. Lethin, *Automatic code generation and data management for an asynchronous task-based runtime*, In Proceedings of the 5th Workshop on Extreme-Scale Programming Tools, (2016), pp. 34–41.
[30] J. Cong, M. Huang, P. Pan, Y. Wang, and P Zhang, *Source-to-source optimization for HLS*, In FPGAs for Software Programmers, (2016), pp. 137–163.
[31] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, and T. F. Ngai, *Data layout transformation for enhancing data locality on nuca chip multiprocessors*, In Parallel Architectures and Compilation Techniques, (2009), pp. 348–357.
[32] B. Jang, P. Mistry, D. Schaa, R. Dominguez, and D. Kaeli, *Data transformations enabling loop vectorization on multithreaded data parallel architectures*, In ACM Sigplan Notices 45 (2010), pp. 353–354.
[33] A. Krechel, H. J. Plum, and K. Stuben, *Parallelization and vectorization aspects of the solution of tridiagonal linear systems*, Parallel Computing, 14 (1990), pp. 31–49.
[34] C. Aykanat, F. Ozguner, and D. S. Scott, *Vectorization and parallelization of the conjugate gradient algorithm on hypercube-connected vector processors*, Microprocessing and Microprogramming, 29 (1990), pp. 67–82.
[35] J. Crawford, Z. Eldredge, and G. A. Parker, *Using vectorization and parallelization to improve the application of the APH hamiltonian in reactive scattering*, In Computational Science and Its Applications ICCSA, Springer Berlin Heidelberg, (2013), pp. 16–30.
[36] L. N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache, *Loop transformations: convexity, pruning and optimization*, In ACM SIGPLAN Notices, 46 (2011), pp. 549–562.
[37] P. Feautrier, *Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time*, International journal of parallel programming, 21 (1992), pp. 389–420.
[38] R. Penrose, *A generalized inverse for matrices*, In Mathematical proceedings of the Cambridge philosophical society, 51 (1955), pp. 406–413.
[39] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G. A. Silber, and N. Vasilache, *GRAPHITE: Loop optimizations based on the polyhedral model for GCC*, 4th GCC Developer's Summit, (2006), pp. 179–198.
[40] *PLUTO - An automatic parallelizer and locality optimizer for affine loop nests*, http://pluto-compiler.sourceforge.net/

Edited by: Dana Petcu
Received: December 2, 2016
Accepted: January 27, 2017

DOI 10.12694/scpe.v18i1.1235
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 18, Number 1, pp. 91–103. http://www.scpe.org

ENABLING AUTONOMIC COMPUTING SUPPORT FOR THE JADE AGENT PLATFORM

ALI FARAHANI[*], ESLAM NAZEMI[†], GIACOMO CABRI[‡], AND NICOLA CAPODIECI[§]

Abstract. Engineering complex distributed system is a real challenge documented in recent literature. Novel paradigms such as Autonomic Computing (AC) approaches appear to be the fittest engineering model in order to face performance instabilities of systems inserted in open and non-deterministic environments. To this purpose, the availability of appropriate development environments will facilitate the design of such systems. Standard agent development platforms represent a good starting point, but they generally lack of rigorous ways to define central AC-related concepts such as the prominent role of feedback loops and knowledge integration in decision making processes: we therefore believe that Agent Oriented Software Engineering (AOSE) can be substantially enriched by taking into account such concepts.

In this paper, a novel extension of the well-known JADE agent development environment is discussed. This extension enhances JADE in order to address the engineering process with Autonomic Computing support. It is called "Autonomic Computing Enabled JADE" or shortly ACE-JADE. The behavioral model of ACE-JADE will be thoroughly described in the context of a case study (NASA ANTS project).

Key words: JADE, Autonomic Computing, Multi-agent Systems, Development Environment

AMS subject classifications. 68T42

1. Introduction. The complexity of information systems has grown dramatically in recent years. This complexity, which is the result of several factors like advances in hardware and infrastructure technology, the growth of Internet networks, etc., has inspired Autonomic Computing (AC) approaches, which enable software components to go through runtime adaptation processes in order to react to changes of their execution contexts [24, 22]. It is important to design software able to promptly react to these changes as unpredictable dynamic environments pose threats to the stability and performance of the designed system. In order to fully exploit the benefits of the mechanisms behind these adaptation processes, different architectures were introduced [20, 36] and they all derive from the first work that introduced the concept of Autonomic Computing, as described by IBM [22].

Autonomic Computing builds upon the concept of autonomous software entity and focuses on the interactions between the decisional process within the software entity and its surrounding environment. The general idea is to establish a feedback loop able to provide additional elements to be used in the decisional process of each software entity and a well-known architectural design for these feedback controlled mechanisms is the *MAPE loop*. This loop consists of four phases: Monitoring, Analyzing, Planning and Executing [24, 23]. During the *Monitor* phase, the system component collects and correlates information from the environment; this data collection phase is enabled through specific sensorial capabilities featured by the component. In the *Analyze* phase, the component will analyze the variables observed during the Monitor phase, so to determine the type and the magnitude of the needed reactions: this is instrumental to correctly respond to threatening environmental changes. Passing the result of analyze phase into the *Plan* phase, the component will select and conclude into more specific sub-set of actions to be performed in the environment to achieve a desired state. And at last, in the *Execute* phase, the component will enact the selected actions through effectors. [12].

As mentioned in [33], feedback loops foster self-* properties within the software systems. These kinds of capabilities are related to the awareness of a system about itself and its environment, hence the ability of a system to autonomously react to environmental changes. There is no general agreement over these concepts and the related terminology but there are main four properties (self-healing, self-protecting, self-optimizing and self-configuring) usually categorized as self-* properties or simply summarized under the term self-managing [24].

[*]Computer Science and Engineering Department, Shahid Beheshti University Tehran, Iran (a_farahani@sbu.ac.ir)
[†]Computer Science and Engineering Department, Shahid Beheshti University Tehran, Iran (nazemi@sbu.ac.ir)
[‡]Department of Physics, Informatics and Mathematics, Università di Modena e Reggio Emilia, Modena, Italy (giacomo.cabri@unimore.it)
[§]Department of Physics, Informatics and Mathematics, Università di Modena e Reggio Emilia, Modena, Italy (nicola.capodieci@unimore.it)

In this paper, we present how AC topics can seamlessly integrate within distributed system engineering methodologies such as Agent Oriented Software Engineering (AOSE). Distributed system is considered as another solution to large scale and complex problems, thanks to their high degree of scalability. Bringing more resources together to satisfy needs for more coverage and computation is a solution, even if managing them leads into another kind of complexity [37]. Having autonomous agents which can deal with changes by their own is a way which can heal the complexity. Appropriate design and simulation frameworks for distributed autonomous system are therefore valuable instruments for the system designer.

Multi-Agent Systems (MAS) is one of the main branches of distributed systems, enabling autonomous components to carry out tasks in a decentralized way, supporting flexibility and scalability. MAS architectures described in literature vary from abstract design methodologies to detailed engineering frameworks or platforms. FIPA (Foundation of Intelligent Physical Agents) proposes a complete reference architecture for MAS [28]. JADE (Java Agent Development Environment) is a FIPA compliance agent development environments [5, 4], which supports the development of multi-agent systems. Bringing AC into the JADE can enable the self-* properties in MAS architectures.

The work presented in this paper aims at extending JADE to bring support for Autonomic Computing in the JADE development environment. This idea has been preliminary discussed in [18] and in this paper we provide much deeper analyses and discussions regarding our novel extension. Section 2 presents the background and related work. Main elements of proposed architecture are discussed in Section 3. Section 4 will be about a case study and implementing the presented extension in a scenario. Conclusion and future work are discussed in Section 5.

2. Background and Related Work. In this section, background information about self-adaptation (and autonomic computing in general) and related concepts are briefly presented, along with JADE as a development environment. The previous researches about providing JADE extensions are also discussed.

2.1. Autonomic Computing in software development. In [17, 25] researchers tried to use component-based approach to help the development of autonomic software. In [17] an infrastructure named AUTONOMIA is presented, which foster engineering and deployment of autonomous applications. In [25] a framework for supporting development of a component-based application (self-managed application) is presented. These solutions represent proof-of-concepts in which we start to develop our extension.

In [7] a toolkit is presented (*ABLE*) for building multi-agent autonomic systems. This toolkit provides a lightweight Java agent framework with a set of beans able to support self-* properties development. This toolkit focuses more on the development of agents' self-* tasks rather than on characterizing the capabilities of multi-agent system. Also, in [14], a framework based on component-based viewpoint is introduced. A conceptual framework for designing autonomic components starting from the artificial immune system paradigm have been proposed in [11]. It covers the self-expression aspect of system based on inspiration from the natural immune system. Researches for bringing autonomic computing in system design and development are not bounded to only high level concepts like framework. For instance a research provides a language (SCEL, Software Component Ensemble Language) to be used in any framework for formally and rigorously describe coordination patterns for autonomic agents and their interactions [10].

The novelty of our work consists in providing an actual implementation in the form of a JADE extension of the topics explored as theoretical subjects by the cited articles described in this section. During the development of this extension, we realized that many important engineering artifacts were missing, thus we integrated this design effort with our own ideas.

2.2. JADE and its extensions. JADE (Java Agent Development Environment) is a development environment which facilitates the process of multi-agent system development, providing a platform that supports the agent design and execution. Multi-agent systems and especially agents themselves have many reference architectures. JADE is compliant with FIPA agent architecture [5].

Adding **AC** (Autonomic Computing) ability to the agent development environment has been subject to previous research. For instance, in [15] an architecture-based extension for supporting feedback loops in agents has been discussed. In particular, authors [15] discussed the possibility of adding architectural patterns so to have feedback loops in MAS.

Most of the JADE extensions that are related to AC concept are built upon the *behaviour* concept, which can be considered as one of the JADE primary elements. In [8] JADE agent behaviour is extended in order to support *behaviour tree* and *flexible behaviour* in JADE. Researches in [1] and [21] bring *organization* and *role* concepts into JADE, but there is still no mention regarding control loops. Control loops have been exploited in the form of coordination patterns into the software development process to have an autonomic computing enabled development kit [30]. The idea of having a role-based design in multi-agent systems alongside coordination mechanisms is able to provide a methodology which its output can be used to generate JADE agent classes.

Other examples of JADE extension in order to provide more formal ways to model adaptivity can be found in [27], [9] and [26]. Also The idea about using JADE and MATLAB (Simulink) have been addressed in [32] as a JADE extension called MACSimJX. This extension is about creating multi-agent control systems. Using Simulink as a way to describe agents, MACSimJX enables in JADE to implement a multi-agent control system. Using this idea in micro-grid environment is discussed in [31].

The common idea behind these extensions is to bring something new into JADE in order to add new features or to exploit it in new application fields. Enhancing JADE with ACE-JADE will provide the ability to facilitate the process of implementing AC enabled software in MAS environment. In [19] a preliminary idea about bringing Autonomic Computing into JADE has been discussed and future challenges addressed. In this paper, the idea of JADE extension will be discussed with more detail.

2.3. Agent Development Platforms. JADE is not the only FIPA-compliant agent platform. There are some researches about self-* system development tools and frameworks (like [16] as a framework for multi-agent systems development in IoT environment) which mostly are compliant with FIPA standards. *Grasshopper* is presented in [2] as an agent development platform based on OMG MASIF and FIPA. A work-flow management system alongside an intelligent system business is supported in [2]. *Grasshopper*, however, appears to be not currently maintained, hence, in order to develop ACE-JADE, we decided to stick with JADE.

There are several development aspects related to ACE-JADE. In [6] researchers mention that knowledge management is one of the critical parts of the proposed development environment. One of the targeted ways for addressing this challenge is rule-based knowledge management. In distributed environment conflicts between each agent's knowledge and reasoning could be a problem. In [29] this challenge has been addressed. Also, some other researches are taking place without tying themselves to FIPA standard. Jackal [13] is one of these solutions. It presents a Java-based tool for development of agents. These kinds of solutions have a downside of not getting the same level of attention and support as the other frameworks that comply with known standards.

The idea of having *autonomic computing* support in the JADE emerged after realizing that all of the previously discussed existing research does not involve the implementation of tools and platform that can be used by the system engineers. The popularity of JADE, due to its open source Java implementation on the top of the FIPA standard, is the ideal starting point for such effort, that will build upon the findings of all the previously cited papers.

3. AC Extension of JADE (ACE-JADE). A framework for MAS can be addressed from two aspects; namely, (1) a *structural* aspect and (2) a *behavioral* aspect. Our MAS extension, therefore, will be discussed considering both these aspects, i.e. extending the JADE java classes related to structural part and also extending the behavior related classes.

More specifically, our AC enabled JADE extension is articulated in five points:

1. *Providing adaptive agents.* In MAS the central component is the *agent*. In order to have an autonomic system, we exploit the concept of agent in terms of generic autonomous software component.
2. *Implementing different feedback loops by extending the class* `Behaviour`. Like the structural concept of (`Agent`), there must be an extension of behavioural concepts in JADE (`Behaviour`) to enable the response to environmental changes.
3. *Adding Knowledge support in the adaptive agent.* Data and knowledge is a vital concept in AC, therefore it will be present in our JADE extension.
4. *Support for* `Sensors` *and* `Effectors`. Sensors and effectors are AC related concepts that deal with the ability of an agent/software component to obtain information regarding the surrounding environment and act on the same environment. Our extension will therefore model environment interactions through Sensors and Effectors classes.

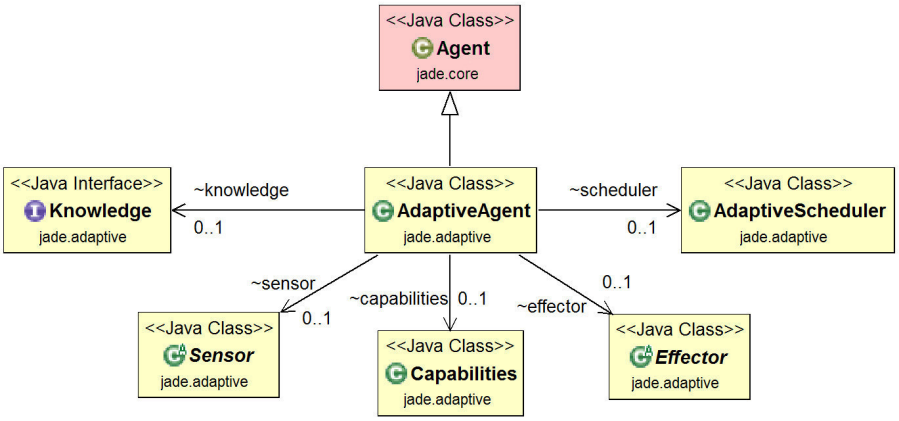


FIG. 3.1. *AdaptiveAgent class diagram.*

5. *Extending* `Message` *to support internal messages among different adaptive agents.* JADE original `Message` class cannot fulfill the need for information sharing in the AC, hence we had to provide an added layer of complexity to the original Message related classes. More specifically, we later detail how Sensors and Effectors will be implemented with a message passing interface.

The mentioned points are vital for making ACE-JADE. In the following sections, these five different aspects will be discussed. Each aspect will be discussed by a *class diagram* and if the behavioural aspect is involved, it will be analyzed though a *sequence diagram*. The class diagrams will explain parts that have been extended in the JADE original structure and the sequence diagram will show the mechanisms that our novel extension will provide in order to obtain the autonomic computing related features on top of JADE.

Starting from Fig. 3.1, all the subsequent class diagrams will use the following notation: in red, we indicate the original classes already present in the JADE standard package, whereas yellow artifacts are actually related to classes and interfaces provided by our extension.

3.1. Providing `AdaptiveAgent`. The `AdaptiveAgent` class (our implementation of autonomic software component) keeps references to `Sensor` and `Effector` instances that characterize the components (see Fig. 3.1). These agents are also composed by instances of `Capabilities` class, which define the list of acceptable *parameters* and list of feasible *actions* for such agents.

In addition to what can be seen in Fig. 3.1, we specify that for each `AdaptiveAgent` component, a set of *behaviours* for adaptive tasks is provided (by means of the class `AdaptiveBehaviourSet`). It consists of a set of `AdaptiveBehaviours` with a list of task priorities (`BehaviourOrder`). Also, there is another proposed class, `AdaptiveScheduler`, in which resides the implementation of how different behaviours are arbitrated in terms of ordering. This `AdaptiveScheduler` extends the `Scheduler` class of JADE. While there is not much to explain with regards to the scheduler, the behaviours are the most important aspect of our work and they are detailed in the next section.

3.2. Extending Behaviour. The most important aspect of AC is responding to changes in the environment in an autonomous and adaptive way. The reaction towards the dynamics of the environment is taken care by the behavioural aspect of ACE-JADE.

Autonomic computing enabled through MAPE feedback loop is composed of four sequential actions (Monitor, Analyze, Plan and Execute) and such actions involve the use of sensors and effectors. Each of these phases can be composed by a plurality of steps. As mentioned in [23], we identify different levels of adaptation according to whether all of these four phases are present: a distributed system with limited adaptation capabilities, for instance, might feature a simplified control loop, in which just the Monitor and Execute phases are present. By defining a control loop as a subset of combinations of the MAPE phases, ACE-JADE is therefore able to model different adaptation capabilities.

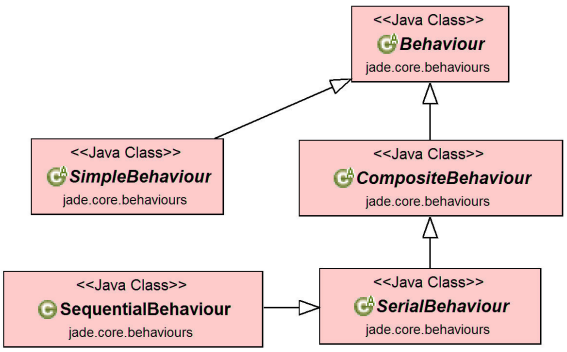


FIG. 3.2. *SerialBehaviour, SequentialBehaviour and CompositeBehaviour relations*

In ACE-JADE, MAPE control loops as a whole, their phases and each step for each phase is a JADE behaviour. From the JADE original `behaviour` package, we infer the class diagram in Fig. 3.2. These are the classes from which our custom control loops will be implemented, but also phases and single steps for each constituent part of the feedback loop.

In order to implement different control loops with each having a different level of adaptation, we have five different classes and they all extends the JADE original `CompositeBehaviour` class:

- class `AutonomicBehaviour`: which allows the implementation of all the MAPE four phases without focusing on a complex Knowledge representation logic.
- class `AdaptiveLoopBehaviour`: which also allows the implementation of all the MAPE phases and forces the user to implement specific Knowledge representation logic.
- class `PredictiveLoopBehaviour`: which allows the definition of simpler control loops, that just consider Monitoring, Analyze and Execution phase.
- class `ManagedLoopBehaviour`: same as above, but focusing on Monitor, Execute and Knowledge representation.
- class `BasicLoopBehaviour`: represents the simplest kind of feedback loop, as just Monitor and Execute phase can be exploited from this class.

In other words, according to the level of desired self-adaptation, the user will select the appropriate class. The selected class relates to specific admissible MAPE phases (detailed in the previous class list). Each of these different classes are composed of one instance of each admissible phase state. A phase state is one of `MonitorState`, `AnalyzeState`, `PlanState` and `ExecuteState` (see Fig. 3.3). Phase state classes are derived from the JADE original class `SequentialBehaviour`.

Each phase state class is therefore composed of 1 to N corresponding steps (`Monitor`, `Analyze`, `Plan` and `Execute`). A step class is derived from the `SimpleBehaviour` class out of the JADE original `behaviour` package.

This particular class hierarchy allows us not only to tune the self-adaptive capability of our system, but also allows for implementing logic for each of the MAPE phase within an arbitrary complexity given by the composition of several steps. This is visible in Fig. 3.4, in which, for brevity, just one of the adaptive level is shown (`AutonomicLoopBehaviour`).

3.3. Introducing Knowledge. Another important requirement to introduce AC in JADE is to enable agents to manage knowledge and information, so an appropriate implementation of these concepts has been introduced in our extension of JADE. An interface named `Knowledge` is introduced, which mainly provides two methods: `toRule` and `fromRule`. The first method defines a behaviour in response to a predicate; such resulting behaviour will be inserted in a knowledge repository. Such knowledge repository can be queried with `fromRule` method.

The `Knowledge` interface is implemented by a specific class for each message type (between each MAPE step). `SensorKnowledge` and `EffectorKnowledge` are implemented to support the tasks of `Sensor` and `Effector`. These classes, with the help of `Capabilities`, will define which *parameters* should be sensed and which *actions* can be performed.

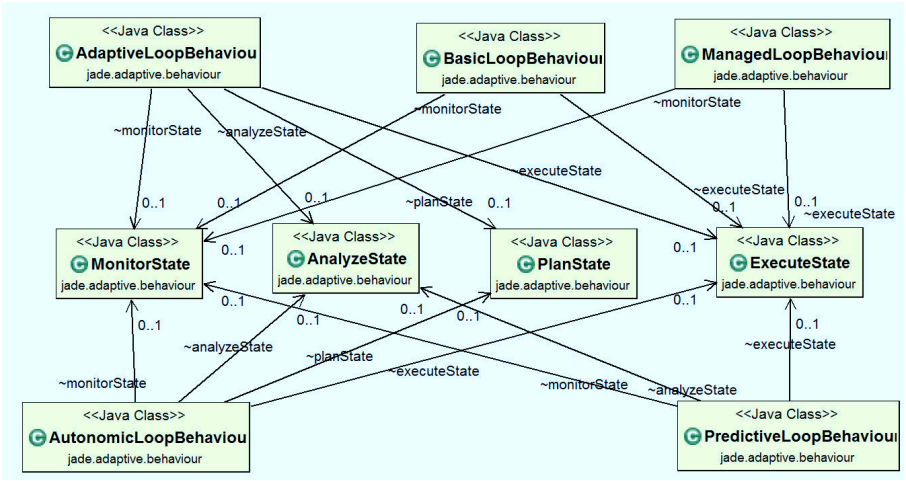


FIG. 3.3. *Behaviour class diagram.*

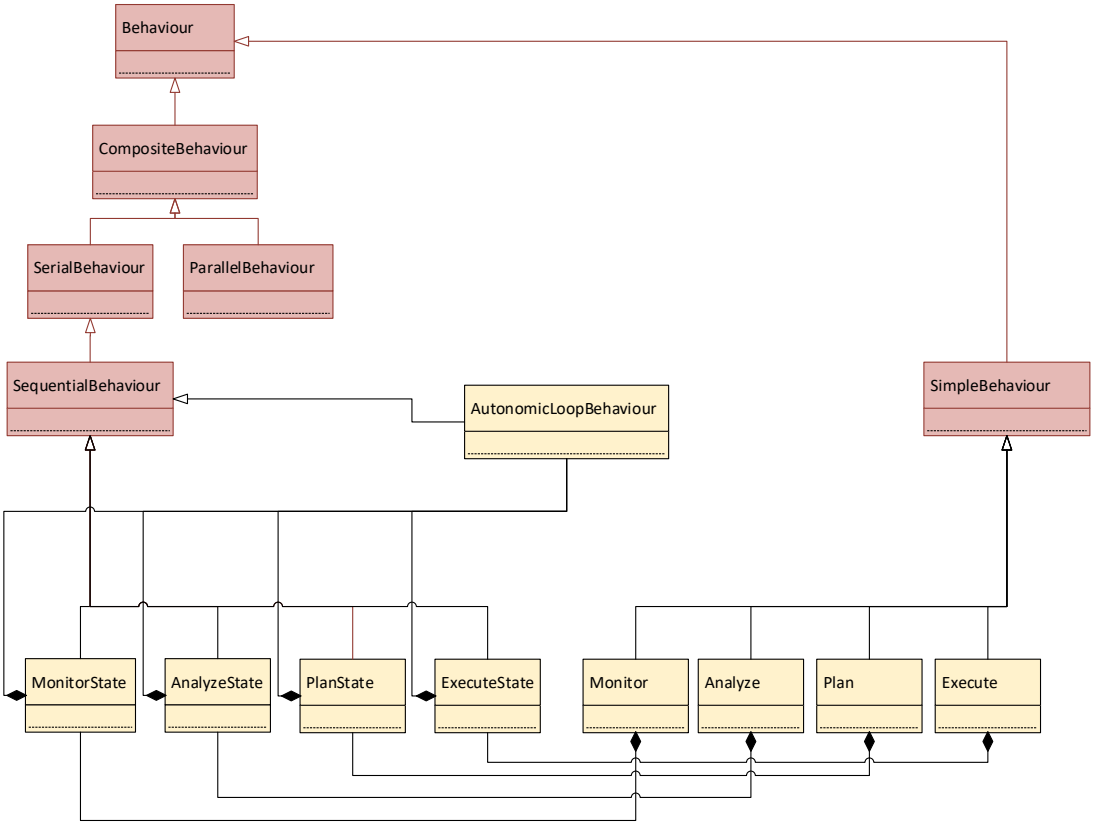


FIG. 3.4. *Holistic view of Class diagrams for Autonomic Loop Behaviour*

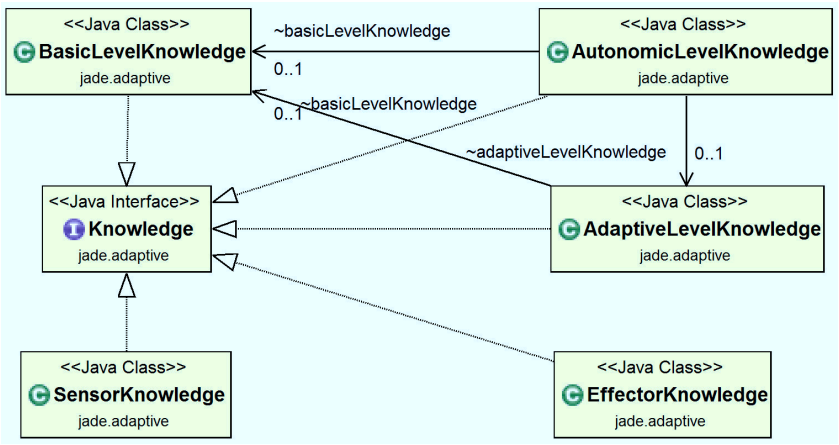


FIG. 3.5. *Knowledge-related class diagram.*

In [23] three levels of knowledge about the environment and reaction to the environment are presented. Each level has its own abstraction and each of the five different kinds of control loop: `BasicLevelKnowledge`, `AdaptiveLevelKnowledge` and `AutonomicLevelKnowledge`. The class diagram of the Knowledge-related class can be seen in Fig. 3.5. According to the complexity of the knowledge repository, the user can select which class to extend.

3.4. Support for Sensor and Effector. `AdaptiveAgent`s will have to deal with `Sensor`s and `Effector`s. These are classes for implementing the logic in which the environment can be sensed and modified according to the behavioural logic expressed by the implemented control loop. Such sensor might be further specified with capabilities. For addressing the capabilities, a class named `Capabilities` that shows the incoming known parameters and also the outgoing accessible parameters and their transformation function is introduced. The transformation operation takes place whenever sensed data needs to be translated in a different format, as different component of our system might implement different data representation schemes. This transformation is performed by two JADE original classes: `IncomingEncodingFilter` and `OutgoingEncodingFilter`, which will be discussed in next subsection. `Effector` is an extension of `OutgoingEncodingFilter` and translates in a different representation scheme the modification to be enacted to the environment.

The agent knows about the environment in which it is situated. The environment is represented by `Environment`, which is an interface that enables sensor and effector to know where they should get the data from and what they should effect. `AdaptiveAgent.environment` can be accessed through the `environment` variable in `Sensor` and `Effector`.

The class diagram of the `Sensor` and `Effector` class can be seen in Fig. 3.6.

3.5. Support for Internal Messaging. Passing information and data within the system needs an appropriate *messaging platform*. JADE has its own messaging platform which is extended by ACE-JADE in order to help in order to deal with sensors and actuators of each component of the system.

- The `Sensor` ACE-JADE class will use the original `IncomingEncodingFilter` class for translating the data based on the data model which is presented by our added concept of `SensorKnowledge`. An instance of `IncomingEncodingFilter` should be created by the developer based on his/her needs and passed to the `SensorKnowledge`'s instance. The developer could translate the monitored parameters from environment into a set of control signals. Filters are used to process only the relevant subsets of information sensed from the environment.
- As far as effectors are concerned, filters are implemented through the original class `OutgoingEncoding-Filter`. This class is therefore able to filter outgoing commands related to the encoding of ACL messages [3]. Filtering rules can be further tuned with the ACE-JADE specific class `EffectorKnowledge`.

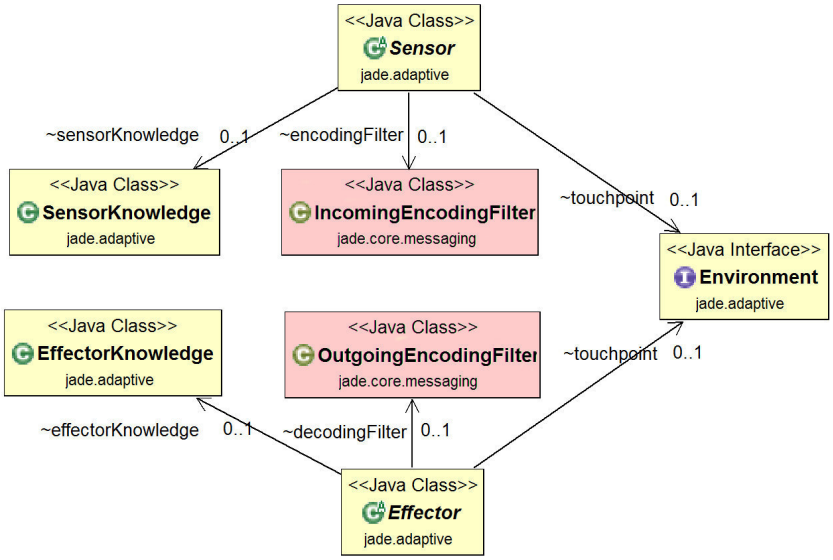


FIG. 3.6. *Sensor and Effector class diagram.*

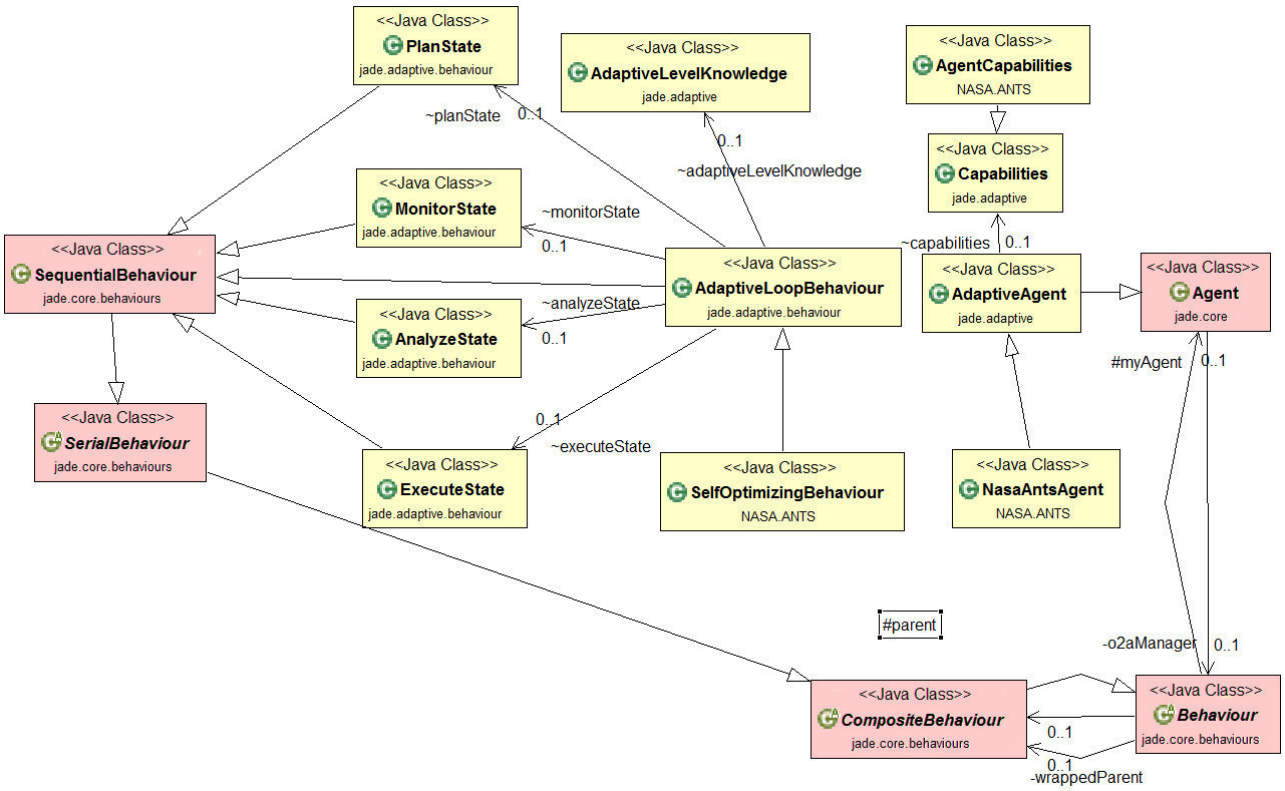


FIG. 3.7. *Self-optimizing scenario implementation class diagram*

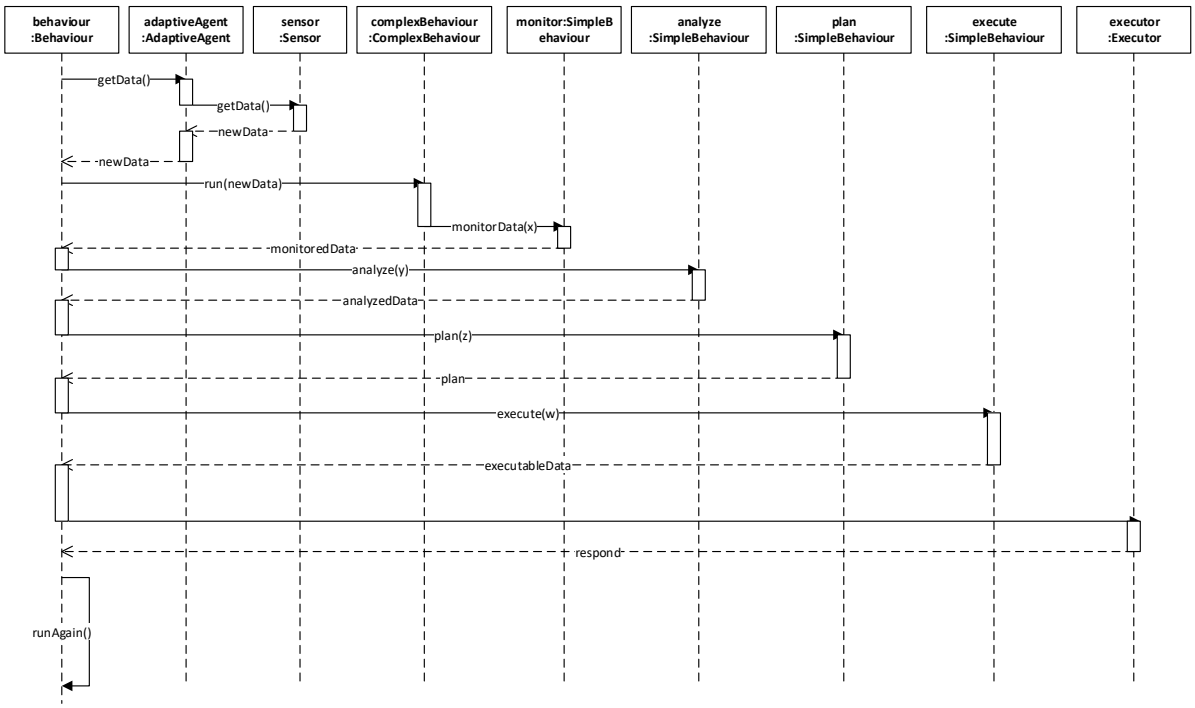


FIG. 3.8. *Sequence Diagram of a simple MAPE loop*

3.6. Behavioural Analysis. While in the previous sections we addressed the *structural* aspect of our extension, in this section we use sequence diagrams in order to better explain the *behavioural* aspects of ACE-JADE.

In Fig. 3.8, a sequence diagram of a control flow for a simple MAPE loop is reported. The `behaviour` is an active thread which, on a defined interval, starts to see if there is any new data available for the process as obtained by a `sensor`. For this purpose, the component-agent sends a message to `adaptiveAgent` and it passes the control to `sensor` in order to analyze the new data. Afterward, new data goes back to the `adaptiveAgent` and after that to `complexBehaviour`. For analyzing the data with MAPE loop, `complexBehaviour` passes the control through all the four steps of MAPE loop on after another. At last in this case, *executablePlan* as a result of `plan()` is executed with the help of `execute()`.

4. Implementing a Case Study. For better understanding the proposed AC extension to JADE, a simple scenario about managing failure (self-healing property) in *ruler* components of the **NASA ANTS** project is implemented. NASA ANTS is a new class mission from NASA [35]. NASA Autonomous Nano Technology Swarm is a milestone for autonomic computing concept and because of its specification, it represents a useful case study for autonomic computing distributed environment to be modeled with ACE-JADE. In the NASA ANTS project thousands of tiny space-craft weighting less than 1.5 kg will work cooperatively to explore the asteroid belt [35, 34]. In the project, three different kinds of space-crafts are working together to carry out the project mission:

- *Worker:* the largest number of space-crafts are of worker kind (80%). They carry different instruments for gathering data.
- *Ruler:* this kind of space-craft defines and update the rule of each space-craft in the team formation.
- *Messenger:* this kind of space-craft enables communication between workers, rules and the earth station.

All of these space-crafts have a degree of autonomy and adaptation to perform necessary tasks.

4.1. Scenario. We chose a scenario that can be addressed by designing an internal MAPE loop for each Messenger space-craft; each Messenger is therefore implmented by an instance of our `AdaptiveAgent` class, as the concrete example of an autonomic component. In this scenario, both *messengers* and *rulers* can perform a change in their states: the former by changing their location, the latter by updating their information about new space-crafts. Rulers therefore are in charge of specifying to the messenger, which messages have to be passed to the workers. The goal in this scenario is to enable *self-optimization* and *self-protection* for all these space-crafts.

In order to foster self-optimization and self-protection, a *messenger* is enhanced with two MAPE control loop. A first loop is defined to improve its positioning over time (as it can communicate within fixed ranges), and a second loop is in charge of detecting the status of local established connections with the other space-crafts, and react by changing the messenger direction of movement.

As far as the first MAPE loop is concerned, a messenger space-craft finds out that by changing to a given location it can reach more space-crafts. It can do this by measuring the density of the space-crafts within its range. This will represent the ability to *self-optimize*.

As far as the second loop is concerned, this loop which monitors the messenger special connections (such as the one with the ruler agents) and when it detects the probability of losing these connections it reacts by changing its movement direction in order not to get too far away from a ruler. This will represent the ability to *self-protect*.

Self-optimization and self-protection are tied together in this scenario: the two MAPE loops are designed in order to balance the goal of varying the position of the messenger so to have reach as many workers as possible, but at the same time its average movement direction shall not bring the messenger itself too far from a ruler, as this situation will threaten the ability of the system of propagating updated messages.

4.2. Implementation with ACE-JADE. We have implemented some new classes based on scenario issues. The list of the newly extended classes for this scenario is:

- `NasaAntsAgent:` specialization of the `Agent` which is implemented and used in the NASA ANTS project. Also, the `Capabilities` class is used to make differences between the three different kinds of the agents in NASA ANTS mission. The texttttNasaAntsAgent is therefore the messenger agent as introduced in the previous section.
- `SelfOptimizingBehaviour:` the extra structure which is needed for an algorithm to deal with environment and performing tasks for achieving *self-optimizing* is implemented in this class (first MAPE loop).
- `SelfProtectingBehaviour:` another structure which contains the *self-protecting* algorithm to maintain important connections of the *messenger* (second MAPE loop).

Also, besides the previous classes which have defined and instantiated in the system, the following classes are used in the case study:

- `AdaptiveLevelKnowledge`
- `Capabilities`
- `MonitorState`
- `AnalyzeState`
- `PlanState`
- `ExecuteState`
- `Sensor`
- `Effector`

Based on the location where the `NasaAntsAgent`'s instance is, it counts its nearby neighbors (within its range) in its monitoring phase and if the measured density is less than the defined density in the agent's *knowledge* (within the *analyze knowledge*, based on the planning *knowledge*, a random path will be picked and applied within the *execute* phase. It will cause a change in the location of the agent. This cycle will continue until the value of monitored neighbors exceeds the predefined number in analyze knowledge.

Simultaneously with the previous loop, another instance of a composite behaviour is running. This new object is instantiated from an implementation which supports agent's connectivity to important nodes. It will

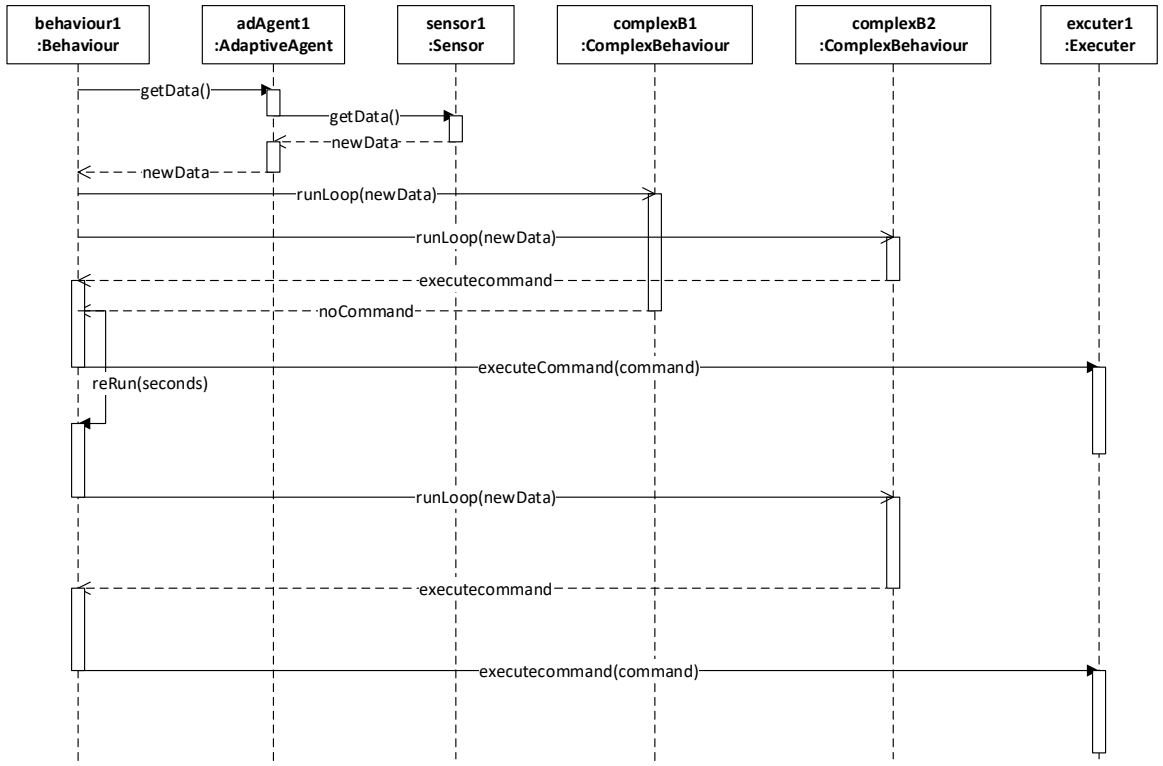


FIG. 4.1. *Sequence diagram of the implemented case study*

cover the self-protecting property and will go after being sure about continue connections which have been flagged as important connections. After the creation of the loop (by means of the appropriate *behaviour*) every defined period the behavior starts and measure the distance of the agent with flagged agents to be sure that the distance is not getting near to the agent's transmission rate. If so, it will start changing the speed of the agent in order to be sure that the other agents are not going to be farther away.

For better understanding the implementation, the sequence diagram of a system run is provided (see Fig. 4.1). The descriptions of objects are reported in the figure. In this sequence diagram two feedback loops are described (*self-optimizing* and *self-protecting*, as mentioned previously). In the first round, there is a run of the loop of `complexB1`, which is about the self-optimization. In this run, the result will be a set of commands (decisions about changes in the location) to improve the number of covered nodes. While this loop is running, another loop (`complexB2`) is called for running. The `complexB2` is about the self-protection. Because there was no change in the location, there is no result for that loop. But as it is reported in the diagram, another run is scheduled for this loop for some time later. After implementing the changes of the `complexB1` with the help of `executor1`, another round of `complexB2` running is started. In this round, because of changes in the location, the result is a correction of the location in order to be sure about being connected to important nodes. Afterward, this decision is applied with the help of `executor1`.

Meanwhile there are not similar researches which can be examined to compare our presented extension, we can compare the time and effort (number of classes) needed for implementing a system by standalone JADE with the proposed extension. This can give a overall proof of applicability and usefulness about the presented extension.

A NASA ANTS case study was implemented with ACE-JADE and also with standalone JADE. The time needed to implement the case-study with JADE was around 90 hours with 1 developer familiar with JADE. The same case study was prepared in around 75 hours with 1 developer which was familiar with ACE-JADE. The number of classes for JADE implementation was 16 classes and for ACE-JADE was 8. A point which should be considered is that ACE-JADE has more than 20 pre-implemented (or at least semi-implemented) classes with respect to JADE. Researchers think that with expanding the problems this improvement and time saving will have a bigger gap with developing a problem which needs autonomic computing as a necessary aspect with bare JADE.

5. Conclusion and Future Work. Having a multi-agent development environment that provides the support for autonomic computing features makes ease the process of creating adaptive multi-agent system. This support can shorten the development process and also can minimize the development expenses. Also, like any other development environment, using autonomic enabled development environment will increase the quality of the developed system.

An extension of JADE for having adaptive agents has been proposed in this paper. For this extension, changes concern the introduction of `AdaptiveAgent`, `Behaviour`, `Message` classes; also the `Sensor` and `Effector` classes are introduced as auxiliary classes to interact with the environment.

For a better explanation, a case study was introduced and implemented. NASA ANTS is a case study that is suitable for examining concepts in adaptive systems in distributed environment. A scenario for self-optimizing and self-protecting was implemented and explained in order to show the applicability of the JADE extension. For better understanding the proposed extension, two sequence diagrams have been discussed: one for describing the ordinary MAPE loop with details and another one is for describing the implemented case study.

In this extension, some aspects are still to be defined in a precise way. One of these aspects is about the usage of `Knowledge`. Besides the implementation of `Knowledge` class, further classes are needed to use it, for instance a *rule engine*; moreover, a *knowledge management* and a *knowledge reasoning* should be implemented by *rules* and *rule engines* in future work.

With regard to future work, we can sketch three directions. First, the introduction of more complex classes for behaviours to support complex adaptive properties (like self-healing) more easily. Also, the explicit implementation of the *knowledge* with well-known knowledge management framework for knowledge part in MAPE-K loop. Finally, the addition of more configurable details in order to support *domain specific* needs (for example, adapting this extension to Wireless Sensor Networks issues).

These kinds of extensions are important towards the definition of standards for AC in multi-agent systems.

REFERENCES

[1] M. Baldoni, G. Boella, M. Dorni, R. Grenna, and A. Mugnaini, *Adding Organizations and Roles as Primitives to the JADE Framework*, Proc. of WOA 2008: Dagli oggetti agli agenti, Evoluzione dell'agent development: metodologie, tool, piattaforme e linguaggi, (2008), pp. 84–92.

[2] C. Baumer, M. Breugst, S. Choy, and T. Magedanz, *Grasshoppera universal agent platform based on omg masif and fipa standards*, in First International Workshop on Mobile Agents for Telecommunication Applications (MATA99), Citeseer, 1999, pp. 1–18.

[3] F. Bellifemine, G. Caire, T. Trucco, and G. Rimassa, *Jade programmer's guide*, vol. 3, CSELT S.p.A., 2000.

[4] F. Bellifemine, A. Poggi, and G. Rimasa, *JADE: a FIPA2000 compliant agent development environment*, International Conference on Autonomous Agents and Multiagent Systems, (2001), pp. 216–217.

[5] F. Bellifemine, A. Poggi, and G. Rimassa, *JADE-A FIPA-compliant agent framework*, Proceedings of PAAM, (1999), pp. 97–108.

[6] F. Bellifemine, A. Poggi, and G. Rimassa, *Developing multi-agent systems with a fipa-compliant agent framework*, Software-Practice and Experience, 31 (2001), pp. 103–128.

[7] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills III, and Y. Diao, *Able: A toolkit for building multiagent autonomic systems*, IBM Systems Journal, 41 (2002), pp. 350–371.

[8] I. Bojic, T. Lipic, M. Kusek, and G. Jezic, *Extending the JADE agent behaviour model with JBehaviourTrees Framework*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 6682 LNAI (2011), pp. 159–168.

[9] S. Bouchenak, F. Boyer, N. De Palma, D. Hagimont, S. Sicard, and C. Taton, *JADE: A Framework for Autonomic Management of Legacy Systems*, Wiki.Jasmine.Ow2.Org, (2006), p. 20.

[10] G. Cabri, N. Capodieci, L. Cesari, R. De Nicola, R. Pugliese, F. Tiezzi, and F. Zambonelli, *Self-expression and dynamic attribute-based ensembles in scel*, in International Symposium On Leveraging Applications of Formal Methods, Verification and Validation, Springer, 2014, pp. 147–163.
[11] N. Capodieci, E. Hart, and G. Cabri, *Artificial immunology for collective adaptive systems design and implementation*, ACM Transactions on Autonomous and Adaptive Systems (TAAS), 11 (2016), p. 6.
[12] A. Computing, *An architectural blueprint for autonomic computing*, IBM White Paper, (2006).
[13] R. S. Cost, T. Finin, Y. Labrou, X. Luan, Y. Peng, I. Soboroff, J. Mayfield, and A. Boughannam, *Jackal: a java-based tool for agent development*, Working Papers of the AAAI-98 Workshop on Software Tools for Developing Agents, 1998.
[14] P.-C. David and T. Ledoux, *Towards a framework for self-adaptive component-based applications*, in IFIP International Conference on Distributed Applications and Interoperable Systems, Springer, 2003, pp. 1–14.
[15] N. H. Dhaminda B. Abeywickrama and F. Zambonelli, *Engineering and implementing software architectural patterns based on feedback loops*, Scalable Computing: Practice and Experience, 15 (2014), pp. 84–92.
[16] N. M. do Nascimento and C. J. P. de Lucena, *Fiot: An agent-based framework for self-adaptive and self-organizing applications based on the internet of things*, Information Sciences, 378 (2017), pp. 161–176.
[17] X. Dong, S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, and S. Rao, *Autonomia: an autonomic computing environment*, in Performance, Computing, and Communications Conference, 2003. Conference Proceedings of the 2003 IEEE International, IEEE, 2003, pp. 61–68.
[18] A. Farahani, E. Nazemi, and G. Cabri, *Ace-jade, autonomic computing enabled jade*, in IEEE SASO 2016, IEEE, 2016.
[19] ———, *A self-healing architecture based on rainbow for industrial usage*, Scalable Computing: Practice and Experience, 17 (2016), pp. 351–368.
[20] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, *Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure*, IEEE Computer, 37 (2004), pp. 46–54.
[21] M. L. Griss, S. Fonseca, D. Cowan, and R. Kessler, *SmartAgent: Extending the JADE agent behavior model*, Proceedings of SEMAS, (2002), pp. 1–10.
[22] P. Horn, *autonomic computing : IBM's Perspective on the State of Information Technology*, IBM White Paper, (2001), pp. 1–10.
[23] B. Jacob, R. Lanyon-Hogg, D. K. Nadgir, and A. F. Yassin, *A Practical Guide to the IBM Autonomic Computing Toolkit*, IBM Redbooks, 2004.
[24] j. O. Kephart and D. M. Chess, *The vision of Autonomic Computing*, IEEE computer society, (2003), pp. 41–50.
[25] H. Liu, M. Parashar, and S. Hariri, *A componentbased progarmming framework for autonomic applications*, in Proceedings of the 1st IEEE International Conference on Autonomic Computing, 2004.
[26] Y. S. Lopes, E. J. T. Gonçalves, M. I. Cortés, and S. S. Emmanuel, *Extending JADE Framework to Support Different Internal Architectures of Agents*, in 9th European Workshop on Multi-agent Systems (EUMAS 2011), 2011, pp. 1–15.
[27] V. Markova, *Autonomous Agent Design Based On Jade Framework*, in Proceedings of the International Conference on Information Technologies (InfoTech-2013), no. September, 2013, pp. 19–20.
[28] P. O'Brien and R. Nicol, *FIPA - towards a standard for software agents*, BT Technology Journal, 16 (1998), pp. 51–59.
[29] D. Petcu, *A parallel rule-based system and its experimental usage in membrane computing*, Scalable Computing: Practice and Experience, 7 (2001).
[30] M. Puviani, G. Cabri, N. Capodieci, and L. Leonardi, *Building self-adaptive systems by adaptation patterns integrated into agent methodologies*, in International Conference on Agents and Artificial Intelligence, Springer, 2015, pp. 58–75.
[31] L. Raju, R. Milton, and S. Mahadevan, *Multiagent systems based modeling and implementation of dynamic energy management of smart microgrid using macsimjx*, The Scientific World Journal, 2016 (2016).
[32] C. R. Robinson, P. Mendham, and T. Clarke, *Macsimjx: A tool for enabling agent modelling with simulink using jade*, (2010).
[33] M. Salehie and L. Tahvildari, *Self-adaptive software: Landscape and research challenges*, ACM Transactions on Autonomous and Adaptive Systems, 4 (2009), pp. 14–56.
[34] W. Truszkowski, L. Hallock, J. Karlin, J. Rash, and G. Michael, *Autonomous and Autonomic Systems with Applications to NASA Intelligent Spacecraft Operations and Exploration Systems*, Springer, 2010.
[35] W. Truszkowski, M. Hinchey, J. Rash, and C. Rouff, *NASA's Swarm Missions: The challenge of Building Autonomous Software*, IT Pro, (2004), pp. 47–52.
[36] D. Weyns and R. Haesevoets, *The MACODO middleware for context-driven dynamic agent organizations*, ACM Transactions on Autonomous and Adaptive Systems (TAAS), 5 (2010).
[37] F. Zambonelli, *Self-management and the many facets of nonself*, IEEE Intelligent systems, 21 (2006), pp. 53–55.

Edited by: Dana Petcu
Received: December 24, 2016
Accepted: March 22, 2017

AIMS AND SCOPE

The area of scalable computing has matured and reached a point where new issues and trends require a professional forum. SCPE will provide this avenue by publishing original refereed papers that address the present as well as the future of parallel and distributed computing. The journal will focus on algorithm development, implementation and execution on real-world parallel architectures, and application of parallel and distributed computing to the solution of real-life problems. Of particular interest are:

Expressiveness:

- high level languages,
- object oriented techniques,
- compiler technology for parallel computing,
- implementation techniques and their efficiency.

System engineering:

- programming environments,
- debugging tools,
- software libraries.

Performance:

- performance measurement: metrics, evaluation, visualization,
- performance improvement: resource allocation and scheduling, I/O, network throughput.

Applications:

- database,
- control systems,
- embedded systems,
- fault tolerance,
- industrial and business,
- real-time,
- scientific computing,
- visualization.

Future:

- limitations of current approaches,
- engineering trends and their consequences,
- novel parallel architectures.

Taking into account the extremely rapid pace of changes in the field SCPE is committed to fast turnaround of papers and a short publication time of accepted papers.

INSTRUCTIONS FOR CONTRIBUTORS

Proposals of Special Issues should be submitted to the editor-in-chief.

The language of the journal is English. SCPE publishes three categories of papers: overview papers, research papers and short communications. Electronic submissions are preferred. Overview papers and short communications should be submitted to the editor-in-chief. Research papers should be submitted to the editor whose research interests match the subject of the paper most closely. The list of editors' research interests can be found at the journal WWW site (`http://www.scpe.org`). Each paper appropriate to the journal will be refereed by a minimum of two referees.

There is no a priori limit on the length of overview papers. Research papers should be limited to approximately 20 pages, while short communications should not exceed 5 pages. A 50–100 word abstract should be included.

Upon acceptance the authors will be asked to transfer copyright of the article to the publisher. The authors will be required to prepare the text in LaTeX 2ε using the journal document class file (based on the SIAM's `siamltex.clo` document class, available at the journal WWW site). Figures must be prepared in encapsulated PostScript and appropriately incorporated into the text. The bibliography should be formatted using the SIAM convention. Detailed instructions for the Authors are available on the SCPE WWW site at `http://www.scpe.org`.

Contributions are accepted for review on the understanding that the same work has not been published and that it is not being considered for publication elsewhere. Technical reports can be submitted. Substantially revised versions of papers published in not easily accessible conference proceedings can also be submitted. The editor-in-chief should be notified at the time of submission and the author is responsible for obtaining the necessary copyright releases for all copyrighted material.