HAL open science

Communication and Memory Optimized Tree Contraction and List Ranking

Jens Gustedt

▶ To cite this version:

Jens Gustedt. Communication and Memory Optimized Tree Contraction and List Ranking. [Research Report] RR-4061, INRIA. 2000, pp.9. inria-00072575

HAL Id: inria-00072575

https://inria.hal.science/inria-00072575

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Communication and Memory Optimized Tree Contraction and List Ranking

Jens Gustedt

N° 4061

Novembre 2000

THÈME 1

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--4061--FR+ENG

Communication and Memory Optimized Tree Contraction and List Ranking

Jens Gustedt*

Thème 1 — Réseaux et systèmes
Projet Résédas

Rapport de recherche n° 4061 — Novembre 2000 — 9 pages

Abstract: We present a simple and efficient algorithm for the Tree Contraction Problem on a Coarse Grained p-Multiprocessor. With respect to its total computing time, its demand of memory and its total inter-processor communication it is only a small constant factor away from optimality. Unless traditional PRAM algorithms it doesn't use List Ranking as a necessary subroutine but specializes to List Ranking when applied to lists.

Key-words: tree contraction, list ranking, coarse grained parallel algorithms

* INRIA Lorraine / LORIA, France. Email: Jens.Gustedt@loria.fr – Phone: +33 3 83 59 30 90

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Contraction d’arbres et classement de listes optimisés pour la communication et l’accès mémoire

Résumé : Nous présentons un algorithme simple et efficace pour le problème de contraction d’arbre sur une machine de p processeurs à gros grain. Pour son coût de calcul total, son besoin de mémoire et sa communication totale entre processeurs il n’est qu’un petit facteur constant de l’optimalité. Autre que les algorithmes PRAM traditionnels il n’utilise pas le classement de listes comme sous-routine mais spécialise au classement de listes si appliqué ainsi.

Mots-clés : contraction d’arbres, classement de listes, algorithmes parallèles à gros grain

1 Introduction and Overview

The *tree contraction* problem is one of the fundamental subroutines for parallel computation in graphs. It addresses the need of performing bottom-up computations in a tree, and the propagation of the corresponding result back down from the root to the leaves. It specialized to list ranking, if the tree is just a path, or computation of connected components in a graph if the tree is spanning. It was intensively studied in the context of the PRAM model of parallel computation, see Karp and Ramachandran (1990) for an overview, but not yet so much special attention has been given to it in coarse grained models. Solutions proposed so far heavily relay on the PRAM approaches, which in turn rely on list ranking, see Caceres et al. (1997). Since implementing list ranking can be considered as only been partially solved, see Guérin Lassous and Gustedt (2000), it is worth looking at a generic algorithm to solve tree contraction directly in that setting of coarse grained models of parallel computation.

The gap between the available parallel architectures and theoretical models was narrowed by Valiant (1990) by defining the so-called *bulk synchronous parallel* machine, BSP. Based upon the BSP, the model that is used in this paper, the so-called *Coarse Grained Multiprocessor*, CGM, was developed to combine theoretical abstraction with applicability to a wide range of architectures, see Dehne et al. (1996). It assumes that the number of processors p is small compared to the size of the data and that communication costs between processors are high. One of the main goals for algorithms formulated for that model is to reduce these communication costs to a minimum. The first measure that was introduced was the number of communication rounds: an algorithm is thought to perform local computations and global message exchanges between processors in alternation, so-called rounds.

The implementation of the algorithm given in this paper in fact proves that putting too much attention on the minimization of the number of rounds is not worth the effort. A number of rounds that is only bounded by $O(\log(n))$, that is a function in the **size of the data**, and not $O(\log(p))$, the **number of processors**, seems sufficient to hide the penalty imposed by the latency of the underlying communication network.

2 The CGM model for parallel computation

The basic ideas that characterize the CGM model are:

uniformity A CGM consists of a small number p of uniform processing units (*processors*). 'Small' here means magnitudes smaller than the size n of the input data.

code sharing All processors execute the same program.

simultaneous alternation The overall execution of an algorithm alternates simultaneously between phases with only computations local to the processors and communications between those processors, so-called *rounds*.

implicit synchronization Synchronization of the different processors is only done implicitly during the communication rounds.

Besides its simplicity, this approach also has the advantage of allowing design of algorithms for a large variety of existing hardware and software platforms. It does this without going into the details and special characteristics of such platforms, but gives predictions in terms of the number of processors p and the number of data items n only.

3 Basics of the Tree Contraction Problem

Tree contraction in its most general form handles the evaluation of expressions that are given by a tree T that prescribes the evaluation order, an operation $\oplus$ that has to be performed at each inner node of T, and constants of some domain D that reside on the nodes (inner nodes and leaves) of T. Two values are of interest on each node v of T, its *value* and its *rank*. The value is the result of the evaluation of the subexpression that is rooted at v. The rank is the result of the evaluation of the subexpression corresponding to the path leading from v to the root r of T.

There are many examples of algorithmic problems that fit into this setting, e.g

- computing the depth of each subtree of a tree,
- computing the distance of a node from the root,
- finding the root of the tree corresponding to each node in a forest, i.e connected components in a forest.

At a first glance it might seem that this problems is easy to solve in parallel. Each subtree rooted at some node v can in general be handled independently from the other. So an easy to program parallelism of control seems to do the trick. The difficulty comes from the possibly irregular structure of the underlying tree T. It might contain long paths of nodes of in-degree 1 and so such a "parallelization" will fail and fall back to a sequential execution.

In the PRAM setting this problem has intensively studied, and the solution adopted their is the combination of list ranking for the contraction of long chains and the so-called SHUNT operation. During that operation, vertices of in-degree 2 "store" the value a_1 of their first subexpression and then apply this value (with some nice tricks) to evaluate the result as a whole from what comes from the other subexpression and this value a_1. Beyond others, this approach has the disadvantage of

- having the need to avoid conflicts when two children try to pass their value to the parent at the same time,
- using list ranking as a subroutine, which has by far no simple solutions, neither have their been portable implementations on real machines that were completely convincing.

4 The Pruning Algorithm

Consider Algorithm 1. It describes the translation of a classical "*pruning*" algorithm into our algorithmical setting. First, it looks for the set L of leaves of the tree, deactivates them and notifies their parents of the degree change. Then, it recurses on the remaining set of nodes. After coming back from recursion, the parent pointers and values of the nodes in L are updated to the final values.

Pruning is then called with the right initialization of states of all vertices in Algorithm 2. The aim of the theory part of this paper is actually to show that Algorithm 2 is correct and can be implemented efficiently in the CGM setting.

A first observation (given in the next proposition) shows that this algorithm does already quite well in our setting if we can ensure that we find *sufficiently* many leaves in each step. Sufficiently here means an ε-fraction of the active elements for some fixed value $0 < \varepsilon < 1$. Line **contract** takes care of the situation when this condition is not fulfilled by calling a function `Contract` which will provide us with another set of leaves L' and which we will explain later.

PROPOSITION 4.1. *Suppose that for some ε there are always sufficiently many leaves in A available. Then the recursion depth is at most $\frac{1}{\varepsilon}\log n$. The overall computation is $O(n)$ and the communication issued is $3n$ words.*

Proof. $|A|$ reduces by a factor of at least $1-\varepsilon$ in each iteration. So the maximum number of iterations i fulfills

$$
\begin{aligned}
& (1-\varepsilon)^i n < 1 && (4.1)\\
\Longrightarrow\quad & n < (1-\varepsilon)^{-i} && (4.2)\\
\Longrightarrow\quad & \log_2 n < i \log_2 \frac{1}{1-\varepsilon} && (4.3)\\
\Longrightarrow\quad & \frac{\log_2 n}{\log_2 \frac{1}{1-\varepsilon}} < i && (4.4)
\end{aligned}
$$

So i can be found to be about

$$
\frac{\log_2 n}{\log_2 \frac{1}{1-\varepsilon}} \approx \frac{\log_2 n}{\frac{1}{1-\varepsilon}-1} = \frac{\log_2 n}{\frac{\varepsilon}{1-\varepsilon}} = (\frac{1}{\varepsilon}-1)\log_2 n \qquad (4.5)
$$

Algorithm 1: `Prune`(R,A)

Input: Set R of n items e with pointer e.parent, indegree $0 \leq e.\mathsf{indeg} \leq 2$. A subset $A \subseteq R$ of *active* elements.

bottom **if** $A = \emptyset$ *on all processors* **then return** ;

de-activate Set $L = L' = \emptyset$;

foreach $a \in A$ *with* $a.\mathit{indeg} = 0$ **do** move a from A to L;

update indeg **begin**

  foreach $a \in L$ **do** prepare a message to a.parent;

$\longleftrightarrow$ Exchange the messages between the processors;

  foreach $r \in R$ *that received a message* **do**

    foreach $s \in S$ *from which* r *received a message* **do**

      Decrement r.indeg;

end

contract **if** $|L| < \varepsilon|A|$ **then**

  `Contract`(R,A);

  foreach $a \in A$ *with* $a.\mathit{indeg} = 0$ **do** move a from A to L';

indeg **Invariant**: For all $r \in R$, r.indeg is the number of direct children of r that are active.

recurse `Prune`(R,A);

collect **begin**

  foreach $a \in L \cup L'$ **do** Prepare a message to a.parent;

$\longleftrightarrow$ Exchange the messages between the processors;

  foreach $r \in R$ *that received a message* **do**

    foreach $s \in L \cup L'$ *from which* r *received a message* **do**

      Prepare a message to s containing r.parent;

$\longleftrightarrow$ Exchange the messages between the processors;

  foreach a *that receives a message* **do**

    Set a.parent to the node that is received;

end

Algorithm 2: `TreeContraction`(R)

Input: Set R of n items e with pointer e.parent

Compute the indegrees for all R;

Set A to the elements that are not themselves the root of their corresponding subtree;

`Prune`(R,A);

For the overall work and communication observe that both are directly related to the size of L and that the union of all L in all recursive calls is R.

At most two messages are sent from a to a.parent and at most one message is sent back. So in total the communication is

$$3\sum_{j=0}^{i} |L_j| \quad \leq \quad 3n \tag{4.6}$$

REMARK 4.1. *Suppose that for some* ε *there are sufficiently many leaves in* A *available. Then invariant* indeg *of Algorithm 1 holds if it holds on entry.*

5 Contracting Degree 1 Nodes

Now let us consider the case that we don't find sufficiently many leaves. Since the number of leaves and the number of degree 2 nodes are directly related this can only be the case if there are many nodes of degree 1. Figure 1 illustrates the operation on such nodes that we would like to perform, namely the contraction operation.

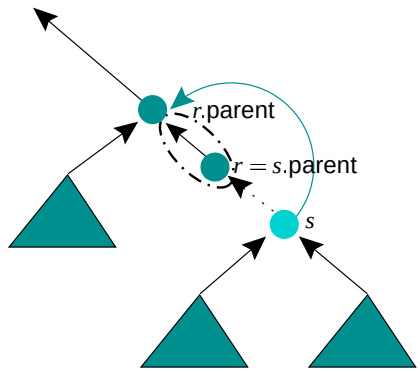

Figure 1: The contraction operation

Here the child s of a degree-1 node r gets its parent changed to its grandparent. Thereby r becomes a leaf, and can be pruned (almost) as before. If r immediately ceases activity, from the point of view of the grandparent not much has changed; it has exactly the same number of active children as before.

Other than with the pruning step we are faced with a problem of concurrency. If s is doing such a contraction operation, r mustn't do the same kind of operation with its parent at the same time. This is reflected in Algorithm 3 where we do not perform this operation on all degree-1 nodes, but only on those that are pointed to by an *independent set* S.

Algorithm 3: `Contract`(R,A,S)

Input: Set R of n items e with pointer e.parent, discovered indegree $0 \le e.\mathsf{indeg} \le 2$. A subset $A \subseteq R$ of *active* elements.

$S =$ `Indep`(R,A);

ε-independent **Invariant**: S consists of pairwise independent items i.e such that for all $s \in S$ we have that $s.\mathsf{parent} \notin S$ and such that $|S| > 2\varepsilon|A|$ for some ε.

foreach $s \in S$ **do** Prepare a message to s.parent;

$\longleftrightarrow$ Exchange the messages between the processors;

foreach $r \in R$ *that receives a message* **do**

  foreach $s \in S$ *from which r receives a message* **do**

    if $r.\mathsf{indeg} > 1$ **then**

      Prepare a message to s containing r;

    else

      Prepare a message to s containing r.parent;

      Set $r.\mathsf{indeg} = 0$;

$\longleftrightarrow$ Exchange the messages between the processors;

foreach $s \in S$ **do** Set s.parent to the node that is received;

PROPOSITION 5.1. *Let $0 < \varepsilon < \frac{1}{2}$ be fixed and suppose there are less then $\varepsilon|A|$ leaves in Line contract and such that* `Indep`(R,A) *computes an independent set $S \subset A$ with $|S| \ge 2\varepsilon|A|$. Then an application of Algorithm 3 produces $\varepsilon|A|$ new leaves in $|A|$.*

Proof. Let n_0, n_1, and n_2 the number of nodes of degree $0, 1, 2$ respectively and $k > 1$ the number of components of the forest. We have that $n_2 = n_0 - k$. The messages sent in Algorithm 3 are sent to nodes of degree at least one. They fail to produce a new leaf if they are received by a node of degree 2. In total there can be no more failures than $n_2 = n_0 - k < n_0 < \varepsilon|A|$. So we produce at least $|S| - n_2 > 2\varepsilon|A| - \varepsilon|A| = \varepsilon|A|$ new leaves.

6 Finding Large Independent Sets

The remaining issue to make the whole setting work is to ensure that we always find a sufficiently large independent set. There are mainly two PRAM approaches to accomplish this tasks:

- random mating and
- deterministic symmetry breaking.

Both approaches have the disadvantage to produce a substantial volume of communication. We will describe two different approaches, one (pseudo-)randomized and one deterministic, see Guérin Lassous and Gustedt (2000) and Gebremedhin et al. (2000) for other applications of these concepts in the CGM context. Both are modifications of random mating which we briefly recall now.

For random mating every node chooses a gender *male* or *female*. Then we choose those *female*s for the independent set that point to a *male*. Clearly these form an independent set and its expected size is $\frac{|A|}{4}$.

As said, the disadvantage for a practical setting of this approach is that we have to communicate the chosen genders to be able to decide whether or not a node belongs to the independent set. This can be avoided by using a pseudo-random number generator $rand(s,t)$ that depends on two *state* values s and t. If a common value for s is chosen for all processors, then every node can determine a gender for a node r and its parent $r.\mathsf{parent}$ by computing $rand(s,r)$ and $rand(s,r.\mathsf{parent})$ without doing any communication. So the only task that is to be accomplished is to ensure that the genders chosen for r in different rounds are sufficiently independent. This can be easily done if s is chosen differently in each round.

A second approach is *deterministic mating*. This approach uses special properties of the CGM to construct a mating (and thus independent set) of the desired properties. Up to our knowledge, this technique was first described by Gebremedhin et al. (2000).

The main idea of that procedure is to divide the sets of (active) nodes on each processor into two equal sized halves, *left* and *right*, such that each other processor can determine directly to which half a particular vertex belongs. This can be accomplished by dividing the nodes according to their *id* (or address) and communicating the value that splits up the two halves to all the other processors.

In a second phase, each processor classifies the arcs of its nodes in *internal* and *external* arcs. Internal are those whose parent is a vertex that is in the same half on the same processor and external are all the others. Then it determines the amount of its external arcs pointing to the two individual halves on each processor.

The result of this operation is a matrix of $O(p^2)$ integer values which is collected by one of the processors. This processor performs a (sequential) computation to obtain a optimized *flipping* of the two sides. That is, if necessary the roles of the two halves on some of the processors are flipped to ensure that at least half of the external edges go between different sides. This can easily be achieved by some greedy algorithm that runs in $O(p^2)$ time, see Gebremedhin et al. (2000) for more details.

Then, each processor assigns (new) genders to nodes which only have internal arcs. The genders of these may be changed without affecting the choices that have been made on other processors, and clearly this can be done greedily in linear time and such that a quarter of them takes part in the independent set.

We summarize with the following theorem.

THEOREM 6.1. *An algorithm* `Indep` *can be implemented in such a way such that it produces an independent set of size* $\Omega(\frac{|A|}{4})$*, runs in three supersteps, and has a communication cost of* $O(p^2)$ *and a total work of* $O(|A|)$.

7 Experimental results

A first partial implementation of the algorithm was realized with the `sscrap`-library[1]. In fact, only the restriction to list ranking has been implemented so far. The implementation proves to be a little bit slower than the one described in Guérin Lassous and Gustedt (2000). But on the other hand due to its lower memory consumption is easier scalable and runs without problems on a multiprocessor with 60 processors or on multiprocessor PCs or PC-clusters. Its break-even point in comparison to a sequential implementation is at about 10 processors.

The advantage of this algorithm, seen as a new variant of list ranking, is that we don't need to invert the list, as do all other algorithms up to our knowledge. Such a list that hold the predecessors of a node, simply doubles the memory requirement of the program. Something that is clearly to avoid for such memory sensitive applications.

Figure 2 shows running times *per item* on SGI Origin2000 multiprocessor. We also have undertaken a

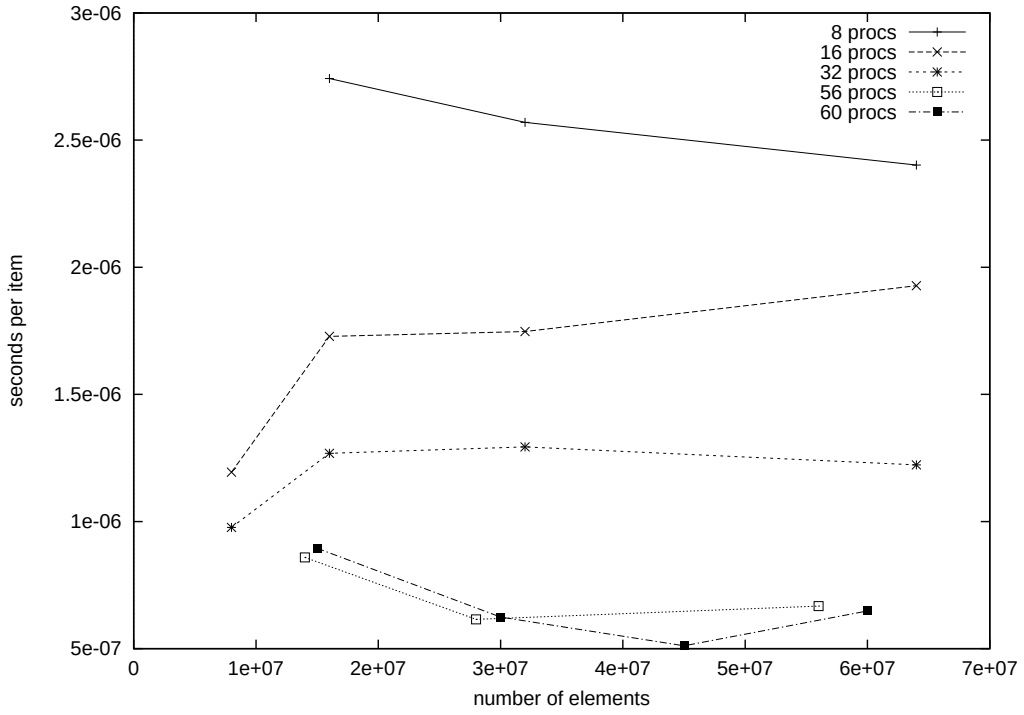

Figure 2: Runtime results on an Origin 2000 multiprocessor

purely sequential implementation to test the memory performance of the algorithm. Unfortunately the running times are not yet completely competitive: the algorithm is for the moment twice as slow as a sequential version. We think that this already is quite promising, having in mind that straight forward implementations of parallel list ranking have a total work load that is at least about 10 times higher than the sequential running time.

References

[Caceres et al., 1997] Caceres, E., Dehne, F., Ferreira, A., Flocchini, P., Rieping, I., Roncato, A., Santoro, N., and Song, S. W. (1997). Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In Degano, P., Gorrieri, R., and Marchetti-Spaccamela, A., editors, *Automata, Languages and Programming*, volume 1256 of *Lecture Notes in Comp. Sci.*, pages 390–400. Springer-Verlag. Proceedings of the 24th International Colloquium ICALP'97.

[1] http://www.loria.fr/~ gustedt/cgm

[Dehne et al., 1997] Dehne, F., Dittrich, W., and Hutchinson, D. (1997). Efficient external memory algorithms by simulating coarsegrained parallel algorithms. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 106–115.

[Dehne et al., 1996] Dehne, F., Fabri, A., and Rau-Chaplin, A. (1996). Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry*, 6(3):379–400.

[Gebremedhin et al., 2000] Gebremedhin, A. H., Guérin Lassous, I., Gustedt, J., and Telle, J. A. (2000). Graph coloring on a coarse grained multiprocessor. In *WG 2000*, page 12 p. Springer-Verlag. to appear.

[Guérin Lassous and Gustedt, 2000] Guérin Lassous, I. and Gustedt, J. (2000). Portable list ranking: an experimental study. In *Workshop on Algorithm Engeneering (WAE 2000)*, LNCS. Springer-Verlag. *to appear*.

[Guérin Lassous et al., 2000] Guérin Lassous, I., Gustedt, J., and Morvan, M. (2000). Handling graphs according to a coarse grained approach: Experiments with MPI and PVM. In Dongarra, J., Kacsuk, P., and Podhorszki, N., editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users' Group Meeting*, volume 1908 of *LNCS*, pages 72–79. Springer-Verlag.

[Karp and Ramachandran, 1990] Karp, R. M. and Ramachandran, V. (1990). Parallel Algorithms for Shared-Memory Machines. In van Leeuwen, J., editor, *Handbook of Theoretical Computer Science*, volume A, Algorithms and Complexity, pages 869–941. Elsevier Science Publishers B.V., Amsterdam.

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

Unité de recherche INRIA Lorraine
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399