A Work-Optimal Algorithm on log delta n Processors for a P-Complete Problem

Jens Gustedt, Jan Arne Telle

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A work-optimal algorithm on $\log^{\delta} n$ *processors for a P-complete problem*

Jens Gustedt — Jan Arne Telle

N° 4174

Avril 2001

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4174--FR+ENG

A work-optimal algorithm on $\log^{\delta} n$ processors for a P-complete problem

Jens Gustedt* , Jan Arne Telle**

Thème 1 — Réseaux et systèmes
Projet Résédas

Rapport de recherche n° 4174 — Avril 2001 — 9 pages

Abstract: We present a parallel algorithm for the *Lexicographically First Maximal Independent Set Problem* on graphs with bounded degree 3 that is work-optimal on a shared memory machine with up to $\log^{\delta} n$ processors, for any $0 < \delta < 1$. Since this problem is P-complete it follows (assuming $\mathcal{NC} \neq \mathcal{P}$) that the algorithmics of coarse grained parallel machines and of fine grained parallel machines differ substantially.

Key-words: P-completeness, coarse grained parallelism, parallel algorithms

* INRIA Lorraine / LORIA, France. Email: Jens.Gustedt@loria.fr – Phone: +33 3 83 59 30 90
** University of Bergen, Bergen, Norway. Email: telle@ii.uib.no

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

Un algorithme de travail optimal sur $\log^{\delta} n$ processeurs pour un problème P-complet

Résumé : Nous présentons un algorithme parallèle pour le problème du *premier ensemble indépendent maximal par ordre lexicographique* sur des graphes avec un dégrée borné à 3. Il est de travail optimal sur une machine à mémoire partagée avec juste à $\log^{\delta} n$ processeurs, pour tous $0 < \delta < 1$. Ce problème est P-complet. Si on suppose que $\mathcal{NC} \neq \mathcal{P}$, l'algorithmique des machines parallèles à gros grain et à grain fin doivent alors différer substantiellement.

Mots-clés : P-complitude, parallèlisme à gros grain, algorithmes parallèles

1 Motivation and Background

The aim of this paper is to give weight to the study of algorithms and computational complexity of coarse grained parallel models, by showing that it provides a refinement over the classification achieved by fine grained models.

For many years the intensive study of the algorithmics of the PRAM, a fine grained parallel machine model with an arbitrary number of processors, was motivated by *Brent's scheduling principle*, see [1], *i.e.* by the fact that any efficient algorithm on the PRAM could be simulated using a smaller number of processors imposing only an overhead on the total work of a constant factor. Thus any work-optimal algorithm on a PRAM could in theory be implemented on a coarse grained machine (*i.e.* with few processors) and still guarantee linear speedup.

But the converse of that statement does *apriori* not hold. A work-optimal algorithm on a machine with few processors does in general not scale up to an efficient PRAM algorithm. Thus in particular also the impossibility results (assuming $\mathcal{NC} \neq \mathcal{P}$) for the PRAM *apriori* do not translate into impossibility results for coarse grained machines.

In this paper we prove that not only does *Brent's scheduling principle* not allow us to scale algorithms *up* instead of *down* but that no other general technique will be able to do this either. We do this by showing that a certain P-complete problem has a work-optimal coarse grained algorithm on $\log^\delta n$ processors, for any $0 < \delta < 1$. Thus any technique that would give the possibility to scale up would in fact give an $\mathcal{NC}$-algorithm for a P-complete problem, something we shouldn't expect.

Although our research was guided by the more practical and realistic coarse grained machine models for parallel computation, see [8,2,3], we will for this paper use simply the PRAM while taking the granularity restriction into account. This is done to make the approach as transparent as possible and not get lost behind certain (practically motivated) constraints of the coarse-grained models. For the time being the algorithm that we present here is of theoretical interest only.

2 Parallel machine models and performance measures

We use a convenient restriction of the classical CREW-PRAM[1], see [6] for an overview of this and related parallel machine models. This classical model has one big (dis)advantage when compared to the real parallel machines available nowadays: to ensure an exponential improvement in running time compared to a sequential RAM it assumes that the number of processors p is a polynomial, at least linear, in the input size n.

The main performance measure of a PRAM algorithm $\mathcal{A}$ is its parallel *running time* $T_{\mathcal{A}}(n)$ which should be polylogarithmic in the input size n. To achieve this it is usually accepted that the overall *work* $C_{\mathcal{A}}(n) = T_{\mathcal{A}}(n)p_{\mathcal{A}}(n)$, where $p_{\mathcal{A}}(n)$ denotes the number of processors used by $\mathcal{A}$, exceeds the time $T_{\mathcal{A}^*}(n)$ of an optimal sequential algorithm $\mathcal{A}^*$ for the problem, by a polynomial in n. Unless such PRAM algorithms are *work-optimal, i.e.* $C_{\mathcal{A}}(n) = O(T_{\mathcal{A}^*}(n))$, they will show poor performance when scaled down, using Brent's principle, to a coarse grained machine.

When we want algorithms that are scalable for a range of processors their running time $T_{\mathcal{A}}(p,n)$ becomes a function of p, the number of processors, too. What is most commonly referred to as a good measure for efficiency is the so-called *speedup* of a parallel algorithm, $S_{\mathcal{A}}(p,n) = T_{\mathcal{A}^*}(n)/T_{\mathcal{A}}(p,n)$, see [5] for a glossary of terminology. By Brent's scheduling principle the speedup is at most p, the number of processors.

Usually, *linear speedup* means that the speedup is $O(p)$. However, we need a more precise definition of linear speedup since we want our algorithms to be scalable for a range of processors. We will say that a parallel algorithm $\mathcal{A}$ is *N-scalable*, for some function $N(p)$, if there are constants p_0 and $e > 1/p_0$ such that for all $p \geq p_0$ it has a linear speedup, *i.e* $S_{\mathcal{A}}(p,n) \geq ep$ for all $n > N(p)$.
In this paper we are dealing with a CREW-PRAM that obeys

$$p \leq \log^\delta n \text{ for some } \delta < 1, \tag{1}$$

[1] Concurrent Read Exclusive Write - Parallel Random Access Machine

or equivalently

$$n \geq^{\Delta p} \text{ for some } \Delta > 1. \tag{2}$$

Theorem 1. *There is a $\mathcal{P}$-complete problem that has, for any constant $\Delta > 1$, a $2^{\Delta p}$-scalable parallel algorithm.*

N-scalability is a property that requires that an algorithm takes the number of processors p as an additional input. On the other hand, for the theoretical design and analysis of an algorithm it is often much easier to let the algorithm choose p as a function of the input size n. Brent's scheduling principle shows that there is a constant d (depending only on the machine model) such that any algorithm $\mathcal{A}$ that has a linear speedup $S_{\mathcal{A}}(p,n) \geq e_p p$ on $p > p'$ processors can be simulated on p' processors such that the simulation $\widetilde{\mathcal{A}}$ has speedup $S_{\widetilde{\mathcal{A}}}(p',n) \geq \frac{e_p}{d} p'$. Thus for a choice of $p_0 > \frac{d}{e_p}$ $\widetilde{\mathcal{A}}$ has a linear speedup for the constant $\frac{e_p}{d}$. The following lemma is an easy extension of this observation that simply allow us to chose p as a function of n for the design of our algorithm.

Lemma 1. *Let $\mathcal{A}$ be an algorithm that solves some problem $\mathcal{P}$, $N(p)$ a monotone unbounded function and $e > 0$ a constant such that*

1. *on p processors $\mathcal{A}$ has $S_{\mathcal{A}}(p,n) \geq ep$ for all n with $N(p) \leq n \leq N(p+1)$,*
2. *there is some constant $d' \geq 1$, such that for any n the value of processors p with $N(p) \leq n \leq N(p+1)$ can be computed in time $\kappa(p)$ such that $\kappa(p) \leq \frac{d'}{e} \frac{T_{\mathcal{A}^*}(n)}{p}$.*

Then there is an algorithm $\widetilde{\mathcal{A}}$ that solves $\mathcal{P}$ and that is N-scalable.

Proof. Suppose we are given an input size n and a machine with p processors such that $n \geq N(p)$. We lookup p' such that $N(p') \leq n \leq N(p'+1)$. Because of the monotonicity of N we must have that $p \leq p'$.

Then we simulate $\mathcal{A}$ with p' processors on our machine of p processors. This takes a total running time of

$$T_{\widetilde{A}}(p,n) = \kappa(n) + dT_{\mathcal{A}}(p',n) \tag{3}$$

We have $T_{\mathcal{A}^*}(n) \geq ep'T_{\mathcal{A}}(p',n)$ and thus $T_{\mathcal{A}^*}(n)/(ep') \geq T_{\mathcal{A}}(p',n)$. The claim is then shown by noting that the over all speedup is then bounded as follows:

$$S_{\widetilde{A}}(p,n) = \frac{T_{\mathcal{A}^*}(n)}{T_{\widetilde{A}}(p,n)} \geq \frac{ep'T_{\mathcal{A}^*}(n)}{(d+d')T_{\mathcal{A}^*}(n)} \geq \frac{ep'}{d+d'} \geq \left(\frac{e}{d+d'}\right) p \tag{4}$$

□

In view of Lemma 1 it will suffice to consider only certain values of n and p, namely such that $p = \lfloor \log^{\delta} n \rfloor$ or equivalently $2^{\Delta p} \leq n < 2^{\Delta(p+1)}$.

The other inconvenience of the parameters that we are using is the fact that they depend on $\mathcal{A}^*$, an optimal sequential algorithm that may not be known. We circumvent this easily as the problem we will consider has a linear-time sequential algorithm.

The main goal of the paper will be to prove Theorem 1. Since the algorithm that we give will use bit-parallelism let us be more precise about the "RAM" part of the machine description. We will assume that each processor is a RAM with word size w and that it supports all conventional operations (*e.g* addition, subtraction, bitwise `and` and `or`) on machine words *in constant time*.

We will also assume that the word-size of our machine is at least logarithmic in the input size (here the number of vertices of a graph G) since otherwise our input could not be randomly addressed in its entirety, *i.e.*

$$\log n \leq w \tag{5}$$

Algorithm 1 Sequential LFMIS

Input: A graph $G=(V,E)$ of max degree 3 with an ordering $v_1,\ldots,v_n$ on V.
Output: The lexicographically first maximal independent set S.
Data Str.: Boolean vector $\texttt{notS}[v_1 \ldots v_n]$ with $\texttt{notS}[v_i]=1$ only if $v_i \notin S$.
begin
  for $i=0$ **to** n **do** $\texttt{notS}[v_i]=0$
  for $i=0$ **to** n **do**
    for all edges $v_iv_j \in E$ with $i<j$ **do**
      if $\texttt{notS}[v_i]==0$ **then** $\texttt{notS}[v_j]=1$
  output $\{v : \texttt{notS}[v]==0\}$
end;

3 LFMIS and the Block Graph

The P-complete problem that we are using to prove Theorem 1 is the following, see the book [4] for an overview.

Problem 1 (Lexicographically First Maximal Independent Set (LFMIS)).
Input: An undirected graph $G=(V,E)$ with an ordering $v_1,\ldots,v_n$ on V and a designated vertex v.
Output: Is vertex v in the lexicographically first maximal independent set?

The LFMIS problem is P-complete even if the input graph is restricted to be planar and have maximum degree 3, see [7]. From now on we consider only graphs of maximum degree 3. In the following we will in fact not solve the decision problem as given by the definition, but give an algorithm that produces the corresponding independent set. We will derive our algorithm from the straightforward linear-time sequential algorithm, see Algorithm 1, that may also stand as the definition of 'lexicographically first maximal independent set'.

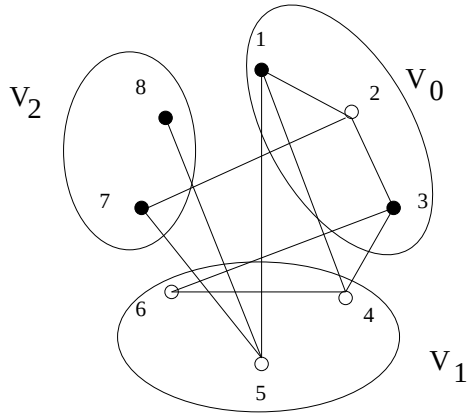

Figure1. The B-block of a graph with max degree 3 and vertices ordered $1\ldots8$ for $B=3$, *i.e.* 3 vertices per block. Black vertices are in the LFMIS. Inter-block edges between V_0 and V_1 are represented by a vector of $3B$ numbers in the range $0\ldots B$, namely $\langle 120\rangle\cdot\langle 000\rangle\cdot\langle 130\rangle$, with $\langle 120\rangle$ denoting that vertex 1 is adjacent to 4 and 5 (which have relative numbers 1 and 2 in V_1), $\langle 000\rangle$ denoting that vertex 2 has no adjacencies, and $\langle 130\rangle$ denoting that vertex 3 is adjacent to vertices 4 and 6 (which have relative numbers 1 and 3). Likewise, intra-block edges for V_0 are represented by the vector $\langle 200\rangle\cdot\langle 130\rangle\cdot\langle 200\rangle$.

The idea presented in this paper to obtain a parallel algorithm is to alter the conventional data structure for the input graph so that several processors will be able to handle a set of different edges concurrently. Therefore we will need to compress information concerning certain vertex and edge sets into machine words. To obtain such a "*compressed*" representation of the input graph, we partition it into vertex blocks of fixed size B, and consider the representation of intra-block and inter-block edges.

Definition 1. *Given an undirected graph $G=(V,E)$ of max degree 3 with an ordering of vertices $v_1,\ldots,v_n$ and an integer $1\le B\le n$, we define the B-block graph of G by the following:*

vertex partition *We partition V into $\lceil n/B\rceil$ blocks $V_0,\ldots,V_{\lfloor n/B\rfloor}$ following the given vertex ordering, i.e. with $V_i=\{v_{iB+1},\ldots,v_{iB+B}\}$ for $0\le i\le\lfloor n/B\rfloor-1$ and $V_{\lfloor n/B\rfloor}$ the remaining vertices. For simplicity we may assume w.l.o.g. that B divides n. A vertex $v=v_i$ of G is thus given a a block number $b_B(v)=\lfloor i/B\rfloor$ and a relative number $r_B(v)=(i \bmod B)+1$ between 1 and B within its block.*

Algorithm 2 Sequential Block-LFMIS

Input: The B-block graph of some G with max degree 3, with an ordering $V_0,\ldots,V_{n/B}$ on vertex blocks, intra-block edges $V_i[1\ldots 3B]$ and inter-block edges $E_{i,j}[1\ldots 3B]$ (if non-empty) for $0 \le i < j \le n/B$.
Output: The lexicographically first maximal independent set S of G.
Data Str.: Boolean matrix $\texttt{notS}[V_0\ldots V_{n/B}][1\ldots B]$
Invariant: $\texttt{notS}[V_i][k] == 1$ only if the kth vertex of V_i (*i.e.* the vertex number $iB+k-1$) is known not to be in S.
begin
  for $i = 0$ **to** n/B **do** $\texttt{notS}[V_i] = 00\ldots0$
  for $i = 0$ **to** n/B **do**
    $\texttt{notS}[V_i] =$ Intra-block-update$(V_i[1\ldots 3B], \texttt{notS}[V_i])$
    for all $j > i$ with $E_{i,j}[1\ldots 3B]$ non-empty **do**
      $\texttt{notS}[V_j] =$ Inter-block-update$(E_{i,j}[1\ldots 3B], \texttt{notS}[V_i])$
  output $\{v : \texttt{notS}[b_B(v)][r_B(v)] == 0\}$
end;

inter-block edges *The vertices of block V_i have neighbors in at most $3B$ other vertex blocks V_j with $i < j$. Each such j constitutes an inter-block edge $E_{i,j}$. For each of these inter-block edges we store a vector $E_{i,j}[1\ldots 3B]$ that encodes the induced subgraph between vertices of blocks V_i and V_j. It has entries $3k-2, 3k-1, 3k$, for $1 \le k \le B$, containing the relative number of the 3 possible neighbors that the kth relative vertex in V_i has in V_j.*

intra-block edges *Intra-block edges inside a vertex block V_i are represented by a similar vector $V_i[1\ldots 3B]$.*

See Figure 1 for an example. A relative vertex number $r_B(v)$ requires $\lceil \log B \rceil$ bits of storage. Total storage *in bits* for the block graph is therefore at most

$$\Big(\underbrace{(n/B)3B}_{\#\text{ inter edges}} + \underbrace{n/B}_{\#\text{ intra edges}}\Big)\ \underbrace{3B\log B}_{\text{vector encoding}} = O(nB\log B). \tag{6}$$

So this encoding of our graph is in fact not compressed in the sense that it occupies less space than a conventional one. But if we assume in addition that

$$2B\log B < w \tag{7}$$

where w is the *word size* of our machine, we see that the number of machine words needed for this new encoding is still linear in the number of vertices (and edges).[2]

We will assume the input edges are sorted by first sort criterion being the number of the lowest numbered vertex endpoint of the edge and second sort criterion being the number of the highest numbered vertex. This will simplify the parallel processing, and this sorted edge ordering can be easily computed by a CREW-PRAM algorithm in constant time and thus this slightly modified problem remains P-complete.

Taking a B-block graph as input we still have a simple sequential linear-time algorithm for LFMIS, but now with a potential for parallelization in the innermost **for**-loop, see Algorithm 2. The subroutines Intra-block-update and Inter-block-update in that algorithm are quite distinct. Intra-block-update is no simpler than the original LFMIS problem, and could for example be handled by the standard algorithm restricted to the subgraph induced by the vertex block. On the other hand, Inter-block-update has no dependency constraints coming from the vertex ordering, as we simply have to find the vertices of block V_j that have a neighbor in $V_i \cap S$.

4 The parallel algorithm

We now consider a CREW PRAM implementation of Algorithm 2, see Algorithm 3. The representation of the block graph will be computed in a pre-processing step that we discuss in Section 4. Moreover, the subroutine

[2] The utility of the constant 2 will only appear later.

Algorithm 3 Parallel LFMIS with processors $P_0,\ldots,P_{p-1}, B = p = \lfloor \log^\delta n \rfloor$.

Input: A graph $G=(V,E)$ of max degree 3 with an ordering $v_1,\ldots,v_n$ on V.
Output: The lexicographically first maximal independent set S.
Data Str.: Boolean matrix $\texttt{notS}[V_0\ldots V_{n/B}][1\ldots B]$ with $\texttt{notS}[V_i][k] == 1$ only if kth vertex of $V_i \notin S$.
Vectors for intra-block edges $V_i[1\ldots 3p]$ and inter-block edges $E_{i,j}[1\ldots 3p]$ (if non-empty) for $0 \le i < j \le n/p$.
Tables $Intra[1\ldots n/p]$ and $Inter[1\ldots n/p]$ giving instructions for Intra-block-update and Inter-block-update.

```
begin
    Compute the p-block graph of G, see Section 4.
    Compute lookup tables Intra and Inter, see Section 5.
    for i = 0 to n/p do P_0: notS[V_i][k] = 00...0
    for i = 0 to n/p do
        P_0: notS[V_i] = Intra[V_i[1...3B], notS[V_i]]
        foreach P_k in-parallel do
            for x = 0 to 2 do
                notS[V_{i_{xp+k}}] = Inter[E_{i,i_{xp+k}}[1...3B], notS[V_i]]
    foreach P_k in-parallel do
        for x = 0 to n/p^2 - 1 do
            for j = 1 to p do
                if notS[V_{xp+k}][j] == 0 then output v_{(xp+k)p+j}
end;
```

calls Intra-block-update and Inter-block-update will be handled by simple table lookups, and these two tables will also be computed in a pre-processing step discussed in Section 5. The index to the tables will be the parameters for the subroutine calls, namely: $V_i[1\ldots 3B]$ (where each entry has $\log B$ bits) plus $\texttt{notS}[V_i][1\ldots B]$ (with boolean entries) for Intra-Block and $E_{i,j}[1\ldots 3B]$ plus $\texttt{notS}[V_i][1\ldots B]$ for Inter-Block. These indices consist of $3B\log B + B$ bits which by assumption (7) fit into one word of our machine.

We choose the block-size equal to the number of processors, $p = B$. To ensure that we can compute the lookup tables in $O(n/p)$ time, we must constrain the table size to n/p, thus

$$(2p+1)\log p \le \log n \tag{8}$$

Constraints (7) and (8) are both met with the granularity condition (1).

Thus the *Intra* and *Inter* tables will have about n/p entries each. Using table lookup, the initialization of all n/p entries of the notS vector and all n/p intra-block updates are done in $O(n/p)$ time by a single processor. For the inter-block edges, there are at most $3p$ such edges out of block V_i, going to at most $3p$ distinct blocks in increasing order $V_{i_0}, V_{i_1}, \ldots, V_{i_{3p-1}}$ and processor P_k, $0 \le k < p$ will be responsible for those going to $V_{i_k}, V_{i_{p+k}}, V_{i_{2p+k}}$.

For the example graph in Figure 1, when handling inter-block edges from V_0, processor P_0 will first update $\texttt{notS}[V_1]$, since $V_{1_0} = V_1$, by setting

$$\texttt{notS}[V_1] = Inter\,[\langle 120\rangle \cdot \langle 000\rangle \cdot \langle 130\rangle, \langle 010\rangle] \tag{9}$$

(since $E_{0,1} = \langle 120\rangle \cdot \langle 000\rangle \cdot \langle 130\rangle$ and $\texttt{notS}[V_0] = \langle 010\rangle$) while P_1 will update $\texttt{notS}[V_2]$ since $V_{2_0} = V_2$ (there are not enough vertex blocks in the example to see the parallel scheme in full effect). The processors will lookup the inter-block update action in parallel, thus possibly reading concurrently, and then write the new information to distinct blocks.

Apart from the pre-processing involved in computing the representation of the block graph and the tables, discussed in the next section, we see that this algorithm takes time $O(n/p)$ using p processors on a CREW PRAM.

5 Pre-processing: the p-block graph

We indicate how to compute the representation of the p-block graph of G using p processors on a CREW PRAM in time $O(n/p)$, see Algorithm 4. Processor $P_k, 0 \le k \le n/p$ will be uniquely responsible for the n/p^2

Algorithm 4 Compute p-block graph with processors $P_0,\ldots,P_{p-1}, p = \lfloor \log^\delta n \rfloor$

Input: A graph $G = (V,E)$ of max degree 3 with an ordering $v_1,\ldots,v_n$ on V.
Data Str.: Vectors for intra-block edges $V_i[1\ldots 3p]$,
and inter-block edges $E_{i,j}[1\ldots 3p]$ (if non-empty) for $0 \le i < j \le n/p$.
begin
  foreach P_k **in-parallel do**
    Radix-Sort the edges.
    for $i = 0$ **to** $n/p^2 - 1$ **do**
      foreach edge $e = v_a v_b$ with $a < b$ and $a \in V_{kn/p^2+i}$ **do**
        compute block number j of v_b and relative numbers of va and v_b
        if $j = kn/p^2 + i$, i.e. e is an intra-block edge **do** update $V_{kn/p^2+i}[1\ldots 3p]$
        else do
          if e is the first inter-block edge $V_{kn/p^2+i}, V_j$ do initialize $E_{kn/p^2+i,j}$ to 0-vector
          update $E_{kn/p^2+i,j}$ in correct position by e
end;

blocks with contiguous indices $kn/p^2, kn/p^2+1, \ldots, kn/p^2+n/p^2-1$ thus avoiding any write conflicts. In a first preprocessing we sort input edges by first sort criterion being the block number of the lowest numbered vertex endpoint of the edge and second sort criterion being the block number of the highest numbered vertex. By using radix-sort, this sorting can be done in time $O(n/p)$ on each processor. A single processor will go through all the at most $3p$ edges out of a block and will spend constant time per edge for total time $O(n/p)$.

When processing edges out of a block V_i, say an edge $v_a v_b$ with $a < b$, the processor must first find the block number and relative number of v_a and v_b, and based on this information it can write to the appropriate word in memory. If this is the first edge between these two blocks initialize $E_{i,j}[1\ldots 3p]$ to the 0-vector, otherwise update $E_{i,j}[1\ldots 3p]$ in the correct bit positions using an OR-operation with the old $E_{i,j}[1\ldots 3p]$ and an appropriate mask. Consider an example: For the graph in Figure 1 when computing the intra-block edge between V_0 and V_1 a single processor will go through the edges in order 14, 15, 34, 36 and for each of these (say 15) the processor merely computes the low-endpoint block-number (0) and high-endpoint block-number (1) and low-endpoint relative number (1) and high-endpoint relative number (2), and this allows it to find the correct $< x1x2x3 >$ slot in the $E_{0,1}$ intra-block edge, and within this slot it first checks if $x1$ is 0 (assume no) then sees if $x2$ is 0 (assume yes) so it now has the appropriate mask to update $E_{0,1}[1\ldots 3p]$ in the correct bit positions using an OR-operation with the old $E_{0,1}[1\ldots 3p]$, thereby inserting the correct relative number (2) at $x2$.

6 Pre-processing: the lookup tables

Now we consider the computation of the lookup tables for block-size p,see Algorithm 5. Note that this is independent of the input graph G, except for the fact that p is chosen as a function of n such that the tables will have n/p entries. The table *Inter* has indices of the form $E_{i,j}[1\ldots 3p]$ (where each entry has $\log p$ bits) plus $\texttt{notS}[V_i][1\ldots p]$ (with boolean entries) thus consisting of $3p\log p + p$ bits total. For each boolean index of this length, we must compute the corresponding update word. The processors will each be responsible for n/p^2 entries, and can spend $O(p)$ time per entry.

For the example of Figure 1, $\texttt{notS}[V_1]$ is updated by inter-block edges from V_0 to V_1 by setting $\texttt{notS}[V_1] = \mathit{Inter}[\langle 120\rangle\cdot\langle 000\rangle\cdot\langle 130\rangle, \langle 010\rangle]$ (since $E_{0,1} = \langle 120\rangle\cdot\langle 000\rangle\cdot\langle 130\rangle$ and $\texttt{notS}[V_0] = \langle 010\rangle$). This data forces all vertices of V_1 to be not in S, thus computation of the lookup table must set $\mathit{Inter}[\langle 120\rangle\cdot\langle 000\rangle\cdot\langle 130\rangle, \langle 010\rangle] = \langle 111\rangle$. As mentioned earlier the crucial point is to find the vertices of block V_j that have a neighbor in $V_i \cap S$. In the index $\langle 120\rangle\cdot\langle 000\rangle\cdot\langle 130\rangle, \langle 010\rangle$ the second component $\langle 010\rangle$ tells us that only the first and third parts of the first component, *i.e.* $\langle 120\rangle$ and $\langle 130\rangle$ are of interest. From these we must union all numbers mentioned, and those bit positions in the output word should be set to 1. All this can be done, for each index, by $O(p)$ word operations.

For the intra-block table *Intra* the procedure is slightly more complicated, as the vertex ordering is important. Thus, for the graph in the example, the update operation $\texttt{notS}[V_0] = \mathit{Intra}[\langle 200\rangle\cdot\langle 130\rangle\cdot\langle 200\rangle, \langle 000\rangle]$

Algorithm 5 Compute lookup tables with processors $P_0,\ldots,P_{p-1}$, $p=\log^\delta n$**.**

Data Str.: Tables $Intra[1\ldots 2^{3p\log p},1\ldots 2^p]$ and $Inter[1\ldots 2^{3p\log p},1\ldots 2^p]$
begin
  foreach P_k **in-parallel do**
    for $i=0$ **to** n/p^2-1 **do**
      update $Intra[kn/p^2+i]$ and $Inter[kn/p^2+i]$
end;

accounts for edges inside block V_0. This data forces the second vertex of V_0 to be not in S, thus computation of the lookup table must set $Intra[\langle 200\rangle\cdot\langle 130\rangle\cdot\langle 200\rangle,\langle 000\rangle]=\langle 010\rangle$. Here we need a sequential traversal through the p parts of the first and second index components simultaneously. Again, this can be done using $O(p)$ word operations.

7 Conclusion

We have shown that the behavior of a problem that is 'hard' in a fine grained PRAM setting may be "easy", *i.e.* solved work-optimally compared to a sequential algorithm, if the number of processors p is restricted to some (slowly) growing function in n. This means that many more subproblems or algorithms known from sequential algorithmics could in principle be used for the design of parallel (coarse grained) algorithms than what is commonly thought. There is *e.g* no known reason why we shouldn't expect to be able to compute a DFS-tree in a coarse grained setting some day.

For each problem $\mathcal{P}$ it might also be interesting to look for the "smallest" function $N(p)$ such that there is an algorithm for $\mathcal{P}$ that is N-scalable. This might lead to a meaningful subdivision of the problem space, and give us knowledge about what can be practically handled in parallel. Certainly, for the next step in that direction it would be important to know if there are P-complete problems with work-optimal parallel algorithms where the number of processors as a function of n is

- $log^\delta n$ as in this paper but for some value $\delta>1$, or
- we can expect n^ε processors, and if
- we even might dream of $n/\log(n)$ processors.

References

1. BRENT, R. P. The parallel evaluation of generic arithmetic expressions. *Journal of the ACM 21*, 2 (1974), 201–206.
2. CULLER, D., KARP, R., PATTERSON, D., SAHAY, A., SCHAUSER, K., SANTOS, E., SUBRAMONIAN, R., AND VON EICKEN, T. LogP: Towards a Realistic Model of Parallel Computation. In *Proceeding of 4-th ACM SIGPLAN Symp. on Principles and Practises of Parallel Programming* (1993), pp. 1–12.
3. DEHNE, F., FABRI, A., AND RAU-CHAPLIN, A. Scalable parallel computational geometry for coarse grained multicomputers. *International Journal on Computational Geometry 6*, 3 (1996), 379–400.
4. GREENLAW, R., HOOVER, J., AND RUZZO, W. *Limits to parallel computation: P-completeness theory*. Oxford University Press, 1995.
5. HAWICK, K., ET AL. High performance computing and communications glossary. see `http://nhse.npac.syr.edu/hpccgloss/`.
6. KARP, R. M., AND RAMACHANDRAN, V. Parallel Algorithms for Shared-Memory Machines. In *Handbook of Theoretical Computer Science* (1990), J. van Leeuwen, Ed., vol. A, Algorithms and Complexity, Elsevier Science Publishers B.V., Amsterdam, pp. 869–941.
7. MIYANO, S. The lexicographically first maximal subgraph problems: P-completeness and NC-algorithms. *Mathematical Systems Theory* (1989).
8. VALIANT, L. G. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (1990), 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