A work-optimal coarse-grained PRAM algorithm for Lexicographically First Maximal Independent Set[⋆]

Jens Gustedt[1] and Jan Arne Telle[2]

[1] LORIA & INRIA Lorraine, France. Jens.Gustedt@loria.fr
[2] University of Bergen, Bergen, Norway. telle@ii.uib.no

Abstract. The *Lexicographically First Maximal Independent Set Problem* on graphs with bounded degree 3 is at most $\sqrt{n}$-complete, and thus very likely not parallelizable in a fine-grained setting. On the other hand, we show that in a coarse-grained setting (few processors and a lot of data) the situation is different, by giving a work-optimal algorithm on a shared memory machine for n and p such that $p \cdot \log p \in O(\log n)$.

1 Motivation and Background

It is commonly believed that not all problems are parallelizable, but what parallelizable means in practice and what theoretical models are able to capture differs somewhat. If $T^*(n)$ is the best sequential runtime on an input of size n, then with p processors the best parallel runtime we can hope for is $\Theta(T^*(n)/p)$, yielding a work-optimal algorithm. Most actual parallel computers are *coarse-grained*, having p orders of magnitude smaller than n for practical problems. However, the well-known parallel complexity class NC requires a parallel runtime polylogarithmic in n, implying $p = \Omega(T^*(n)/polylog(n))$, and this we call *fine-grained*. Showing that a problem is P-complete, meaning that it is not in NC unless P=NC, is therefore an argument for non-parallelizability only on a fine-grained computer, and may not have practical implications.

To remedy this situation Kruskal et al[10] studied parallel complexity classes EP, AP and SP that require only parallel runtime $O(T^*(n)^{1-\epsilon})$ for some $\epsilon > 0$. EP implies work-optimality, whereas AP (and SP) allows a factor polylogarithmic (and polynomial) in n away from work-optimality. Vitter et al showed that some P-complete problems indeed are parallelizable in this sense [14]. Condon[2] extended this work also with non-parallelizability results by showing, roughly, problems that could not have $O(\sqrt{n}^{1-\epsilon})$ parallel runtime unless all problems in P had a similar parallel speedup over its best sequential runtime. She showed this for the Lexicographically First Maximal Independent Set (LFMIS) problem, among others.

In the current paper, we look for a positive parallel result for the LFMIS problem, for which Condon gave only the negative result. The requirement that a lexicographic ordering of vertices must be respected gives the problem an inherently sequential flavor.

[⋆] Research partially supported by the "Pôle rgional lorrain *Intelligence Logicielle*" and a visiting grant by *Région Lorraine* for Jan Arne Telle

Hardly any results are known on parallel algorithms for LFMIS, whereas efficient parallel solutions exist if the ordering requirements are dropped, see e.g Ferreira and Schabanel [5] and Gebremedhin et al. [6]. Uehara [12] gave an NC algorithm for LFMIS restricted to graphs with a $polylog(n)$ bound on the length of the longest path respecting the lexicographic order given in the input.

We instead focus on a variant of LFMIS that is known to remain P-complete [11], that we call LFMIS3, where input graphs have bounded degree 3. We first extend Condon's result on the hardness of LFMIS to LFMIS3. Then, our main result is stated as follows:

Theorem 1. *There is a parallel algorithm for LFMIS3 that is work-optimal for all n and p such that $p \cdot \log p \in O(\log n)$.*

The paper is organized as follows. In the following section, we will introduce the models of parallel computation that will be used throughout the paper. We also provide a brief discussion about the complexity issues involved. Then Section 3 will introduce the problem and our main technique to handle it which we call *block graph*. The parallel algorithm itself is described in Section 3, followed by two sections that discuss the two different phases of preprocessing that this parallel algorithm needs. The main part of the paper being presented for a PRAM, in Section 7 we find it convenient to outline how the assumption of using a shared random access memory might be relaxed. This is done by specifying a concrete communication pattern between different processors.

2 Parallel machine models and performance measures

Although our research was guided by the more practical and realistic coarse grained machine models for parallel computation, see [13, 3, 4, 7], 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. At the end of this paper we will indicate how our results can be extended to a distributed coarse-grained setting.

We use a convenient modification of the classical CREW-PRAM[1], see [9] for an overview. The algorithm that we give will use bit-parallelism so we have to be more precise about the "RAM" part of the machine description. We will assume that each processor is a word-RAM with word size w and that it supports all conventional operations (*e.g* memory access, addition, subtraction, bitwise `and` and `or`) on machine words *in constant time*.

The choice of a word-RAM as a base for the machine model is in contrast to some of the complexity theoretic work cited above, e.g [2]. The difference in the performance measures as presented hereafter when using a more restricted RAM (so mainly considering *bit complexity*) would be a w-factor on time and cost. To allow for a fair comparison, that factor would have to be taken into account for both sequential and parallel algorithms. So, as long as we handle such a factor consistently when measuring

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

the speed up of a parallel algorithm over a sequential one this effect would cancel out. Thus we may choose this variant of the PRAM for convenience.

We will 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.*

$$\log n \leq w. \tag{1}$$

The main performance measures of a PRAM algorithm $\mathcal{A}$ are its parallel *running time* $T_{\mathcal{A}}(n)$ and its *overall work* or *cost* $C_{\mathcal{A}}(n) = T_{\mathcal{A}}(n)p_{\mathcal{A}}(n)$, where $p_{\mathcal{A}}(n)$ denotes the number of processors used by $\mathcal{A}$. 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, see [1], to fewer processors.

An inconvenience of requiring work-optimality is the fact that the definition depends on $\mathcal{A}^*$, an optimal sequential algorithm that for a specific problem may not be known. But for this paper we luckily avoid this since the problem we will consider has a linear-time sequential algorithm. So for work-optimality it will be sufficient to prove a linear upper bound on $C_{\mathcal{A}}(n)$.

When we want algorithms that are scalable for a range of processors their running time $T_{\mathcal{A}}(p, n)$ and cost $C_{\mathcal{A}}(p, n)$ become also functions of p, the number of processors. In this paper we aim for an algorithm $\mathcal{A}$ that will be work-optimal for n and p such that $p \cdot \log p \in O(\log n)$. To be more precise on the quantifications there are constants $1 \geq \delta, e > 0$ and p_0 (all independent from n and p) such that for all n and p that fulfill $p \geq p_0$ and

$$p \log p \leq \delta \log n. \tag{2}$$

we have

$$C_{\mathcal{A}}(p, n) \leq e T_{\mathcal{A}^*}(n) \tag{3}$$

or equivalently

$$T_{\mathcal{A}}(p, n) \leq e \frac{T_{\mathcal{A}^*}(n)}{p}. \tag{4}$$

To ease the design and presentation of our algorithm $\mathcal{A}$ we will consider p as being determined by n, namely maximal p such that it fulfills (2). The full algorithm $\mathcal{A}'$ with input of size n and p' processors would then consist of computing the right value of p as imposed by maximizing for (2) and then simulating $\mathcal{A}$ (with n and p) via Brent's principle on p' processors. The only obstacle for the design of $\mathcal{A}'$ is the computation of p on p' processors in time $O(n/p')$ which can be done easily for what is needed here.

3 LFMIS and the Block Graph

Given a linear ordering on the vertices of a graph, the lexicographically first maximal independent set is the subset of vertices built starting from the empty set by considering vertices in the order given, and adding the considered vertex to the set if it does not have a neighbor already in the set. The problem that is treated in Theorem 1 is the following, see the book [8] for an overview.

Problem 1 (Lexicographically First Maximal Independent Set-3 (LFMIS3)).
Input: An undirected graph $G = (V, E)$ of maximum degree 3 with an ordering $v_1, \ldots, v_n$ on V such that the edges are lexicographically sorted, and a designated vertex v.
Output: Is vertex v in the lexicographically first maximal independent set?

The assumption on the ordering of the edges which is a bit more than is usually required for LFMIS has no impact on parallel complexity of the problem: because of the bounded degree of the graph it can be achieved efficiently if necessary by a linear-work algorithm: in $O(\log n)$ time on a PRAM, or in $O(n/p)$ time in a conventional coarse grained setting.

The LFMIS3 problem is P-complete even if the input graph is restricted to be planar and have maximum degree 3, see [11]. From now on we consider only graphs of maximum degree 3. We first note that the P-completeness proof given for LFMIS3 by Miyano in [11] in fact also shows that LFMIS3 is hard in the sense defined by Condon in [2].

Corollary 1. *The LFMIS3 problem is at most $\sqrt{n}$-complete for P.*

Proof. In the paper [11] LFMIS3 is shown to be P-complete by a reduction from the circuit value problem CVP, which itself was shown in the paper [2] to be at most $\sqrt{n}$-complete for P. Since the given reduction preserves the input size, i.e. the graph is linear in the circuit given as input, it follows from the results of [2] that also LFMIS3 is at most $\sqrt{n}$-complete for P. □

This corollary means, roughly, that if anyone would give a parallel algorithm for LFMIS3 with a parallel runtime $O(n^\epsilon)$ for $\epsilon < \frac{1}{2}$ then all problems in P would have a parallel algorithm with polynomial speedup over their best sequential algorithm.

We now turn to our parallel algorithm for LFMIS3. 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 a straightforward linear-time sequential algorithm, see Algorithm 1.

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.

Algorithm 1 Sequential LFMIS3

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

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

vertex partition *We partition V into $\lceil n/B \rceil$ blocks $V_0, \dots, V_{\lfloor n/B \rfloor}$ following the given vertex ordering, i.e. with $V_i = \{v_{iB+1}, \dots, v_{iB+B}\}$ for $0 \leq i \leq \lfloor n/B \rfloor - 1$ 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.*

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 pair i, j constitutes an inter-block edge $E_{i,j}$. For each of these inter-block edges we store a vector $E_{i,j}[1 \dots 3B]$ that encodes the induced subgraph between vertices of blocks V_i and V_j. It has entries $3k-2, 3k-1, 3k$, for $1 \leq k \leq B$, 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 $V_i[1 \dots 3B]$.*

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

$$\Big(\underbrace{(n/B)3B}_{\text{\# inter edges}} + \underbrace{n/B}_{\text{\# intra edges}} \Big) \underbrace{3B \log B}_{\text{vector encoding}} = O(nB \log B). \tag{5}$$

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

$$2B \log B < \log n \tag{6}$$

by our bound for the word size (1) we obtain

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

So the number of machine words needed for this new encoding is still linear in the number of vertices (and edges).

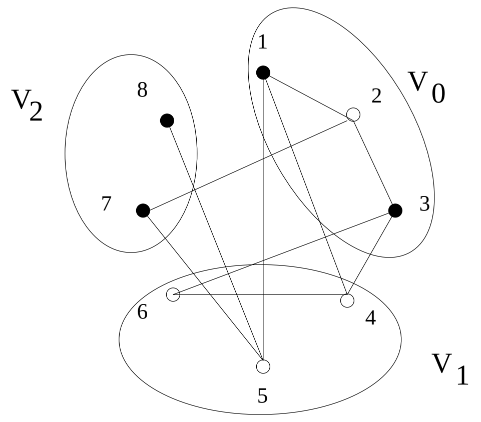

Fig. 1. The B-block of a graph with max degree 3 and vertices ordered $1 \dots 8$ for $B = 3$, *i.e.* 3 vertices per block. Black vertices are in the LFMIS.

To discuss the example of Figure 1 we will write machine words as a vector of (small) numbers. We note such words as $\langle\, a\, b\, \dots \rangle_\ell$ where $a, b, \dots$ are numbers that are written with ℓ bits and that are concatenated in the machine word. E.g

$$\begin{aligned}\Big\langle \langle\, 1\, 2\, 0\, \rangle_2 \cdot \langle\, 0\, 0\, 0\, \rangle_2 \cdot \langle\, 1\, 3\, 0\, \rangle_2 \Big\rangle_6 &= \langle\, 24 \cdot 0 \cdot 28\, \rangle_6 \\ &= \langle\, 0\,1\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1\,1\,1\,0\,0\, \rangle_1 \end{aligned} \tag{8}$$

which represents the decimal number 98332.

Inter-block edges between V_0 and V_1 are represented by a vector of $3B$ numbers in the range $0 \dots B$, namely $\Big\langle \langle\, 1\, 2\, 0\, \rangle_2 \cdot \langle\, 0\, 0\, 0\, \rangle_2 \cdot \langle\, 1\, 3\, 0\, \rangle_2 \Big\rangle_6$, with $\langle\, 1\, 2\, 0\, \rangle_2 \approx 24$ denoting that vertex 1 is adjacent to 4 and 5 (which have relative numbers 1 and 2 in V_1), $\langle\, 0\, 0\, 0\, \rangle_2 \approx 0$ denoting that vertex 2 has no adjacencies, and $\langle\, 1\, 3\, 0\, \rangle_2 \approx 28$ 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

$$\Big\langle \langle\, 2\, 0\, 0\, \rangle_2 \cdot \langle\, 1\, 3\, 0\, \rangle_2 \cdot \langle\, 2\, 0\, 0\, \rangle_2 \Big\rangle_6 \approx 132896. \tag{9}$$

Taking a B-block graph as input we still have a simple sequential linear-time algorithm for LFMIS3, 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 LFMIS3 problem, and could for example be handled by the standard algorithm restricted to the subgraph induced by the vertex block.

Inter-block-update returns a bit-vector that has a 0 in a particular position if the corresponding vertex should not be in S and a 1 otherwise. So anding the bits of $\mathsf{S}[V_j]$ with this value accumulates the constraints (of not being in S) imposed by different

Algorithm 2 Sequential Block-LFMIS3

```
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 $0 \leq i < j \leq n/B$.
Output: The lexicographically first maximal independent set $S$ of $G$.
Data Str.: Boolean matrix $\mathsf{S}[V_0 \ldots V_{n/B}][1 \ldots B]$
Invariant: $\mathsf{S}[V_i][k] == 0$ only if the $k$th 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 $\mathsf{S}[V_i] = \langle 1 \cdot 1 \cdots 1 \rangle_1$
    for $i = 0$ to $n/B$ do
        $\mathsf{S}[V_i] =$ Intra-block-update$(V_i[1 \ldots 3B], \mathsf{S}[V_i])$
        for all $j > i$ with $E_{i,j}[1 \ldots 3B]$ non-empty do
            $\mathsf{S}[V_j] = \mathsf{S}[V_j]$ bit-and Inter-block-update$(E_{i,j}[1 \ldots 3B], \mathsf{S}[V_i])$
    output $\{v : \mathsf{S}[b_B(v)][r_B(v)] == 1\}$
end;
```

neighbors. 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 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 $\mathsf{S}[V_i][1 \ldots B]$ (with boolean entries) for Intra-Block and $E_{i,j}[1 \ldots 3B]$ plus $\mathsf{S}[V_i][1 \ldots B]$ for Inter-Block. These indices consist of $3B \log B + B$ bits which by (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

$$(2p + 1) \log p \leq \log n \tag{10}$$

Constraints (7) and (10) are both met with the granularity condition (2).

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 S 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, $0 \leq k < p$ will be responsible for those going to $V_{i_k}, V_{i_{p+k}}, V_{i_{2p+k}}$.

Algorithm 3 Parallel LFMIS3 with processors P_i, $i = 0,\ldots p-1$ such that $p\lceil\log p\rceil \leq \lfloor\delta\log n\rfloor$ and $B = p$.

```
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 $\mathsf{S}[V_0 \ldots V_{n/B}][1 \ldots B]$ with $\mathsf{S}[V_i][k] == 0$ only if $k$th
    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 $0 \leq i < j \leq n/p$.
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 $P_0$: $\mathsf{S}[V_i][k] = \langle 1 \cdot 1 \cdots 1 \rangle_1$
    for $i = 0$ to $n/p$ do
        $P_0$: $\mathsf{S}[V_i] = Intra[V_i[1 \ldots 3B], \mathsf{S}[V_i]]$
        foreach $P_k$ in-parallel do
            for $x = 0$ to $2$ do
                $\mathsf{S}[V_{i_{xp+k}}] = \mathsf{S}[V_{i_{xp+k}}]$ bit-and $Inter[E_{i,i_{xp+k}}[1 \ldots 3B], \mathsf{S}[V_i]]$    (*)
    foreach $P_k$ in-parallel do
        for $x = 0$ to $n/p^2 - 1$ do
            for $j = 1$ to $p$ do
                if $\mathsf{S}[V_{xp+k}][j] == 1$ then output $v_{(xp+k)p+j}$
end;
```

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

$$\mathsf{S}[V_1] = \mathsf{S}[V_1] \textbf{ bit-and } Inter\left[\left\langle \langle 1\,2\,0 \rangle_2 \cdot \langle 0\,0\,0 \rangle_2 \cdot \langle 1\,3\,0 \rangle_2 \right\rangle_6, \langle 1\,0\,1 \rangle_2\right] \tag{11}$$

(since $E_{0,1} = \left\langle \langle 1\,2\,0 \rangle_2 \cdot \langle 0\,0\,0 \rangle_2 \cdot \langle 1\,3\,0 \rangle_2 \right\rangle_6$ and $\mathsf{S}[V_0] = \langle 1\,0\,1 \rangle_2$) while P_1 will update $\mathsf{S}[V_2]$ 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.

Algorithm 4 Compute p-block graph with processors P_i, $i = 0, \dots p-1$ such that $p\lceil \log p \rceil \leq \lfloor \delta \log n \rfloor$

```
Input: A graph $G = (V, E)$ of max degree 3 with an ordering $v_1, \dots, v_n$ on $V$.
Data Str.: Vectors for intra-block edges $V_i[1 \dots 3p]$,
and inter-block edges $E_{i,j}[1 \dots 3p]$ (if non-empty) for $0 \leq i < j \leq n/p$.
begin
    foreach $P_k$ in-parallel do
        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 $v_a$ and $v_b$
                if $j = kn/p^2 + i$, i.e. $e$ is an intra-block edge do
                    update $V_{kn/p^2+i}[1 \dots 3p]$
                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;
```

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 $P_k, 0 \leq k \leq n/p$ will be uniquely responsible for the n/p^2 blocks with contiguous indices $kn/p^2, kn/p^2 + 1, \dots, kn/p^2 + n/p^2 - 1$ thus avoiding any write conflicts. 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 $E_{i,j}[1 \dots 3p]$ to the 0-vector, otherwise update $E_{i,j}[1 \dots 3p]$ in the correct bit positions using an OR-operation with the old $E_{i,j}[1 \dots 3p]$ 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 $(1,4), (1,5), (3,4), (3,6)$ and for each of these (say $(1,5)$) 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 $\langle\, x_1 x_2 x_3 \,\rangle_6$ slot in the $E_{0,1}$ intra-block edge, and within this slot it first checks if x_1 is 0 (assume no) then sees if x_2 is 0 (assume yes) so it now has the appropriate mask to update $E_{0,1}[1 \dots 3p]$ in the correct bit positions using an OR-operation with the old $E_{0,1}[1 \dots 3p]$, thereby inserting the correct relative number, 2, at x_2.

Algorithm 5 Compute lookup tables with processors P_i, $i = 0, \dots p-1$ such that $p\lceil \log p \rceil \leq \lfloor \delta \log n \rfloor$.

```
Data Str.: Tables $Intra[1 \dots 2^{3p\log p}, 1 \dots 2^p]$ and $Inter[1 \dots 2^{3p\log p}, 1 \dots 2^p]$
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;
```

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 $E_{i,j}[1 \dots 3p]$ (where each entry has $\log p$ bits) plus $\mathsf{S}[V_i][1 \dots p]$ (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, $\mathsf{S}[V_1]$ is updated by inter-block edges from V_0 to V_1 by setting

$$\mathsf{S}[V_1] = Inter\left[\left\langle \langle\, 1\,2\,0\,\rangle_2 \cdot \langle\, 0\,0\,0\,\rangle_2 \cdot \langle\, 1\,3\,0\,\rangle_2 \right\rangle_6, \langle\, 1\,0\,1\,\rangle_2\right] \tag{12}$$

(since $E_{0,1} = \left\langle \langle\, 1\,2\,0\,\rangle_2 \cdot \langle\, 0\,0\,0\,\rangle_2 \cdot \langle\, 1\,3\,0\,\rangle_2 \right\rangle_6$ and $\mathsf{S}[V_0] = \langle\, 1\,0\,1\,\rangle_2$). This data forces all vertices of V_1 to be not in S, thus the lookup table must be set

$$Inter\left[\left\langle \langle\, 1\,2\,0\,\rangle_2 \cdot \langle\, 0\,0\,0\,\rangle_2 \cdot \langle\, 1\,3\,0\,\rangle_2 \right\rangle_6, \langle\, 1\,0\,1\,\rangle_2\right] = \langle\, 0\,0\,0\,\rangle_2 . \tag{13}$$

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 $\left\langle \langle\, 1\,2\,0\,\rangle_2 \cdot \langle\, 0\,0\,0\,\rangle_2 \cdot \langle\, 1\,3\,0\,\rangle_2 \right\rangle_6, \langle\, 1\,0\,1\,\rangle_2$ the second component $\langle\, 1\,0\,1\,\rangle_2$ tells us that only the first and third parts of the first component, *i.e.* $\langle\, 1\,2\,0\,\rangle_2$ and $\langle\, 1\,3\,0\,\rangle_2$ are of interest. From these we must union all numbers mentioned, and those bit positions in the output word should be set to 0. 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

$$\mathsf{S}[V_0] = Intra\left[\left\langle \langle\, 2\,0\,0\,\rangle_2 \cdot \langle\, 1\,3\,0\,\rangle_2 \cdot \langle\, 2\,0\,0\,\rangle_2 \right\rangle_6, \langle\, 1\,1\,1\,\rangle_2\right] \tag{14}$$

accounts for edges inside block V_0. This data forces the second vertex of V_0 to be not in S, thus the lookup table must be set

$$Intra\left[\left\langle \langle\, 2\,0\,0\,\rangle_2 \cdot \langle\, 1\,3\,0\,\rangle_2 \cdot \langle\, 2\,0\,0\,\rangle_2 \right\rangle_6, \langle\, 1\,1\,1\,\rangle_2\right] = \langle\, 1\,0\,1\,\rangle_2 . \tag{15}$$

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 Organizing the communication

The force of the recent coarse grained parallel models (*e.g* BSP [13], CGM [4] and PRO [7]) is that they are able to account for communication more realistically than the PRAM. They assume that each processor has its private share of memory and that all information needed by more than one processor has to be communicated explicitly between the processors via messages. When doing so, they account for the sending *and* receiving of message. So to be efficient, in general an algorithm has to ensure that at any moment every processor sends out and also receives about the same amount of data. Otherwise the running time on the different processors would desequilibrate.

To fit into such a setting we have to replace the random access to memory by communication between processors. Therefore we have to design a communication pattern that is able to fulfill these constraints of sending and receiving the same amount of data at any processor and time. In fact, for most of what was described above this is easy to do: *e.g* in Algorithm 4 the processors mainly do all computation by their own. They only have to communicate the tables that they computed at the very end.

The memory access that is difficult to handle is the line (*) in Algorithm 3. We will assume that each processor P_k will assemble the values $\mathsf{S}[V_i]$ for all k such that $i = j \cdot p + k$ for some j.

The algorithm performs in steps where each processor P_k performs the following:

1. Receive a previously computed value $\mathsf{S}[V_{i'}]$ for some i' from P_{k-1}.
2. Perform the line (*) for at most three block-edges, namely such that the source block-vertex of the edge is before $V_{i''}$ and the target block-vertex is V_i.
3. Send a value $\mathsf{S}[V_{i''}]$ for some i'' to processor P_{k+1}.

Some tedious choice of the indices i', i'' and for the block-edges and some computation shows that the number of steps can be bounded to $O(n/p)$. We postpone the detailed arguments and proofs to the journal version of this paper.

8 Conclusion

We have shown that the behavior of a problem that is notoriously hard in a fine grained PRAM setting may be solved *work-optimally* compared to a sequential algorithm, if the number of processors p is restricted to some (slowly) growing function in n. This result is first of all a theoretical, we would not expect it to be efficiently implemented in a realistic setting. Nevertheless, it proves the potential of such a setting and shows that the complexity of problems can be quite different in fine grained and coarse grained settings.

Perhaps 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. A prominent example would be the computation of a DFS-tree in a coarse grained setting.

For each problem $\mathcal{P}$ it might also be interesting to look at the borderline for what function of p in n such work-optimal parallel algorithms exist, see *e.g* [7] for such an approach.

References

1. BRENT, R. P. The parallel evaluation of generic arithmetic expressions. *Journal of the ACM 21*, 2 (1974), 201–206.
2. CONDON, A. A theory of strict P-completeness. *Computational Complexity 4* (1994), 220–241.
3. 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.
4. 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.
5. FERREIRA, A., AND SCHABANEL, N. A randomized BSP/CGM algorithm for the maximal independent set. *Parallel Processing Letters 9*, 3 (2000), 411–422.
6. GEBREMEDHIN, A. H., GUÉRIN LASSOUS, I., GUSTEDT, J., AND TELLE, J. A. Graph coloring on a coarse grained multiprocessor. In *WG 2000* (2000), U. Brandes and D. Wagner, Eds., vol. 1928 of *LNCS*, Springer-Verlag, pp. 184–195.
7. GEBREMEDHIN, A. H., GUÉRIN LASSOUS, I., GUSTEDT, J., AND TELLE, J. A. PRO: a model for parallel resource-optimal computation. In *16th Annual International Symposium on High Performance Computing Systems and Applications* (2002), IEEE, The Institute of Electrical and Electronics Engineers, pp. 106–113.
8. GREENLAW, R., HOOVER, J., AND RUZZO, W. *Limits to parallel computation: P-completeness theory*. Oxford University Press, 1995.
9. 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.
10. KRUSKAL, C. P., RUDOLPH, L., AND SNIR, M. A complexity theory of efficient parallel algorithms. *Theoretical Computer Science 71*, 1 (march 1990), 95–132.
11. MIYANO, S. The lexicographically first maximal subgraph problems: P-completeness and NC-algorithms. *Mathematical Systems Theory 22*, 1 (1989), 47–73.
12. UEHARA, R. A measure for the lexicographically first maximal independent set problem and its limits. *International Journal of Foundations of Computer Science 10*, 4 (1999), 473–482.
13. VALIANT, L. G. A bridging model for parallel computation. *Communications of the ACM 33*, 8 (1990), 103–111.
14. VITTER, J. S., AND SIMONS, R. A. New classes for parallel comlexity. *IEEE Trans. Comput. 35* (1986), 403–418.