Implementing Luby's Algorithm on the Cray T3E

Jürgen Gross, Markus Lohrey

Universität Stuttgart, Institut für Informatik
Breitwiesenstr. 20–22, 70565 Stuttgart, Germany
lohreyms@informatik.uni-stuttgart.de
jngross@gmx.de

Abstract. We present an implementation of Luby's algorithm for the calculation of maximal independent sets in graphs on the Cray T3E.

1 Introduction

Due to the increasing practical availability of powerful parallel architectures, the investigation of parallel algorithms has become a major research topic in the field of theoretical computer science. A widely used model for the high level description of parallel algorithms is the PRAM-model, see e.g. [8]. A computational problem is considered to be efficiently solvable in parallel, if it can be solved in polylogarithmic time, i.e., time $O(\log^k(n))$ for a fixed $k \geq 0$, with polynomially many processors on a PRAM. The class of all these computational problems is called NC, see e.g. [16]. The development of NC-algorithms for practically and theoretically relevant problems is a major research field in theoretical computer science.

In this paper we consider the problem of calculating a maximal independent set in a given graph, briefly MIS problem. A maximal independent set in a graph is a set I of nodes such that two arbitrary nodes of I are not connected by an edge, but every node which does not belong to I is connected with a node in I. The MIS problem is of practical interest since many problems in computational geometry can be reduced to the MIS problem, see e.g. [12,10,4]. But also from a theoretical point of view the MIS problem is very important. There exists a trivial sequential linear time algorithm for the MIS problem, but for some time all attempts in designing an NC-algorithm for this problem failed. In fact it was even conjectured that the MIS problem does not belong to NC [17]. The first NC-algorithm for the MIS problem was presented in [9], but especially Luby's algorithm from [13] received a lot of attention. It was the first example of the so called derandomization technique [6,2], see [1,3,15,5] for further applications. Roughly speaking the derandomization technique is based on the transformation of a randomized NC-algorithm (which is easier to design) into a deterministic NC-algorithm by simulating the randomized algorithm in parallel for several possible outcomes of its random variables. Usually there are exponentially many different outcomes for these random variables, but under certain conditions (pairwise independence of the random variables) it is sufficient to simulate the algorithm only for a polynomially large subset of the set of all possible outcomes.

To the knowledge of the authors at present there is not very much experience in the implementation of derandomized parallel algorithms on parallel architectures. In this paper we present an implementation of the randomized and a (partially) derandomized version of Luby's algorithm on the Cray T3E. We think that the experimental results obtained from our implementation may also serve as a guideline for the implementation of other derandomized algorithms. In Section 2 we give some theoretical background on Luby's algorithm. In Section 3 we give an overview of our implementation on the Cray T3E. Finally in Section 4 we present our experimental results. This work is based on the Masters thesis of the first author [7].

2 Luby's algorithm randomized and derandomized

In this section we briefly explain Luby's algorithm and the necessary preliminaries. Our outline follows the excellent exposition in [11], where also the necessary prerequisites are explained in more detail.

Graphs In this paper a *graph* G is a pair (V, E), where V is a finite set of nodes and E is a set of edges of the form $\{u, v\}$, where $u, v \in V$, $u \neq v$. Thus we consider undirected graphs without loops and multiple edges. For a set $I \subseteq V$, with $N(I)$ we denote the set of all nodes that are incident with a node in I. The degree $d(v)$ of a node v is the number of nodes that are incident with v.

Probabilities A set of events $\mathcal{A}$ (in some probability space) is called *independent* if for every $\mathcal{B} \subseteq \mathcal{A}$ we have

$$\mathrm{Prob}\left(\bigcap \mathcal{B}\right) = \prod_{A \in \mathcal{B}} \mathrm{Prob}(A).$$

The set of events $\mathcal{A}$ is called *pairwise independent* if for all $A, B \in \mathcal{A}$ with $A \neq B$ we have $\mathrm{Prob}(A \cap B) = \mathrm{Prob}(A) \cdot \mathrm{Prob}(B)$.

Luby's algorithm Luby's algorithm is executed in stages. Each stage finds an independent set I of nodes in parallel. Then the set $I \cup N(I)$ and all edges incident to $I \cup N(I)$ are deleted from the graph. This process is repeated until the graph is empty. The final independent set is the union of all the independent sets I found in each stage. In the randomized version of the algorithm a random process is used for the selection of an independent set. More precisely a stage of the randomized Luby's algorithm consists of the following steps, where V and E are the current sets of nodes and edges, respectively, before the execution of the stage, and $n = |V|$, $m = |E|$:

1. Create a set $S \subseteq V$ of candidates as follows: In parallel for each vertex $v \in V$ include v into S with probability $\frac{1}{2d(v)}$. This can be seen as a biased coin flip, where the outcome $v \in S$ corresponds to heads and $v \notin S$ corresponds to tails.
2. In parallel for each edge $\{u, v\} \in E$, if both u and v are in S remove the node with the lower degree from S (ties are resolved arbitrary). The resulting set of nodes is I.

It can be shown that the expected number of edges that are removed from the graph after a stage is at least $\frac{m}{72}$. This has the effect that the expected value of the total number of executed stages is logarithmic in the initial number of edges.

Of course in order to generate the n biased coin flips in step 1 of a stage we need n independent random bits if we require that these coin flips are independent. Therefore we say that the above version of Luby's algorithm uses *long random numbers.* However the analysis of Luby's algorithm shows that independence of the coin flips is not really necessary, but the weaker condition of pairwise independence is already sufficient. Now Luby has shown that that in order to generate n pairwise independent coin flips only $O(\log(n))$ (independent) random bits are sufficient. This leads to a (deterministic) NC-algorithm for the MIS problem. One stage of the algorithm consists of the following steps: In parallel consider all possible $2^{O(\log(n))} = n^{O(1)}$ bit strings of length $O(\log(n))$ that represent all possible outcomes of $O(\log(n))$ random bits. Each such bit string can be used in order to generate n pairwise independent coin flips for which a stage of Luby's algorithm can be simulated. Since we expect to remove $\frac{m}{72}$ many edges, in one of the polynomially many simulations at least that many edges must be deleted. Now pick such a simulation and disregard the others.

We conclude this section with a brief outline on how to generate n pairwise independent biased coin flips with $O(\log(n))$ random bits. Let p be a prime number with $n \leq p \leq 2n$. We assume that the nodes of our graph are elements of the field $\mathbb{F}_p$ with p elements. Now for each vertex v let a_v be an arbitrary integer with $0 \leq a_v < p$ such that the fraction $\frac{a_v}{p}$ is as close as possible to $\frac{1}{2d(v)}$. Let A_v any subset of $\mathbb{F}_p$ of size a_v. In order to simulate the biased coin flip for the node v we choose elements x and y uniformly at random from $\mathbb{F}_p$. For this we need only $2\log(p) = O(\log(n))$ random bits. Now we declare the flip for vertex v to be heads if $x + v \cdot y \in A_v$ and otherwise tails. Then the probability for heads is sufficiently close to $\frac{1}{2d(v)}$ (the exact value $\frac{1}{2d(v)}$ is not necessary). Furthermore it can be shown that pairwise independence of the coin flips is guaranteed. We say that the above version of Luby's algorithm uses *short random numbers.* In the next section we will present an implementation of Luby's algorithm with short random numbers. Finally note that a completely derandomized version of Luby's algorithm with short random numbers would need $O(p^2) = O(n^2)$ many simulations at each stage. For large graphs this is not feasible.

3 An implementation on the Cray T3E

For details on the architecture of the Cray T3E see [14]. For efficiency reasons we did not use the MPI interface of the Cray T3E for the implementation of Luby's algorithm on the Cray T3E but used an macro extension of C++ for the Cray T3E by some parallel programming constructs and data types. These extensions build directly on the Cray T3E operating system UNICOS/mk. Let us briefly discuss these extensions.

The data type group A group consists of several processors that run synchronously, while different groups can work asynchronously. A group is identified by a unique group number. With the function `current_group()` the group object that executes this statement is returned. Let `g` be an object of type group. With `g.group_size()` we can obtain the number of processors in the group `g` and `g. group_id()` gives the group number of g. Finally the processors that constitute the group `g` are numbered from 0 to `g.group_size()-1`. With `g.proc_id()` the number of the processors that executes this statement relatively to the group `g` is returned. The statement proc_id() is equivalent to `current_group().proc_id()` and similarly for `group_size()` and group_id(). If a group executes an `if`-statement then all processors of the group must evaluate the condition of the `if`-statement to the same Boolean value. The same has to hold for the condition of a `while`-loop. If this is not guaranteed then the group must be split before into several single-processor groups. For this the `FORK`-construct can be used:

The FORK construct With the FORK construct it is possible to split a group into several groups. There exist three three different variants of this construct but for our outline we need only two of them. With

```
FORK(proc_id())
  {stmt}
END_FORK
```

the current group is split into `group_size()` many one-processor groups which then can operate asynchronously. More generally with

```
FORK(proc_id() / {m})
  {stmt}
END_FORK
```

the current groups is split into `group_size() / m` many groups of size m. The new group number of a processor is `proc_id() mod m` .

Shared sets With the declaration `sh_set_int M(n)` we can declare a subset of $\{0, \ldots, n-1\}$. After the declaration `M` is empty. This data type is implemented as a Boolean array, where each processor of the group `g` that has generated the object M contains `n / g.group_size()` many array entries. Shared sets can be manipulated with the following statements, the first four of them can only be executed by the group that has generated M and N. The variable x must be an integer-variable.

```
M.cardinality()  // returns |M|
M = false;       // M := {}
M = N;           // M := N
M |= N;          // M := M ∪ N
M -= x;          // M := M \ {x}
M |= x;          // M := M ∪ {x}
x <= M           // returns x ∈ M
```

In order to execute a statement sequentially for all elements of some integer set we use *iterators*. With the declaration `sh_set_int::const_iterator iter` the object iter is declared to be an iterator for integer sets. With the following program we can iterate over all elements of the integer set M.

```
sh_set_int::const_iterator iter;
FORK(proc_id())
  {
    iter = M.local_begin();
    while(iter != M.local_end())
      { iter++ } // to something with *iter
  }
END_FORK;
```

This program must be executed by the group that has generated the set M. With `M.local_begin()` and `M.local_end()` we can calculate the first and last element of M, respectively, (i.e. the first (last) array entry that is true) on a particular processor. The FORK-construct is necessary, since each processor may contain a different number of elements of M and thus the condition of the while loop may terminate at different time-points on the different processors.

Graphs With the declaration `sh_graph_ext G` we can define an extended graph G. This data type builds on a simpler data type `sh_graph`. The set of nodes and edges of G are G.V and G.E, respectively. Both are of type `sh_set_int`. The end points of an edge i ∈ G.E can be calculated with the procedure `G.edge(i,v,w)`. The degree of the node v is `G.degree(v)`. With the statement `G -= M`, where M ⊆ G.V, we can remove a set of nodes M and all incident edges from the graph G. Finally `G.Next(I,M)` sets M to the set N(I) of neighbors of I, where I ⊆ G.V.

Now we are ready to present our implementation of Luby's algorithm with short random numbers. We start with the implementation of a procedure `phase` which performs a single phase of Luby's algorithm. This procedure gets two short random numbers a and b, which correspond to the numbers x

and y from the end of Section 2, and a prime number p with $n \leq$ p $\leq 2n$ (n is the initial number of nodes). G is the graph that will be reduced and MIS is an independent set of the initial graph that will be enlarged by an independent set of G. The procedure phase can only be executed by the group that has generated G and MIS.

```
void phase(sh_graph_ext& G, sh_set_int& MIS,
           int a, int b, int p)

{
  sh_set_int                 S(G.V.size());
  sh_set_int                 I(G.V.size());
  sh_set_int                 M(G.V.size());
  int                        d, v, w, A_v;
  sh_set_int::const_iterator iter;

  // forall v ∈ V pardo: {put v with probability 1/(2d(v)) in S }
  S = false;        // S := {}
  FORK(proc_id())
    {
      iter = G.V.local_begin();
      while(iter != G.V.local_end())
        {
          v = *iter;
          d = G.degree(v);
          if (d==0)
            S |= v;   // S := S ∪ {v}
          else
            {
              // put node with probability 1/2d(v) into S:
              A_v = p / (2*d);
              if ( ((a+b*v)%p) <= A_v )
                S |= v;
            }
          iter++;    // go to next node
        } }
  END_FORK;
  // forall  {v,w} ∈ E dopar:
  //     if v,w ∈ S then remove node with smaller d(v)
  I = S;
  FORK(proc_id())
    {
      iter = G.E.local_begin();
      while(iter != G.E.local_end())
        {
          G.edge(*iter, v, w);  // obtain end points of *iter
```

```
          if (v <= S  &&  w <= S)  // if (v ∈ S ∧ w ∈ S)
            {
              if (G.degree(v) <= G.degree(w))
                I -= v;     // I := I \ {v}
              else
                I -= w;     // I := I \ {w}
            }
          iter ++;    // go to next edge
        } }
  END_FORK;
  MIS |= I;       // MIS := MIS ∪ I
  G.Next(I, M);   // M := N(I)
  M |= I;         // M := M ∪ I
  G -= M;         // G := G \ M
}
```

The procedure `phase` must be iterated until the graph is empty. This will be done by the procedure `mis`.

```
void mis(sh_graph_ext G, sh_set_int& MIS, int p)
{
  int        a,b;
  rand_gen   prand(12345); // initialize a random generator

  MIS = false;    // MIS := {}
  while(G.V.cardinality() != 0)
    {
      a = prand(p);  // 0 ≤ a < p
      b = prand(p);  // 0 ≤ b < p
      phase(G,MIS,a,b,p);
    }
}
```

Under the assumption that a communication between two processors only needs time $O(1)$, we can estimate the expected theoretical running time of the procedure `mis` by

$$t(n, m, p) = c \cdot \log(m) \left(\frac{n+m}{p} + \log(p) \right). \tag{1}$$

Here c is some constant, n is the initial number of nodes, m is the initial number of edges, and p is the number of processors. The term $c \cdot \log(m)$ is an upper bound for the number of phases. Since in each phase, the nodes and edges that are stored on a single processor must be executed sequentially, we need time $O(\frac{n+m}{p})$. The additional summand $\log(p)$ arises from the internal implementation of the data type `sh_graph_ext`: In an extended graph, the

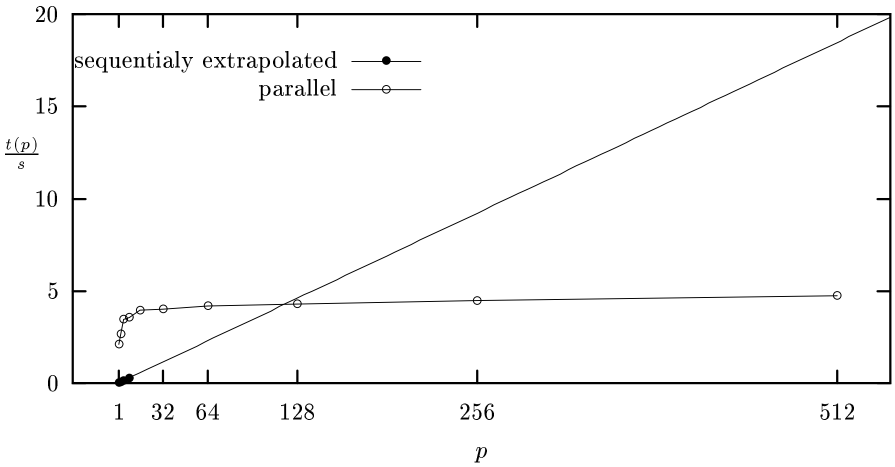

Fig. 1. Comparison of the sequential and parallel implementation

degree of each node is stored, which can be calculated in time $O(\frac{n+m}{p}+\log(p))$ with p processors, see [7]. Note that the last statement `G -= M;` in the the procedure `phase` makes a recalculation of the degrees necessary.

Finally let us mention that we also implemented a partially randomized version of Luby's Algorithm. In the implementation of this algorithm we run the procedure `phase` for several different values of the random numbers `a` and `b`, and choose the simulation that removes the most number of edges. For the implementation we used a `FORK(proc_id() / n_proc)` statement where n_proc is the number of processors that is used in a single simulation. Each group calculates their own random numbers a and b, see [7] for the details.

4 Experimental results

In Figure 1 we compare our implementation of Luby's algorithm with a (one-processor) implementation of a simple sequential algorithm on the Cray T3E on randomly generated graphs. This sequential algorithm just removes a node v together with all its neighbors from the graph and adds v to the independent set. This process is repeated until the graph is empty. In Figure 1 we run our algorithms on random graphs with $16666 \cdot p$ nodes and $50000 \cdot p$ edges, where p is the number of processors which varies between 1 and 512. Thus on each processor 16666 nodes and 50000 edges are stored. The experimental results confirm the theoretical running time in (1). For $p \geq 8$ it was not possible to run the sequential algorithm since the memory of a single processor did not suffice to store the graphs, Therefore we interpolated the running time of the sequential algorithm.

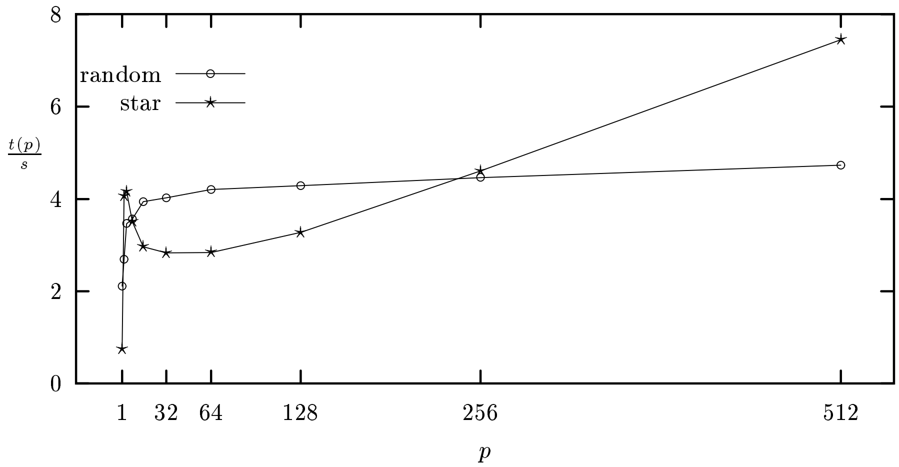

Fig. 2. Good and bad graphs

In Figure 2 we consider the running time of the parallel algorithm on different graphs. The curve "random" is taken from Figure 1. The curve "star" shows the behavior of the algorithm on a star-shaped graph which was generated as follows: Each processor contains 16666 nodes. Each node except the nodes of the first processor are connected with an arbitrary node of the first processor. Nodes on the same processors are not incident. Since all processors have to communicate with the first processor, which therefore becomes a bottle neck, for $p \geq 64$ the algorithm shows a poor linear running time. For $p < 64$ the running time decreases in a small interval. This is because if p increases more nodes will be removed in the first few phases.

In Figure 3 we investigate the dependence of the running time on the density of the graphs. The density of a graph is the quotient of the number of nodes and the number of edges. In Figure 3 the sum $n+m$ is the constant value $512 \cdot 50000$. The number n of nodes increases exponentially on the x-axis. The algorithm was run on 512 processors. Note that for very dense graphs (n small) the running time decreases. The reason for this is that the number of executed phases decreases for dense graphs.

Finally in Figure 4 we investigate the number of phases of a partially derandomized version of Luby's algorithm. We run this algorithm on a graph with 10000 nodes and 100000 edges. With s we denote the number of different simulations that are executed in parallel at each stage and from which the best one is selected, it varies between 1 and 256. The total number of phases that are executed by the algorithm is denoted by ϕ. It is interesting to note that for $s \geq 8$ the number of executed phases keeps almost constant. Note that for the fully derandomized algorithm we would need about $n^2 = 10^8$

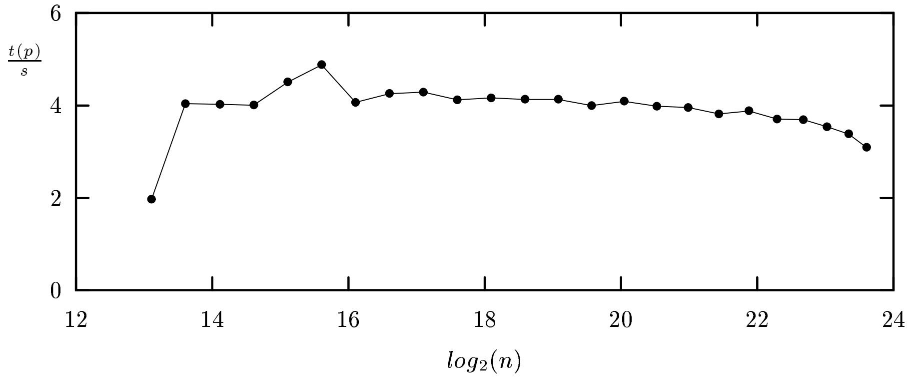

Fig. 3. Dense graphs

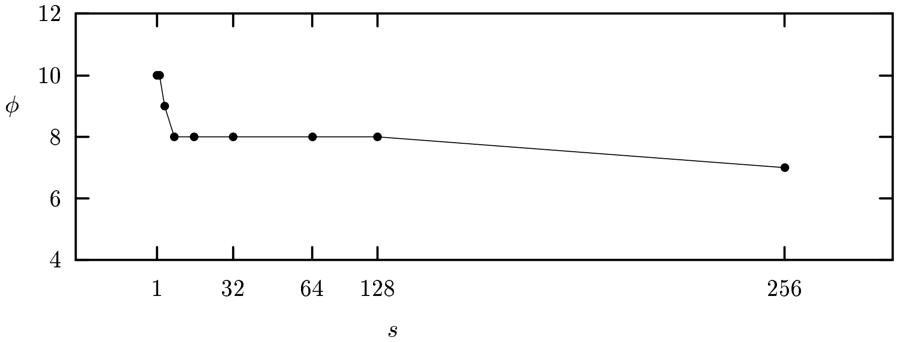

Fig. 4. Reduction of the number of phases

simulations. We conclude that in practice full derandomization is not only unfeasible but also unnecessary. A small number of parallel simulations of the randomized algorithm is completely sufficient.

References

1. N. Alon, L. Babai, and A. Itai. A fast and simple randomized parallel algorithm for the maximal independent set problem. *Journal of Algorithms*, 7:567–583, 1986.
2. A. E. Andreev, A. E. F. Clementi, and J. D. P. Rolim. A new general derandomization method. *Journal of the ACM*, 45(1):179–213, January 1998.
3. B. Chor and O. Goldreich. On the power of two–point sampling. *Journal of Complexity*, 5:96–106, 1989.

4. M. Chrobak and M. Yung. Fast algorithms for edge–coloring planar graphs. *Journal of Algorithms*, 10:35–51, 1989.
5. M. T. Goodrich and E. A. Ramos. Bounded-independence derandomization of geometric partitioning with applications to parallel fixed-dimensional linear programming. *GEOMETRY: Discrete & Computational Geometry*, 18, 1997.
6. K. Gopalakrishnan and D. R. Stinson. Derandomization. In *C. J. Colbourn and J. H. Dinitz (Editors), The CRC Handbook of Combinatorial Designs*, pages 558–560. CRC Press, 1996.
7. J. Gross. Eine Implementierung von Lubys algorithmus für die Cray T3E. Diplomarbeit 1790, Univ. Stuttgart, Fakultät Informatik, 1999. 62 pages.
8. J. JáJá. *Parallel Algorithms*. Addison Wesley, 1992.
9. R. M. Karp and A. Wigderson. A fast parallel algorithm for the maximal independent set problem. *Journal of the ACM*, 32(4):762–773, 1985.
10. D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12:28–35, 1983.
11. D. Kozen. *The Design and Analysis of Algorithms*. Springer, 1992.
12. R. J. Lipton and R. E. Miller. A batching method for coloring a planar graph. *Information Processing Letters*, 7:185–188, 1978.
13. M. Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15:1036–1053, 1986.
14. W. Oed. Technische Dokumentation, Cray Reserach, Massiv-paralleles Prozessorsystem CRAY T3E. Technical report, 1996.
15. G. E. Pantziou, P. G. Spirakis, and C. D. Zaroliagis. Fast parallel approximations of the maximum weighted cut problem through derandomization. *FST & TCS: Foundations of Software Technology and Theoretical Computer Science*, 9, 1989.
16. C. H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.
17. L. Valiant. Parallel computations. In *7th IBM Symposium on Mathematical Foundations of Computer Science*, pages 173–189, 1982.