University of Glasgow | School of Computing Science

Distributing an Exact Algorithm for Maximum Clique: maximising the costup

Ciaran McCreesh and Patrick Prosser

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Abstract

We take an existing implementation of an algorithm for the maximum clique problem and modify it so that we can distribute it over an ad-hoc cluster of machines. Our goal was to achieve a significant speedup in performance with minimal development effort, i.e. a *maximum costup*. We present a simple modification to a state-of-the-art exact algorithm for maximum clique that allows us to distribute it across many machines. An empirical study over large hard benchmarks shows that speedups of an order of magnitude are routine for 25 or more machines.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: ———————————   Signature: ———————————

Contents

Chapter 1

Introduction

Intel's tera-scale computing vision (*a parallel path to the future*) is to aim for hundreds of cores on a chip. In their white paper [9] Intel puts "programmability" at the top of the list of challenges for the multi-core era, and considers the development of multi-core software to be amongst the greatest challenges for tera-scale computing. However, Hill and Marty [10] propose that in exploiting multi-cores we should not just aim for *speedup* but also *costup*, i.e. an increase in performance that is greater than the increase in cost, be it measured in money or energy.

And that is our target, to maximise costup. We were presented with the following challenge. The second author had just completed an empirical study of exact algorithms for the maximum clique problem and a variety of algorithms had been implemented in Java [17]. Being summer, a University environment and the students away, we had access to over 100 teaching machines that we could use but not modify (limiting us to using SSH, NFS and Java). We didn't have a shared memory system with hundreds of cores but we did have access to a hundred networked PCs, but only for four weeks. Could we take one of our programs, make minimal changes to it, distribute it over the machines available and solve some big hard problems quickly? To be more precise, starting mid-week (Wednesday 4th July) could we do this by Friday (the 6th), use the available resources over the weekend (7th and 8th), solve some big hard problems and analyse the results on Monday (the 9th)? The problem studied was the *maximum clique problem*.

Definition 1 *A **simple undirected graph** G is a pair (V(G),E(G)) where V(G) is a set of vertices and E(G) a set of edges. An edge $\{u, v\}$ is in E(G) if and only if $\{u, v\} \subseteq V(G)$ and vertex u is adjacent to vertex v.*

Definition 2 *A **clique** is a set of vertices $C \subseteq V(G)$ such that every pair of vertices in C is adjacent in G.*

Clique is one of the six basic NP-complete problems given in [7]. It is posed as a decision problem [GT19]: given a simple undirected graph G = (V(G),E(G)) and a positive integer k $\leq$ |V(G)| does G contain a clique of size k or more? The optimization problems is then to find a *maximum clique*, whose size is denoted $\omega(G)$. A graph can be *coloured* by assigning colour values to vertices such that adjacent vertices take different colour values. The minimum number of different colours required is then the *chromatic number* of the graph $\chi(G)$, and $\omega(G) \leq \chi(G)$. Therefore a colouring of the graph G can be used as an upper bound on $\omega(G)$. Finding the chromatic number is NP-hard, but fast approximations exist such as [24, 1].

In the next section we describe our starting point, the algorithm MC. We then show a simple modification to the algorithm that allows distribution at various levels of granularity. Implementation details are then given followed by a report of our computational study. We then reflect, considering what we might have done differently given more time, and then conclude.

Chapter 2

An Exact Algorithm for Maximum Clique (MC)

We can address the optimization problem with an exact algorithm, such as a backtracking search [6, 18, 25, 3, 15, 14, 20, 19, 12, 22, 23, 13, 2]. Backtracking search incrementally constructs the set C (initially empty) by choosing a *candidate vertex* from the *candidate set* P (initially all of the vertices in V(G)) and then adding it to C. Having chosen a vertex the candidate set is then updated, removing vertices that cannot participate in the evolving clique. If the candidate set is empty then C is maximal (and if it is a maximum we save it) and we then backtrack. Otherwise P is not empty and we continue our search, selecting from P and adding to C.

There are other scenarios where we can cut off search, e.g. if what is in P is insufficient to unseat the champion (the largest clique found so far) then search can be abandoned. That is, an upper bound can be computed. Graph colouring can be used to compute an upper bound during search, i.e. if the candidate set can be coloured with k colours then it can contain a clique no larger than k [25, 6, 20, 12, 22, 23]. There are also heuristics that can be used when selecting the candidate vertex, different styles of search, different algorithms to colour the graph and different orders in which to do this.

For our study we use MC (**M**aximum **C**lique), Algorithm 1. MC is essentially algorithm MCSa1 in [17] and corresponds to Tomita's MCS [23] with the colour repair step removed. MCSa1 is a state of the art algorithm and a close competitor to San Segundo's BBMC [20][1]. The algorithm performs a binomial search (see pages 6 and 7 of [11]) and uses a colour cutoff. Vertices are selected from the candidate set P and added to the growing clique C. The graph induced by the vertices in P is coloured such that each vertex in P has an associated colour. Vertices are then selected in non-increasing colour order (largest colour first) for inclusion in C. Assume a vertex v is selected from P and has colour k. The graph induced by the vertices in P, including v, can be coloured with k colours and can therefore contain a clique no larger than k. Consequently if the cardinality of C plus k is no larger than that of the largest clique found so far search can be abandoned. Crucial to the success of the algorithm is the quality of the colouring of the candidate set and the time taken to perform that colouring. Empirical evidence suggests that good performance can be had with any of the three colour orderings studied in [17].

Algorithms 1 and 2 are presented as procedures that deliver a result, possibly void (line 12) and possibly a tuple (line 22). We assume that a **Set** is an order preserving structure such that when an item v is added to a **Set** S, i.e. $S \leftarrow S \cup \{v\}$, the last element in S will be v and when $S_2 \leftarrow S_0 \cap S_1$ the elements in S_2 will occur in the same order as they appear in S_0.

MC, Algorithm 1, takes as parameter a graph G and has three global variables (lines 3 to 5): integer n the number of vertices in G, C_{max} the set of vertices in the largest maximal clique found so far and integer ω_* the

---

[1] A java implementation of MCQ, MCR, MCSa, MCSb and BBMC is available at http://www.dcs.gla.ac.uk/~pat/maxClique

size of that clique. MC then calls expand (line 8) to explore the backtrack tree to find a largest clique in G. Procedure $expand$ is called (in line 8) with three arguments: the candidate set P (line 6) and the growing clique C (line 7, initially empty) and the graph G. Initially the candidate set contains all the vertices in the graph, V(G), and is sorted in non-increasing degree order (the call to $sort$ in line 6) and this order is then used for colouring the graph induced by P.

Procedure $expand$ starts by colouring the graph induced by P (step 12), delivering a pair $(S, colour)$ where S is a stack of vertices and colour is an array of colours (integers). If vertex v is at the top of the stack then vertex v has colour $colour[v]$ and all vertices in the stack have a colour less than or equal to $colour[v]$. The procedure iterates over the stack (while loop of line 13), selecting and removing a vertex from the top of the stack (line 14). If the colour of that vertex is too small then the graph induced by v and the remaining vertices in the stack and the vertices in the growing clique C will be insufficient to unseat the current champion and search can be terminated (line 15). Otherwise the vertex v is added to the clique (line 16) and a new candidate set is produced P' (line 17) where P' is the set of vertices in P that are adjacent to the current vertex v (where $N(v, G)$ delivers the *neighbourhood* of v in G), consequently each vertex in P' is adjacent to all vertices in C. If the new candidate set is empty then C is maximal and if it is larger than the largest clique found so far it is saved (line 18). But if P' is not empty C is not maximal and C can grow via the recursive call to $expand$ (line 19). Regardless, when all possibilities of expanding the current clique with the vertex v have been considered that vertex can be removed from the current clique (line 20) and from the candidate set (line 21).

Procedure $colourSort$, line 22, corresponds to Tomita's NUMBER-SORT in [22]. The vertices in P are sequentially coloured (assigned a colour number) and sorted (the vertices are delivered as a stack such that the vertices in the stack appear in descending colour order, largest colour at top of stack). In line 24 an integer array of colours is created and in line 25 an array of sets is produced, such that the set $colourClass[k]$ contains non-adjacent vertices that have colour k, i.e. $colourClass[k]$ is an *independent set*. The candidate set is iterated over in line 28, and this will be in non-increasing degree order as a consequence of the initial sorting in line 6. Line 30 searches for a colour class for the current vertex v: if any vertex in $colourClass[k]$ is adjacent to v we then look in the next colour class (increment k)[2]. In line 31 we have found a suitable $colourClass[k]$ (possibly empty) and we add v to that $colorClass$, assign that colour to the vertex (line 32) and take note of the number of colours used (line 33)[3]. Once all vertices have been added to colour classes we sort the vertices using a pigeonhole sort (lines 34 and 35): for each colour class we push the vertices in that colour class onto the stack. The procedure then finishes by returning the colour-sorted vertices with their colours (line 36) as a pair.

---

[2]$adjacent$ (lines 37 to 41) delivers $true$ if in the graph G vertex v is adjacent to a vertex in the set S.

[3]Lines 28 to 33 correspond to the sequential colouring of [24].

Algorithm 1: The sequential maximum clique algorithm MC

---

1 **Set** $MC(\textbf{Graph }G)$

2 **begin**

3   **Global** $n \leftarrow |V(G)|$

4   **Global** $C_{max} \leftarrow \emptyset$

5   **Global** $\omega_* \leftarrow 0$

6 $P \leftarrow sort(V(G), G)$

7 $C \leftarrow \emptyset$

8 $expand(C, P, G)$

9   **return** C_{max}

10 **void** $expand(\textbf{Set }C, \textbf{Set }P, \textbf{Graph }G)$

11 **begin**

12 $(S, colour) \leftarrow colourSort(P, G)$

13   **while** $S \neq \emptyset$ **do**

14 $v \leftarrow pop(S)$

15   **if** $colour[v] + |C| \leq |C_{max}|$ **then return**

16 $C \leftarrow C \cup \{v\}$

17 $P' \leftarrow P \cap N(v, G)$

18   **if** $P' = \emptyset$ **and** $|\textbf{C}| > \omega_*$ **then** $C_{max} \leftarrow C, \omega_* \leftarrow |C|$

19   **if** $P' \neq \emptyset$ **then** $expand(C, P', G)$

20 $C \leftarrow C \setminus \{v\}$

21 $P \leftarrow P \setminus \{v\}$

22 $(\textbf{Stack}, \textbf{integer}[])$ $colourSort(\textbf{Set }P, \textbf{Graph }G)$

23 **begin**

24 $colour \leftarrow new \textbf{ integer}[n]$

25 $colourClass \leftarrow new \textbf{ Set}[n]$

26 $coloursUsed \leftarrow 0$

27 $S \leftarrow new \textbf{ Stack}(\emptyset)$

28   **for** $v \in P$ **do**

29 $k \leftarrow 1$

30   **while** $adjacent(v, colourClass[k], G)$ **do** $k \leftarrow k + 1$

31 $colourClass[k] \leftarrow colourClass[k] \cup \{v\}$

32 $colour[v] \leftarrow k$

33 $coloursUsed \leftarrow max(k, coloursUsed)$

34   **for** $i \leftarrow 1$ to $coloursUsed$ **do**

35   **for** $v \in colourClass[i]$ **do** $push(S, v)$

36   **return** $(S, colour)$

37 **boolean** $adjacent(\textbf{integer }v, \textbf{Set }S, \textbf{Graph }G)$

38 **begin**

39   **for** $w \in S$ **do**

40   **if** $adjacent(v, w, G)$ **then return** $true$

41   **return** $false$

Chapter 3

Distributing MC (MCDist)

There are a number of ways we might distribute MC across p processors. We could split the problem into p parts, run each part individually and merge the results, as a map-reduce style implementation. But we cannot partition the problem into p roughly-equally sized chunks. We could instead split the problem into n parts, where n is the number of vertices in the graph, and then use a worker pool model of execution. Each job would expand a backtrack tree rooted on a node at level 1 where the current clique contained a single vertex. That is, we kick off n jobs each with a different clique of size one. But given the uneven size of the search trees, n is likely still too small to give well-balanced workloads. More generally, we could divide at level 2 potentially kicking off $\binom{n}{2}$ jobs where each process has an initial clique containing two adjacent vertices. More generally we might kick off m jobs where each job expands $\binom{n}{k}/m$ backtrack trees rooted at specified nodes at depth k. MCDist allows us to do that.

In Algorithm 2 MCDist (line 1) takes the following arguments: the graph G, a set of sets T where each element of T is of size $arity$, and integer c the size of the largest clique reported by other processes. Elements of the set T describe the nodes in the backtrack tree to be expanded by this process. For example, if $T = \{\{1\}\}$ arity would equal 1 and a call to $MCDist(G, T, 1, 0)$ would explore the backtrack tree rooted on the clique $\{1\}$.

If $T = \{\{1\}, \{2\}, \ldots, \{n\}\}$ a call to $MCDist(G, T, 1, 0)$ would be equivalent to the call $MC(G)$[1]. If $T = \{\{1, 2, 3\}, \{1, 2, 4\}, \ldots, \{1, 2, n\}, \{1, 3, 4\}, \ldots, \{n-2, n-1, n\}\}$ with $arity = 3$ and $c = k$ $MCDist$ would start expanding search from all triangles (level 3) looking for cliques of size greater than k. And finally, we might divide the set of edges $E(G)$ equally amongst the sets T_1 to T_m and distribute calls to $MCDist(G, T_i, 2, c)$, for $i \in \{1, \ldots, m\}$, across the available processors. Each call to $MCDist(G, T_i, 2, c)$ would occur within a separate job and the current best clique size c would be used at dispatch time.

In Algorithm 2 we replace expand with its distributed counterpart distExpand. The essential difference between $expand$ and $distExpand$ is the call to $considerBranch$ in line 16, i.e. the lines 16 to 20 in Algorithm 1 are now executed conditionally. Procedure $considerBranch$ determines if the current clique can be considered for expansion. If the clique is below or above the critical size then expansion can proceed, otherwise search can proceed only if $|C| = arity$ and C corresponds to a specified node of interest, i.e. $C \in T$.

Procedure $distExpand$ is similar to the search state re-computation technique used by [16] and it can be made more efficient. Assume the argument $arity$ equals α. A call to $colourSort$ is made on each call to $distExpand$ as the clique $\{v_1, \ldots, v_{\alpha-1}\}$ is incrementally constructed (repeatedly passing the test $|C| < arity$ at line 25) although it might ultimately be rejected when $C = \{v_1, \ldots, v_\alpha\}$ (failing the test $C \in T$ at line 25). In the worst case, if T was empty or contained a single tuple that did not correspond to any node in the backtrack tree, $colourSort$ would be called $O(\sum_{k=1}^{\alpha} \binom{n}{k})$ times to no effect. This is the cost of making a simple modification to MC to give us $MCDist$. However in practice much of this cost is easily avoidable and in our

---

[1]... as would a call to $MCDist(G, \{\emptyset\}, n+1, 0)$.

Algorithm 2: The distributed maximum clique algorithm MCDist

---

1 Set $MCDist($**Graph** $G,$ **Set** $T,$ **integer** $arity,$ **integer** $c)$
2 begin
3   **Global** $n \leftarrow |V(G)|$
4   **Global** $C_{max} \leftarrow \emptyset$
5   **Global** $\omega_* \leftarrow c$
6 $P \leftarrow sort(V(G), G)$
7 $C \leftarrow \emptyset$
8 $distExpand(C, P, T, arity, G)$
9   **return** C_{max}

10 void $distExpand($**Set** $C,$ **Set** $P,$ **Set** $T,$ **integer** $arity,$ **Graph** $G)$
11 begin
12 $(S, colour) \leftarrow colourSort(P, G)$
13   **while** $S \neq \emptyset$ **do**
14 $v \leftarrow pop(S)$
15   **if** $colour[v] + |C| \leq |C_{max}|$ **then return**
16   **if** $considerBranch(C, T, arity)$ **then**
17 $C \leftarrow C \cup \{v\}$
18 $P' \leftarrow P \cap N(v, G)$
19   **if** $P' = \emptyset$ **and** $|\mathbf{C}| > \omega_*$ **then** $C_{max} \leftarrow C,\ \omega_* \leftarrow |C|$
20   **if** $P' \neq \emptyset$ **then** $distExpand(C, P', T, arity, G)$
21 $C \leftarrow C \setminus \{v\}$
22 $P \leftarrow P \setminus \{v\}$

23 boolean $considerBranch($**Set** $C,$ **Set** $T,$ **integer** $arity)$
24 begin
25   **return** $|C| < arity$ **or** $|C| > arity$ **or** $C \in T$

---

studies this overhead has always been tolerable. A second improvement is to enhance $considerBranch$ such that rather than delivering true if $|C| < arity$ we deliver true if $|C| < arity$ and there exist a set $S \in T$ such that $C \subset S$. This will reduce redundant search. A further improvement is to remove elements of T after they are expanded and put a test immediately after line 11 that makes a **return** if T is empty.

Chapter 4

Implementation

Possibilities for implementation were constrained by the available resources. We intended to reuse an existing implementation of the algorithm, which was written in Java. This was convenient, since the available machines all had a JVM installed. For communication we were limited to NFS (with file-level locking only) and SSH.

The existing code was modified in line with the differences between MC and MCDist. Rather than passing T explicitly, for a graph with n vertices an integer t between 0 and $8n - 1$ was used to parameterise subproblems by splitting on the second level of the binomial search tree, as if $arity$ was 2.

A simple implementation is evident: we may number the top level nodes from $n - 1$ down to 0, from right to left (corresponding to the order in which they are popped from S). Then for a given t, every element of T contains the element we numbered $t \bmod n$. On the second level of the tree, we again label elements of S from right to left, from $|S| - 1$ down to 0, and for a given t, take those where the node's number divided by n equals t, modulo 8.

For each value of t, a file named for that number was created in a "pending" directory. These files were split by the last two digits between 100 subdirectories, to reduce contention and to keep each directory sufficiently small to avoid NFS scalability problems (initially no split was used, and lock contention prevented scalability beyond 10 machines).

The restriction to $8n - 1$ jobs was necessary to avoid having too many files, but still have significantly more jobs than machines so that they can be distributed using a worker-pool model. In addition a global "best so far" file was used to hold the value c for MCDist. This was set to contain 0 initially.

The worker programs were launched by SSH (using public key / keychain logins to avoid having to repeatedly enter passwords), with one worker program per machine. Each worker program reads in the graph, performs the initial colour ordering on the vertices, and then starts running subproblems. The worker randomly shuffles the 100 directories (to reduce contention), and then for each directory in turn, while that directory is not empty, picks a job file from that directory and moves it to a "running" directory. The worker then reads in the "best so far", runs the subproblem, saves the result to the job file and moves it to a "results" directory. The "best so far" is then updated, and another problem is attempted. Note that the "best so far" is only read in before starting any individual subproblem.

Two sets of locking are required to avoid race conditions. Firstly, a lock file is associated with each subproblem directory, to ensure that two machines cannot both start running the same problem. Here exclusive locking is used. Secondly, the "best so far" file needs to be locked to avoid races when it is being updating, and to avoid the possibility of inconsistent reads. Initially, a shared lock was used when reading in the value before launching a problem, and an exclusive lock was used when checking and updating the file afterwards. It was observed that

this was a serious limiting factor on scalability (on smaller problems, more than 50% of the total runtime was being wasted waiting for a lock). Thus, a more sophisticated mechanism for updating the file was introduced.

The value in the "best so far" file may only increase over time and its largest possible value is typically much smaller than the number of subproblems. This means most of the exclusive locks we were obtaining were in fact not being used to change the value. This suggests a better strategy: when a worker finishes a problem, it obtains a shared lock on the "best so far" file, and compares the existing value to the value it calculated. The lock is then released. If the newly calculated value is better than the existing value, then an exclusive lock is obtained (avoiding the possibility of deadlock, since the shared lock is already released). The value is then re-compared (in case it has been updated in between releasing the shared lock and obtaining the exclusive lock) and written if necessary. We observed that in practice, this mechanism substantially reduced overhead.

After execution, obtaining the results is a simple matter of checking every file in the "results" directory. (The "best so far" file only contains the size of the best clique, not its members.) The existing implementation of the algorithm produced data on the number of nodes (i.e. calls to $distExpand$) and the time spent working as well as the clique found as part of its results.

Chapter 5

Computational Study

Our study was performed over hard DIMACS[1] instances, instances from the BHOSLIB suite (Benchmarks with Hidden Optimum Solutions[2]) and Erdós-Rënyi random graphs. These instances were selected because they took between minutes and weeks on a single machine and were hard enough to cover the start up costs of distribution. Approximately 100 student lab PCs were used, running Fedora 13 with an AMD Athlon 64 X2 5200+, 4GBytes RAM and access to an NFS server. Machines were largely idle, but there was no guarantee regarding availability (several machines were switched on and off or became unavailable whilst experiments were being run). In all cases a problem was split into $8n$ jobs and these were then distributed across the machines, e.g. frb35-17-1 has 450 vertices, 3,600 jobs were produced and these were dispatched over 25 machines, then 50 machines and finally 100 machines. Run time was measured in seconds and is the difference between the wall clock time at the start of the first job and the wall clock time at the end of the last job. Clocks on the machines were loosely synchronised, sometimes differing by a couple of seconds. The single machine runtimes are for the sequential (undistributed) algorithm and exclude the read-in times for the problem instance as in [17], whereas the distributed results include program startup and read-in times.

5.1   DIMACS and BHOSLIB

Table 5.1 shows the run time in seconds to find and prove optimality using 25, 50 and 100 machines compared to the undistributed time. We list the number of vertices in the graph (n), the size of the maximum clique (ω) and when more than one machine was used the speed up (gain). The instances MANN-a45, brock400 and p-hat500-3 are from DIMACS and frb* from BHOSLIB.

Looking at the 25 machine column we see that the worst speed up was 8.0 (frb35-17-5) and the best 39.6 (frb30-15-4) and this is a super-linear speed (also seen in p-hat500-3) and should come as no surprise [21, 5][3]. This occurs because an early job terminated with a good lower bound on the clique size and this allowed subsequent jobs to terminate quickly. Instance frb30-15-1 is a "good" instance, showing an increasing speedup as we increase the number of machines. This is analysed in Figure 5.1.

In Figure 5.1 step graphs in the top row give the number of machines busy (y-axis) at a given time (x-axis). The first graph on the left is for the run with 25 machines, the middle for 50 machines and on the right 100 machines. We see a relatively sharp cut-off with nearly all machines kept busy right up to termination of the last job. We also see an increasing ramp-up cost for kicking off jobs, most notable for 100 machines, taking about 50

---

[1] Available from ftp://dimacs.rutgers.edu/pub/dsj/clique
[2] Available from http://www.nisde.buaa.edu.cn/∼kexu/benchmarks/graph-benchmarks.htm
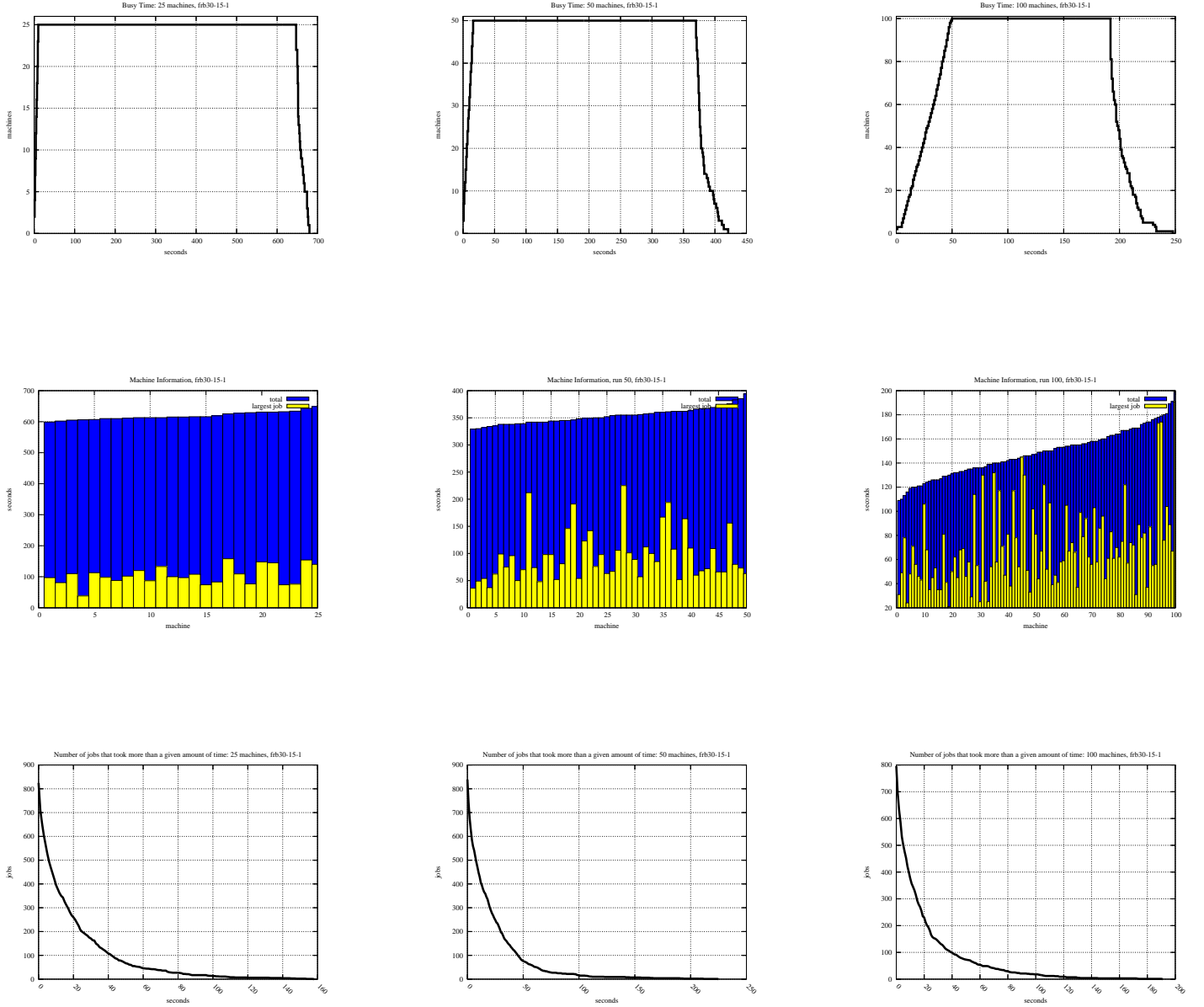[3] [5] went so far as to call this a *combinatorial implosion*.

Figure 5.1: Instance fr30-15-1 on 25, 50 and 100 machines. Top step graph shows the number of machines busy at any time. Middle histograms shows total nodes explored on each machine in blue (sorted in increasing order) and in yellow the number of nodes in longest job. Bottom contours show the number of jobs that took more than a given amount of time.

instance	n	ω	mc_1	mc_{25}	(gain)	mc_{50}	(gain)	mc_{100}	(gain)
MANN-a45	1,035	345	10,757	992	(10.8)	1,009	(10.7)	989	(10.9)
brock400-1	400	27	4,973	186	(26.7)	254	(19.6)	121	(41.1)
brock400-2	400	29	3,177	243	(13.1)	137	(23.2)	130	(24.4)
brock400-3	400	31	2,392	113	(21.2)	128	(18.7)	121	(19.8)
brock400-4	400	33	1,201	150	(8.0)	122	(9.8)	91	(13.2)
frb30-15-1	450	30	11,430	679	(16.8)	420	(27.2)	247	(46.3)
frb30-15-2	450	30	17,649	1,046	(16.9)	469	(37.6)	427	(41.3)
frb30-15-3	450	30	5,877	442	(13.3)	389	(15.1)	192	(30.6)
frb30-15-4	450	30	31,176	787	(39.6)	707	(44.1)	701	(44.5)
frb30-15-5	450	30	9,827	620	(15.9)	421	(23.3)	1,131	(8.7)
frb35-17-1	595	35	624,722	32,106	(19.5)	16,800	(37.2)	33,978	(18.4)
frb35-17-2	595	35	1,133,097	69,105	(16.4)	28,285	(40.1)	27,978	(40.5)
frb35-17-3	595	35	331,542	26,898	(12.3)	16,035	(20.7)	12,677	(26.2)
frb35-17-4	595	35	359,966	29,255	(12.3)	18,234	(19.7)	16,759	(21.5)
frb35-17-5	595	35	1,921,917	100,715	(19.1)	75,871	(25.3)	40,144	(47.9)
p-hat500-3	500	50	3,051	112	(27.2)	875	(3.5)	119	(25.6)

Table 5.1: Large hard instance: run time in seconds, using 1 to 100 machines. An entry of — corresponds to job that did not terminate after one week.

seconds to kick off the first 100 jobs, i.e. the start up phase was about 20% of the total time. The three histograms on the middle row give in yellow the length of the longest job on a machine and in blue the total of the jobs' run times on that machine. The data has been sorted by increasing total run time. For the 25 and 50 machine runs we see that no machine has been dominated by a single long job, whereas in the 100 machine case we see numerous cases where a single machine was tied up with a single long job. Why is this? When few machines are used jobs must wait to be kicked off and when they are started they have a new lower bound, whereas in the 100 machine case it is more likely that a hard job is initiated with a very small initial lower bound and this condemns that job to a long isolated execution. The three graphs at the bottom plot the number of jobs (y-axis) that took more than a given amount of time (x-axis). This shows that the majority of jobs were short (easy) and the minority were long (hard). And this is what we should expect. Many jobs had small candidate sets and relatively large lower bounds and these terminated quickly. Conversely, there were a few jobs with large candidate sets and small lower bounds and these are hard. There were also jobs that had a candidate set that contained a largest clique or a clique close to that size and it was hard to prove optimality, i.e. where we expect to find the hard problems [4, 8].

Table 5.1 shows that frb35-17-1 is *not* a "good" instance. As we increase the machines from 25 to 50 speedup improves (from 19.5 to 37.2) but then falls drastically (to 18.4) with 100 machines. This is analysed in Figure 5.2. Looking at the rightmost graphs we see at top right a long tail: after 10,000 seconds 90 machines are idle and the remaining 23,000 seconds is taken up by a few hard jobs. The rightmost (middle) histogram shows why that is so: there are three jobs that exclusively consumed all of the time on three machines. The jobs forming the long tail were started with "best so far" values of 12, 12 and 14; had these jobs been started with a "best so far" value of 30 or greater (or had they been fed that value later), they would each have terminated in under a second.

5.2   Random Instances

Table 5.2 gives results on 30 random graphs $G(n, p)$ all with $n = 1000$ vertices. The first 10 graphs have edge probability of $p = 0.1$, $p = 0.5$ for the next 10 graphs and $p = 0.6$ for the last 10. The $G(1000, 0.1)$ instances are easy on a single processor, taking less than a second. Therefore the run time on many machines is largely overhead of dispatching $n \times 8 = 8,000$ jobs and collecting their results. This demonstrates that our approach is only applicable to large hard instances where that overhead can be amortised. The $G(1000, 0.5)$ instances are relatively hard (about 20 minutes on a single machine) and the overhead pays off modestly with a speedup between 1.8 (1000-50-09 with 50 machines) and 5.6 (1000-50-00 with 100 machines). Why so modest? Analysing instance 1000-50-00 shows that none of the jobs took more than two seconds runtime yet incurred the same overhead as the easier $G(1000, 0.1)$ instances. So again, the overhead remains significant for these
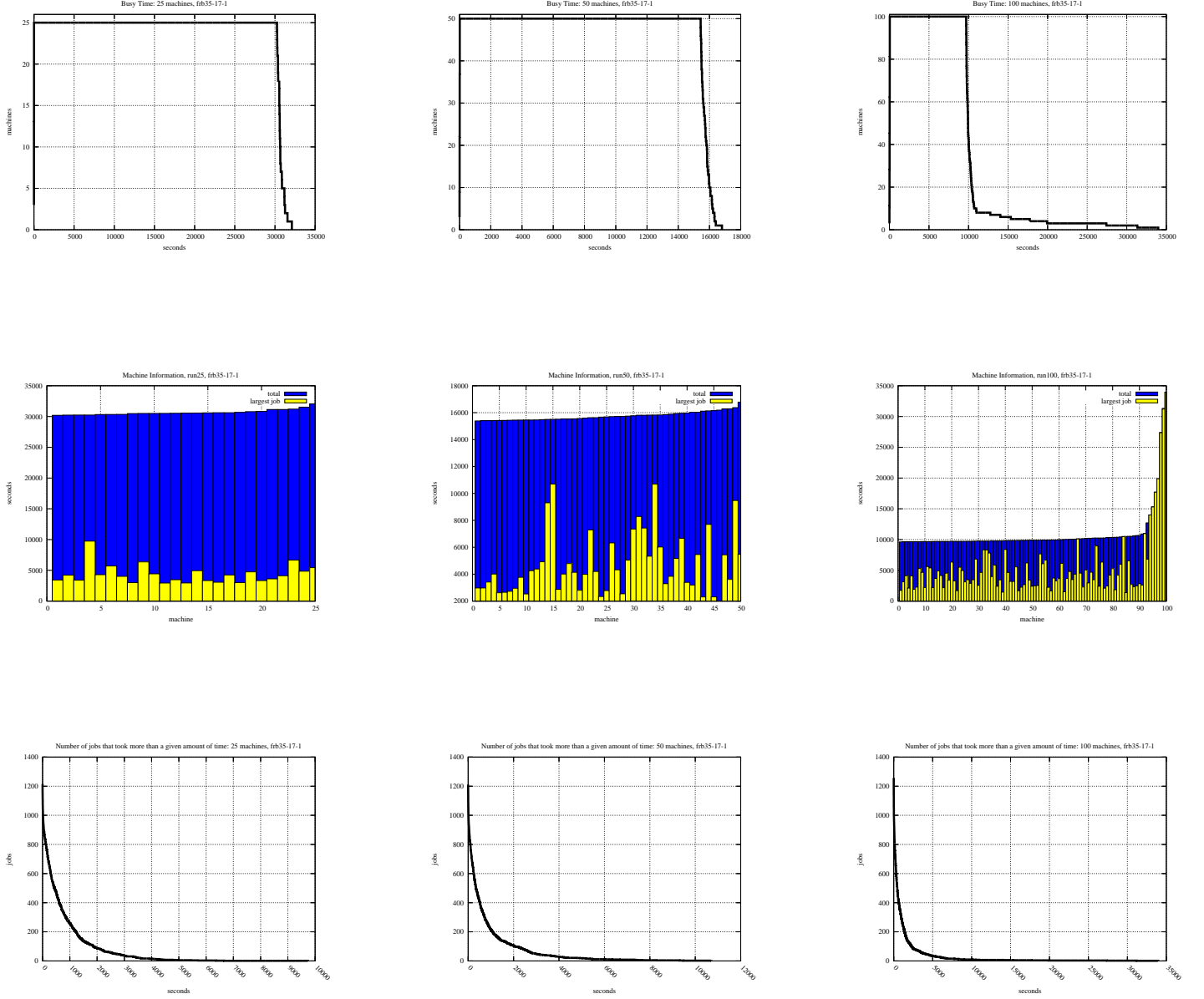
Figure 5.2: Instance fr35-17-1 on 25, 50 and 100 machines. Top step graph shows the number of machines busy at any time. Middle histograms shows total nodes explored on each machine in blue (sorted in increasing order) and in yellow the number of nodes in longest job. Bottom contours show the number of jobs that took more than a given amount of time.

instance	n	p	ω	mc_1	mc_{25}	(gain)	mc_{50}	(gain)	mc_{100}	(gain)
1000-10-00	1,000	0.1	6	0	266	(0.0)	267	(0.0)	393	(0.0)
1000-10-01			6	0	271	(0.0)	411	(0.0)	331	(0.0)
1000-10-02			6	0	274	(0.0)	261	(0.0)	255	(0.0)
1000-10-03			5	0	248	(0.0)	238	(0.0)	255	(0.0)
1000-10-04			5	0	252	(0.0)	257	(0.0)	353	(0.0)
1000-10-05			6	0	237	(0.0)	261	(0.0)	371	(0.0)
1000-10-06			6	0	233	(0.0)	263	(0.0)	264	(0.0)
1000-10-07			6	0	239	(0.0)	234	(0.0)	266	(0.0)
1000-10-08			6	0	242	(0.0)	259	(0.0)	287	(0.0)
1000-10-09			5	0	250	(0.0)	252	(0.0)	267	(0.0)
1000-50-00	1,000	0.5	15	1,380	392	(3.5)	371	(3.7)	246	(5.6)
1000-50-01			15	1,397	411	(3.4)	360	(3.9)	263	(5.3)
1000-50-02			15	1,489	421	(3.5)	285	(5.2)	276	(5.4)
1000-50-03			15	1,406	372	(3.8)	324	(4.3)	384	(3.7)
1000-50-04			15	1,491	417	(3.6)	304	(4.9)	370	(4.0)
1000-50-05			15	1,476	316	(4.7)	358	(4.1)	397	(3.7)
1000-50-06			15	1,415	283	(5.0)	401	(3.5)	325	(4.4)
1000-50-07			15	1,430	292	(4.9)	373	(3.8)	352	(4.1)
1000-50-08			15	1,458	348	(4.2)	322	(4.5)	579	(2.5)
1000-50-09			15	1,438	334	(4.3)	812	(1.8)	333	(4.3)
1000-60-00	1,000	0.6	19	58,287	2,425	(24.0)	1,959	(29.8)	—	(–)
1000-60-01			19	64,421	2,710	(23.8)	1,544	(41.7)	—	(–)
1000-60-02			20	49,135	2,486	(19.8)	1,144	(43.0)	—	(–)
1000-60-03			19	68,230	2,842	(24.0)	1,646	(41.5)	—	(–)
1000-60-04			19	59,667	2,524	(23.6)	1,596	(37.4)	—	(–)
1000-60-05			19	65,670	2,847	(23.1)	1,554	(42.3)	—	(–)
1000-60-06			19	63,603	2,807	(22.7)	1,879	(33.8)	—	(–)
1000-60-07			20	45,740	2,213	(20.7)	1,557	(29.4)	—	(–)
1000-60-08			19	61,185	2,919	(21.0)	1,469	(41.7)	—	(–)
1000-60-09			19	63,723	2,700	(23.6)	1,749	(36.4)	—	(–)

Table 5.2: Random instance: run time in seconds, using 1 to 100 machines.

instances. For $p = 0.6$ only 50 machines were available to us, nevertheless we see speedups from a minimum of 19.8 (25 machines) up to 42.3 (50 machines).

5.3 Fault Tolerance

On several occasions during execution of some of the larger problems, a machine was either powered off or rebooted. Here the NFS implementation was an advantage: the lost task could easily be identified and restarted. The "results" directory also provides an easy check that every task was in fact executed. The system is in no way tolerant of failures of the NFS server, but experience suggests that the NFS server is far more reliable than the lab machines.

Chapter 6

Potential Improvements

On reflection, there are many things we could do to improve performance if we were given more time.

Order of Task Execution: We use a random order of task execution as a simple way of reducing contention. This is unlikely to be optimal, as the initial ordering of vertices has a large effect upon performance. This also significantly affects reproducibility of results. Although reproducibility could be improved by using fixed per-machine orderings, a more sophisticated implementation that did not have to rely upon NFS would be able to dispatch jobs in the order in which they would be executed in the non-parallel version of the algorithm.

Slow Startup: With the current implementation it can take up to a minute for all 100 machines to start running. This is because SSH login attempts are deliberately rate limited. We could avoid this cost with a more sophisticated startup mechanism.

Re-reading the Best So Far: We only read and write the file containing the best clique found so far at the start and end of a task respectively. It would be better to do this more frequently. However, due to locking, this has considerable overhead on NFS. This also affects reproducibility of results: in some cases, a small delay before starting a job would lead to it having a better "best so far" value, which in turn would vastly reduce runtimes. However, some of the long tails in other problems would not be removed by this method. In some cases, a better initial "best so far" is of no help for the most time-consuming subproblems.

Finer Granularity vs Work-Stealing: To reduce the long tail in cases where re-reading "best so far" does not help, we could split the second level fully into (less than) m^2 jobs, or split on the first three levels. This was not possible with the NFS server available to us, but would be an option for better implementations. An increase in splitting is still not enough to remove every long tail, however. In some cases there are a small number of areas deep down in the tree that contribute most to the runtime. Dealing with these would need some kind of work stealing mechanism, which in turn requires much more sophisticated communication than was available.

Threading the Workers: Each of the lab machines we had available was dual core. Memory limitations prevented us from running two instances of the worker program per machine; threading the client (and adding a second set of in-process locking, since file locks are per-program rather than per-thread) could possibly give us the equivalent of doubling the number of machines, at the expense of a more complicated implementation.

Not Using Java: Java was used due to an existing implementation being available, and because of the ease of running Java programs on non-identical systems. A reimplementation in a faster language and the use of bit encoded sets (such as in [20]) would produce a substantial speed-up, at the cost of increased development time and complexity of implementation.

Chapter 7

Conclusion

So, how did we do? Did we get a *costup*? Did we get an increase in performance greater than the increase in cost? We think so. Just looking at the frb35 instances, each instance typically takes weeks to solve on a single machine. We solve frb35-17-5 in 11 hours, frb35-17-2 in under 7 hours (and over 13 days on a single machine), frb35-17-1 in about 9 hours (and more than a week on a single machine), frb35-17-4 in under 5 hours and frb35-17-3 in under 4 hours. We have spent less than a week to do this, less than a week to get more than a week's speedup.

Many of our difficulties were down to performance limitations with NFS, and these would be vastly reduced if we were using a shared memory multi-core or even just a NUMA multi-processor system. In other words, we believe that things are going to get better for this kind of costup in the future, not worse.

Bibliography

[1] Daniel Brélaz. New Methods to Color the Vertices of a Graph. *Communications of the ACM*, 22(4):251–256, 1979.

[2] Renato Carmo and Alexandre P. Züge. Branch and bound algorithms for the maximum clique problem under a unified framework. *J. Braz. Comp. Soc.*, 18(2):137–151, 2012.

[3] Randy Carraghan and Panos M. Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9:375–382, 1990.

[4] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proceedings IJCAI'91*, pages 331–337, 1991.

[5] Scott H. Clearwater, Bernardo A. Huberman, and Tad Hogg. Cooperative Solution of Constraint Satisfaction Problems. *Science*, 254(5035):1181–1183, 1991.

[6] Torsten Fahle. Simple and Fast: Improving a Branch-and-Bound Algorithm for Maximum Clique. In *Proceedings ESA 2002, LNCS 2461*, pages 485–498, 2002.

[7] M.R. Garey and D.S. Johnson. *Computers and Intractability*. W.H. Freeman and Co, 1979.

[8] Ian P. Gent, Ewan MacIntyre, Patrick Prosser, and Toby Walsh. The constrainednss of search. In *Proceedings AAAI'96*, pages 246–252, 1996.

[9] J. Held, J. Bautista, and S. Koehl. From a few cores to many: A tera-scale computing research overview. intel White Paper, 2006.

[10] M.D. Hill and M.R. Marty. Amdahl's law in the Multicore Era. *IEEE Computer*, 2008.

[11] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 3: Generating all Combinations and Permutations*. Addison-Wesley, 2010.

[12] Janez Konc and Dušanka Janežič. An improved branch and bound algorithm for the maximum clique problem. *MATCH Communications in Mathematical and Computer Chemistry*, 58:569–590, 2007.

[13] Chu Min Li and Zhe Quan. An efficient branch-and-bound algorithm based on maxsat for the maximum clique problem. In *AAAI'10*, pages 128–133, 2010.

[14] Patric R. J. Östergård. A fast algorithm for the maximum clique problem. *Discrete Applied Mathematics*, 120:197–207, 2002.

[15] Panos M. Pardalos and Gregory P. Rodgers. A Branch and Bound Algorithm for the Maximum Clique Problem. *Computers and Operations Research*, 19:363–375, 1992.

[16] L. Perron. Search procedures and parallelism in constraint programming. In *Proceedings CP 1999*, pages 346–360, 1999.

[17] P. Prosser. Exact Algorithms for Maximum Clique: a computational study. *ArXiv e-prints*, July 2012. http://arxiv.org/abs/1207.4616.

[18] Jean-Charles Régin. Using Constraint Programming to Solve the Maximum Clique Problem. In *Proceedings CP 2003, LNCS 2833*, pages 634–648, 2003.

[19] Pablo San Segundo, Fernando Matia, Diego Rodríguez-Losada, and Miguel Hernando. An improved bit parallel exact maximum clique algorithm. *Optimization Letters*, 2011.

[20] Pablo San Segundo, Diego Rodríguez-Losada, and Augustín Jiménez. An exact bit-parallel algorithm for the maximum clique problem. *Computers and Operations Research*, 38:571–581, 2011.

[21] Herb Sutter. Going Superlinear. *Dr. Dobb's Report*, January 2008.

[22] E. Tomita, Y. Sutani, T. Higashi, S. Takahashi, and M. Wakatsuki. An efficient branch-and-bound algorithm for finding a maximum clique. In *DMTC 2003, LNCS 2731*, pages 278–289, 2003.

[23] E. Tomita, Y. Sutani, T. Higashi, S. Takahashi, and M. Wakatsuki. A simple and faster branch-and-bound algorithm for finding maximum clique. In *WALCOM 2010, LNCS 5942*, pages 191–203, 2010.

[24] D.J.A. Welsh and M.B. Powell. An upper bound for the chromatic number of a graph and its application to timetabling problems. *The Computer Journal*, 10(1):85–86, 1967.

[25] David R. Wood. An algorithm for finding a maximum clique in a graph. *Operations Research Letters*, 21:211–217, 1997.