University of Glasgow | School of Computing Science

Multi-Threaded Maximum Clique

Ciaran McCreesh

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

Level 4 Project — March 21, 2013

Abstract

Deciding whether a graph contains a clique of a given size is one of the fundamental NP-complete problems. Here we discuss finding a clique of maximum size, a problem which has applications not just in computing science, but also in mathematics, biology, biochemistry, electrical engineering and communications.

We implement variations of an existing state of the art maximum clique algorithm, and show that its performance is competitive with published results. By using standard techniques for parallelism branch and bound algorithms, we present a threaded adaptation and implementation that is able to make use of the multi-core parallelism offered by modern computers. This implementation is tested on a variety of standard and random benchmarks, and its performance is compared to the sequential implementation. We show that a near-linear speedup can consistently be obtained on non-trivial problems, and that super-linear speedups are common. Furthermore, when super-linear speedup does happen, it can make some hard graph instances easy. We provide a theoretical explanation for these results, and conjecture that the techniques and results presented will generalise to other similar problems.

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

The maximum clique problem is theoretically interesting [GJ90] and practically important. Within computing science, applications include computer vision and pattern recognition; beyond, they extend to mathematics, biology, biochemistry, electrical engineering and communications [BBPP99, BW06]. Some existing algorithms for the maximum clique problem are suitable for use on dense and challenging graphs, but are sequential. Others have been presented as suitable for parallelisation, but are only able to be used with sparse graphs [PPG+12]. Here we present a shared memory parallel adaptation of a state of the art algorithm which can handle dense graphs.

In Chapter 2 we start with a simple branch and bound algorithm for the maximum clique problem and develop it into a state of the art algorithm. We then discuss using bitset encodings for a further performance increase. In Chapter 3 we review existing parallel algorithms and techniques, then develop a threaded version of our algorithm. We also analyse the potential for speedup.

Chapter 4 discusses an implementation of these algorithms, and in Chapter 5 this implementation is evaluated experimentally. We begin by showing that our sequential implementation performs competitively with published results. We then present experimental results evaluating various possible implementation choices for the threaded algorithm. With these choices made, we run our threaded implementation on a variety of standard and random benchmarks and show that we can consistently obtain close-to-linear speedups on non-trivial problems, with super-linear speedups being common. Chapter 6 concludes.

But first we begin with some background: we describe the maximum clique problem and related concepts, and discuss parallelism, with a focus on the form of homogeneous shared memory parallelism offered by a typical modern multi-core desktop computer. We also justify the importance of exploiting this parallelism.

1.1 Graphs, Cliques and Colouring

A *graph* consists of a pair of finite sets (V, E). The elements of V are known as *vertices*. The elements of E are *edges*, represented as pairs $(v_1, v_2) \in V \times V$. We call vertices v_1 and v_2 *adjacent* if $(v_1, v_2) \in E$. Throughout, we assume that our graphs are *undirected*, that is $(v_1, v_2) \in E \Rightarrow (v_2, v_1) \in E$, and contain no *loops*, that is for all $(v_1, v_2) \in E$ we have $v_1 \neq v_2$.

If $G = (V, E)$ is a graph, we may write $\mathrm{V}(G)$ for V and $\mathrm{E}(G)$ for E. For $v \in V$ we define the *neighbourhood* of v to be the vertices adjacent to v, that is $\mathrm{N}(G, v) = \{v' \in V : (v, v') \in E\}$, and the *degree* of v to be $|\,\mathrm{N}(G, v)|$.

We define the *order* of a graph $G = (V, E)$ by $|G| = |V|$. If $|G| = 0$ we say G is empty.

A graph $G' = (V', E')$ is called a *subgraph* of a graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. The subgraph *induced by* V' is the subgraph with vertices V' and all edges between those vertices. A subgraph is *induced* if it is induced by some set of vertices.

A graph $G = (V, E)$ is called *complete* if all its vertices are adjacent—that is, for every distinct pair of vertices v_1 and v_2 we have $(v_1, v_2) \in E$. A complete subgraph is known as a *clique*. We may represent a clique by the vertex set that induces it, and we define the size of the clique as the size of this vertex set. Dually, an *independent set* is a set of vertices, no two of which are adjacent.

A clique is *maximal* if it cannot be extended by adding a vertex from the main graph, and *maximum* if there is no other clique of larger size. Any maximum clique is maximal. Given a graph G, we may ask whether G contains a clique of size k. This is the *clique decision problem*. We may instead ask for a maximum clique in G (there may be more than one such clique). Determining the size of such a clique, which we call the *clique number*, is the *clique optimisation problem* or the *maximum clique problem*. We denote the clique number by $\omega(G)$, or ω where the graph is clear. The clique number of the empty graph is defined to be 0.

A *colouring* of a graph $G = (V, E)$ is a function $c : V \to C$ for some set C such that the preimage of each value in C under c is an independent set—that is, if $(v_1, v_2) \in E$ then $c(v_1) \neq c(v_2)$. We may think of the elements of C as being colours, so a colouring is an assignment of colours to vertices such that adjacent vertices are differently coloured. We say G is k-colourable if a colouring exists with $|C| \leq k$. The smallest k such that G is k-colourable is called the *chromatic number* of G, denoted $\chi(G)$ or χ. The chromatic number of the empty graph is defined to be 0.

A *greedy colouring* of a graph is a colouring obtained by considering each vertex in turn in some particular order, and assigning it the first available colour. For any non-empty graph G and greedy colouring χ^* we have the inequalities [HHM08]

$$\omega(G) \leq \chi(G) \leq \chi^*(G) \leq |G|.$$

1.2 Complexity of Clique and Colouring

The clique decision problem is NP-complete, and may be viewed as one of the "basic" problems in that class [GJ90]. The optimisation problem is NP-hard, as is approximation to within $n^{1-\varepsilon}$ for any $\varepsilon > 0$ [Zuc06].

Despite this, we are able to provide exact solutions to many large problems: in Chapter 5 we deal with graphs with up to 15,000 vertices and over 10,000,000 edges. This is because although *some* graph instances require exponential time to solve, many are in practice much easier—this phenomenon is common for hard problems [CKT91, MM11]. As a rule of thumb, most sparse graphs are "easy", and some dense graphs are "hard"; this is why we consider only being able to operate on sparse graphs to be a significant limitation for an algorithm.

Deciding whether a graph is k-colourable is also NP-complete [Kar72], and approximation to within $n^{1-\varepsilon}$ for any $\varepsilon > 0$ is NP-hard [Zuc06]. A greedy colouring may be computed in polynomial time [WP67].

1.3 Parallelism

Multi-core machines are now the norm [SL05, Sut05], and it is expected that core numbers will continue to increase [HBK06]. Our goal is to make use of multi-core parallelism to reduce the start-to-finish runtime of the algorithm—that is, to produce a *speedup*.

We defined a speedup of S as being

$$S = \frac{T_{seq}}{T_{par}},$$

where T_{seq} is the runtime for the sequential algorithm and T_{par} is parallel runtime. We call a speedup of n from n cores *linear*. A speedup of greater than n is said to be *super-linear*, and a speedup of less than 1 is called a *slowdown*.

Obtaining a speedup may require doing more overall work (i.e. computational effort). We say that a parallel algorithm that performs effectively the same amount of work as its sequential variant is *work efficient*. We consider non-work-efficient algorithms to be acceptable if this leads to faster completion. We are also not directly concerned with maximising processor utilisation—sometimes not using all available computational resources may reduce runtimes.

We do not require a parallel algorithm to produce the same answer as the sequential version—in the same way that sorting algorithms may not be stable, we only require a parallel algorithm to provide *a* correct answer, not a *particular* correct answer. For the clique optimisation problem, this translates to always finding the size of a maximum clique correctly, but possibly providing a different witness to this fact. Due to scheduling and communication nondeterminism, we do not even require that a parallel algorithm produce the same witness each time it is run.

Here we deal exclusively with homogeneous shared memory parallelism of the kind found in a typical desktop computer. We have a number of threads, each with its own context and stack, which may be run in parallel on identical[1] processing cores[2]. There is a shared pool of memory that may be accessed uniformly by each thread. We also have access to various synchronisation mechanisms to allow multiple threads to read from and write to the same section of memory.

The potential for improvement from multi-core systems is substantial. Existing work on the maximum clique problem using bitset encodings for local parallelism [SSRLJ11, SMRLH11] claims performance improvements of between a factor of two and twenty. The number of cores available on inexpensive modern hardware falls within the same range; if we can translate this to a similar speedup, we will have made a major impact. We note Karp's comment [Fre86] that

> "even though you may never be able to go from exponential to polynomial, it's also clear that there is tremendous scope for parallelism on those problems, and parallelism may really help us curb combinatorial explosions".

[1] This is somewhat complicated by hyper-threading. We discuss this issue further in Section 3.3.2.

[2] Such an environment is also offered by some systems with multiple processors rather than multiple cores on a single processor. The system used for evaluation in Chapter 5 has multiple processors each with multiple cores, but for our purposes we do not need to make any kind of distinction. For simplicity we use the term "core" to emphasise that we are talking about shared memory.

Chapter 2

Sequential Algorithms for the Maximum Clique Problem

In this chapter we start with a simple branch and bound algorithm for the maximum clique problem, and develop it into a state of the art algorithm. We then discuss using bitset encodings for a further performance increase.

2.1 A Simple Branch and Bound Algorithm

Let $G = (V, E)$ be a graph, and $v \in V$. We observe that any maximal clique in G either does not contain v, or contains only v and possibly some vertices adjacent to v. This provides the basis for the branching part of a simple branch and bound algorithm: we build up a candidate clique recursively from our choices of whether to take a vertex.

For a bound, we keep track of the size of the largest clique found so far, which we call the *incumbent*. If the size of the current candidate clique plus the size of the set of remaining undecided vertices that are adjacent to every vertex in the candidate clique is not greater than the size of the incumbent, we know we cannot find a larger clique at the current location, so we abandon search and backtrack. This leads to Algorithm 1 on the following page, which we call **mc**. Here, c is our growing candidate clique, and p (for 'potential') contains those vertices adjacent to every vertex in c. The incumbent is stored in b (for 'best').

We note that we may represent c and p as stacks, and implement the choice at ❶ by selecting the top of the stack. In this case, the two lines marked ❸ of **expand** correspond to pop operations. Empirical testing suggests that for non-trivial problems, a pre-allocated array is the best underlying data structure for this stack; improvements to this algorithm require a richer set of operations, but an array remains suitable.

We verify that ❷, where we set p' to be p intersected with the vertices adjacent to v, is sufficient to ensure that p' contains only vertices adjacent to every vertex in c: although we are only intersecting p with the neighbourhood of the vertex we just added to c, we know that p contains only vertices adjacent to every other vertex in c. Thus we do not need any further adjacency testing to ensure that p' is valid for the recursive call.

Figure 2.1 on the next page illustrates a possible sequence of recursive calls to **expand**, plus the final values of c when p is empty, that could be made by **mc** on a small graph. Nodes shown in grey are eliminated by the bound.

In practice, this algorithm is too naïve to be of use on anything but the most trivial problems. However, it forms the basis for current state of the art algorithms.

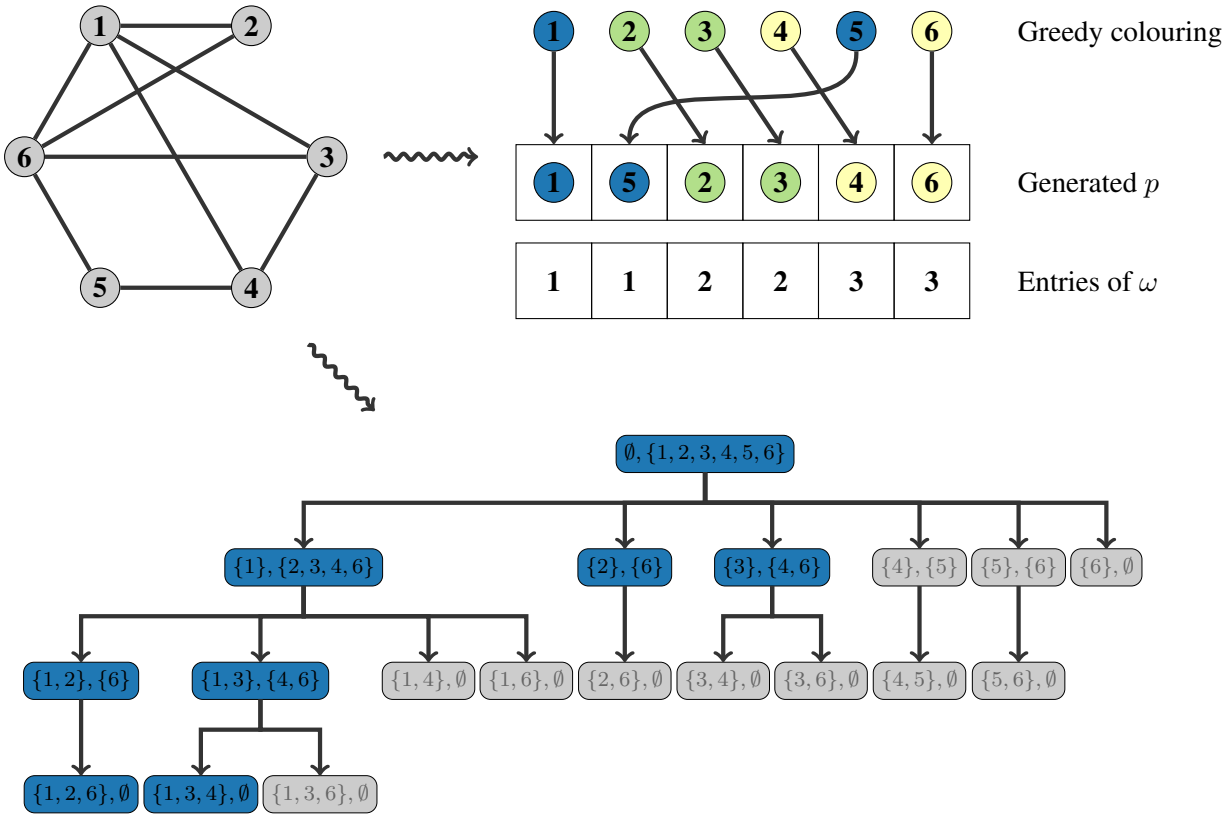
Algorithm 1: mc, a very simple maximum clique algorithm.

```
1  mc :: (Graph g) → Vertices
2  begin
3      b ← ∅
4      expand(g, ∅, V(g), b)
5      return b

6  expand :: (Graph g, Vertices c, Vertices p, Vertices b)
7  begin
8      while p ≠ ∅ and |c| + |p| > |b| do                    (1)
9          v ← a vertex chosen from p
10         c ← c ∪ {v}                                       (2)
11         p′ ← p ∩ N(g, v)
12         if p′ = ∅ then
13             if |c| > |b| then b ← c
14         else expand(g, c, p′, b)
15         c ← c \ {v}                                       (3)
16         p ← p \ {v}
```

c: Candidate clique
p: Vertices adjacent to everything in c
b: Incumbent

Figure 2.1: A small graph. Below, a possible execution tree for **mc**: boxes show values of c and p, and boxes in grey show nodes that are eliminated by the bound. To the right, a constructive colouring as produced by **colourise** from **mcsa** (Algorithm 2 on page 7).



2.2 Improving the Algorithm

The order in which vertices are selected has a profound effect upon search speed. Guaranteeing an ideal ordering is effectively as hard as the main problem; however, there are heuristics which provide good results. Here we will order vertices by non-increasing degree at the top of search.

The other major contribution to search speed is the strength of the bound. Rather than using $|p|$ as a measure of how much further our candidate clique could grow, we can get a better estimate from graph colouring. By the inequality $\omega(G) \leq \chi(G) \leq |G|$ on the remaining subgraph induced by p, we see that the chromatic number provides a bound that is as least as good as using size, and potentially better. We cannot efficiently calculate the chromatic number, but a greedy approximation is still an improvement.

The simplest greedy colouring algorithm iterates over vertices, colouring each vertex in turn with the first available colour [WP67]. As with the main algorithm, vertex ordering can have a large effect upon how close a greedy colouring is to the chromatic number. A good heuristic is to colour vertices of highest degree first. If two vertices have equal degree, we may tie-break arbitrarily, or by more sophisticated methods [Bré79]. We may recalculate the order used for colouring at every stage, or we may determine a static degree ordering at the start of search. We may also attempt to improve the colouring obtained by recolouring vertices to avoid introducing a new colour number [TSH+10]. However, more intricate methods do not necessarily yield an improvement in runtimes [Pro12].

A greedy colouring algorithm is constructive: as well as a number, it may easily be extended to give us a witness (that is, a colouring of vertices). This observation is central to Tomita's **MCQ** [TS03], **MCR** [TK07] and **MCS** [TSH+10], which make use of a constructive colouring to reduce the total number of colourings that need to be performed. This technique couples together the two improvements—we show this in Algorithm 2 on the following page, which we call **mcsa**. This corresponds to Prosser's **MCSa1** [Pro12], which is Tomita's **MCS** with a non-increasing degree ordering and without the colour repair step.

We begin by calculating a vertex ordering, which is held in the variable o throughout. This ordering is used to perform the graph colouring, which in turn gives us p at (1). Here p is a permutation of o (the orders are related, but not the same—the degree ordering influences colouring directly, and p only indirectly), and ω is an array of colour numbers, not a single value. To understand this interaction, we must look at how colouring is done in more detail. The variable k inside **colourise** (3) may be thought of as a series of coloured buckets (it is an array where the elements are a sequence of vertices, and each sequence forms an independent set). Vertices are placed in turn, using the order in o, into the first bucket of k that does not create a colour conflict. We then iterate over each item in each bucket in turn to produce an ordered p (in the manner of a pigeonhole sort), whilst recording in the result ω_i the number of colours required to colour the first i vertices of p.

We illustrate this in Figure 2.1 on the previous page. With the given graph, taking o to be in numerical order, we greedily colour vertex 1 as blue, vertices 2 and 3 as green, vertex 4 as yellow, vertex 5 as blue, and vertex 6 as yellow[1]. Flattening the buckets then brings like-coloured vertices together to produce our order for p. From this, we know not only that we can colour the entire graph using three colours, but also that we can colour the graph induced by vertices $\{1, 5, 2, 3, 4\}$ using three colours, the graph induced by $\{1, 5, 2, 3\}$ using only two colours, the graph induced by $\{1, 5\}$ using only one colour, and so on.

At (2), we must remove v from o. Unlike the removals from c and p, which may be simple stack pop operations as per Algorithm 1, v may be located anywhere inside o. We none-the-less use an array for o, and do not mind paying an $O(n)$ penalty for this removal since both n and the constant factor are small.

An observant reader may question our assertion that this algorithm is Prosser's **MCSa1**. Prosser passes both p and (what we call) o as parameters to **expand**, uses p rather than o for bound calculations, and performs

[1] Where "blue" may be "dark grey", "green" may be "medium grey" and "yellow" may be "light grey".

Algorithm 2: mcsa, a state of the art maximum clique algorithm.

```
1  mcsa :: (Graph g) → Vertices
2  begin
3      b ← ∅
4      expand(g, ∅, V(g) in non-increasing degree order, b)
5      return b

6  expand :: (Graph g, Vertices c, Vertices o, Vertices b)
7  begin
8      (ω, p) ← colourise(g, o)                          ❶
9      for i ← |p| downto 1 do
10         if |c| + ω_i > |b| then
11             v ← p_i
12             c ← c ∪ {v}
13             o′ ← o ∩ N(g, v)
14             if o′ = ∅ then
15                 if |c| > |b| then b ← c
16             else expand(g, c, o′, b)
17             c ← c \ {v}
18             p ← p \ {v}
19             o ← o \ {v}                               ❷

20 colourise :: (Graph g, Vertices o) → (Array of Integer, Vertices)
21 begin
22     k ← an empty array of sets of vertices            ❸
23     for i ← 1 to |o| do
24         l ← the first entry in k that does not contain a vertex adjacent to o_i
25         append o_i to l
26     ω ← an empty array of integers
27     p ← ∅
28     for i ← 1 to |k| do
29         for v ∈ k_i do
30             append v to p
31             append i to ω
32     return (ω, p)
```

c: Candidate clique
o: Vertices adjacent to everything in c, in order
b: Incumbent

adjacency filtering on both sets. In fact our algorithm is equivalent, but as we do not require the same degree of modularity, we may make certain simplifications and perform less work. We observe that p and o always contain the same number of elements, so we may use either for the bound. We also do not need to pass p as a parameter (and thus do not need to perform two lots of adjacency filtering): **colourise** produces p from o, and does not use the previous value of p to do so.

2.3 Bitset Encodings

It is possible to represent g, c and p as bitsets rather than arrays [SSRLJ11, SMRLH11]. This allows us to perform some parts of the algorithm using bitwise operations—for example, intersecting a vertex set with the neighbourhood of a given vertex may be treated as a bitwise-and operation. This can be viewed as a form of (local) parallelism, since we are operating on a number of vertices equal to the word size of the machine simultaneously.

We may implement Algorithm 2 using a bitset encoding, obtaining Algorithm 3, which we call **bmcsa**. This corresponds to to San Segundo's **BBMCI** [SMRLH11] with a simpler initial vertex ordering, or Prosser's **BBMC1** [Pro12]. The trace of the algorithm is unchanged—when executed, the same sequence of calls to **expand** are carried out. The difference is in the representation.

In **mcsa** we pass in a vertex ordering as a parameter to **expand**. Here we opt for a different approach: at ❶, we instead permute the entire graph. (We may also take this opportunity to re-encode g using bitsets.) This is necessary for the implementation of **colourise**.

Inside **expand** we make use of the bitset encoding at ❷ and ❹. We replace unions and intersections by bitwise-or and bitwise-and operations respectively, and removing an element from a set corresponds to unsetting a particular bit; we use the recoded graph to obtain $\mathrm{N}(g, v)$ as a bitset. We may determine $|c|$ at ❸ either by using a "population count" operation[2], or by passing an additional parameter to **expand**.

The bitset implementation of **colourise** is not a direct translation from the one of **mcsa**, and deserves explanation. We cannot permute a bitset, so as well as returning colour numbers, we return a conventionally represented vertex ordering. At ❺, ω and o are to form our result, Ω is the current colour number, and p' contains the vertices in p that we have yet to colour. While some of these vertices remain uncoloured, we pick the first of these vertices, colour it, and then try to give further vertices the same colour. The variable q at ❻ contains those vertices that have yet to be coloured, and that are not adjacent to any of the vertices to which we have already assigned the current colour. The operation at ❼ is commonly known as "find first set bit", and is often implemented directly in hardware[3]. It selects for us the next vertex that we can colour using the current colour (this is why we permute the graph at ❶). The operations marked ❽ again rely upon bitset representations—the bitwise-and is an intersection operation that removes from q all vertices adjacent to the vertex just coloured.

It is not entirely obvious that this method produces the same colouring as for **mcsa**. Rather than allocating colours to vertices in turn, we are allocating vertices to colour classes (independent sets). The reader will convince himself that although we appear to be appending vertices in a different order, we do in fact end up with the same result due to the bucket flattening done in **mcsa**.

[2] GCC makes this operation available as an intrinsic named **popcount**.
[3] GCC makes this operation available as an intrinsic named **ffs**.

Algorithm 3: bmcsa, a bitset encoding of **mcsa**.

```
1  bmcsa :: (Graph g) → Vertices
2  begin
3      b ← ∅
4      permute g so that the vertices are in non-increasing degree order        ← 1
5      expand(g, ∅, V(g), b)
6      return vertices corresponding to the set bits in b

       c: Candidate clique
       p: Vertices adjacent to everything in c
       b: Incumbent

7  expand :: (Graph g, BitSet c, BitSet p, BitSet b)
8  begin
9      (ω, o) ← colourise(g, p)
10     for i ← |p| downto 1 do
11         if |c| + ω_i > |b| then
12             v ← o_i
13             set bit v in c                                                   ← 2
14             p′ ← p ∧ N(g, v)
15             if p′ = ∅ then
16                 if |c| > |b| then b ← c                                      ← 3
17             else expand(g, c, p′, b)
18             unset bit v in c                                                 ← 4
19             unset bit v in p

20 colourise :: (Graph g, BitSet p) → (Array of Integer, Vertices)
21 begin
22     ω ← an empty array of integers
23     o ← an empty array of vertices                                           ← 5
24     p′ ← p
25     Ω ← 0
26     while p′ ≠ ∅ do
27         Ω ← Ω + 1                                                            ← 6
28         q ← p′
29         while q ≠ ∅ do                                                       ← 7
30             v ← the first set bit in q
31             unset bit v in p′
32             unset bit v in q                                                 ← 8
33             q ← q ∧ N(g, v)
34             append Ω to ω
35             append v to o
36     return (ω, o)
```

Chapter 3

Parallel Algorithms for the Maximum Clique Problem

In this chapter we review existing parallel algorithms and techniques, and develop threaded versions of our algorithm from Chapter 2. We then analyse the potential for speedup.

We may split opportunities for parallelism into two categories. On the one hand, we may exploit local or data parallelism—that is, to try speed up parts of the algorithm by implementing, say, the bound function in parallel. The bit parallelism in **bmcsa** may be considered to be of this type. Here though we discuss global parallelism—that is, we decompose the problem into smaller work units which may be executed by different threads in parallel.

3.1 Existing Parallel Algorithms for Clique-Related Problems

The maximum clique algorithm presented by Pattabiraman et. al. [PPG+12] is described as being "well-suited for parallelization", although no speedup figures are given. The algorithm is specifically designed for sparse graphs, and does very badly for dense graphs. For example, "MANN_a27" from DIMACS [DIM] is reported as taking over 10,000 seconds to solve, but we would expect **mcsa** to require only a few seconds to produce a result. The paper also claims that Tomita's **MCQ** [TS03] (a precursor to our **mcsa**) is "inherently sequential or otherwise difficult to parallelize", which we dispute.

Methods for enumerating all maximal cliques are provided by Karp and Zhang [KZ93], and by Schmidt [SSTP09]. Both focus upon sparse graphs. Methods for partitioning a graph are provided by Szabó [Sza11], but no implementation is discussed. Pullan et. al. present a cooperative local search method that scales well over multi-core processors [PMB11], but as it is an approximation method it can only be used to find a large clique, not a maximum clique.

The results of Pardalos et. al. [PRR98], and previous experiments by Prosser and the author [MP12], suggest that a parallel implementation of a state of the art algorithm is viable. The former is based upon a sequential algorithm by Carraghan [CP90], which is consistently outperformed by newer algorithms, uses MPI rather than threading (we discuss this in Section 4.1), and was not taken beyond four processors. The latter uses a "quick and dirty" approach to adapt code by Prosser [Pro12] with as little work as possible to run on a cluster of student PCs; we aim for a much more refined implementation.

3.2 Parallel Branch and Bound

To parallelise an algorithm we look for "units of work" that may be executed independently. We may view our algorithm's execution as forming a tree (in a similar way to Figure 2.1 on page 5), with each call to **expand** being a node. Each recursive call is then on "the next level down" of the tree, and each step of the loop can be viewed as going "from left to right". We can treat these nodes as potentially separate work units. Each node is dependent upon all the nodes directly above it, since the appropriate values of c and o must have been calculated. It may appear that each node is also dependent upon every node to its left, since executing one of these sibling nodes may result in our node being eliminated. However, we may ignore this dependency, and speculatively execute a node anyway. This may turn out to be wasted work, but it does not affect the correctness of the algorithm.

Thus a possible design presents itself. We have a global queue, whose entries represent nodes in the execution tree (i.e. each entry contains the values for c and o) together with all the children of that node (i.e. the recursive calls). This queue is populated with parts of the tree, and a number of worker threads each take items from the queue and execute them. For completeness, it suffices to ensure that every part of the tree that cannot be eliminated ends up being processed at least once.

For branch and bound in general this technique has been widely studied [Rou87, PR90, GC94, Gro95, BHP04, BB10]. In particular, the assumption that we may ignore the left-to-right dependency appears to be a reasonable one. Pardalos [PRR98] takes exactly this approach. Conceptually method used by McCreesh and Prosser [MP12] is similar, but rather than a queue, a fixed pool of subtrees is used; this is due to environment limitations, rather than being a genuine improvement.

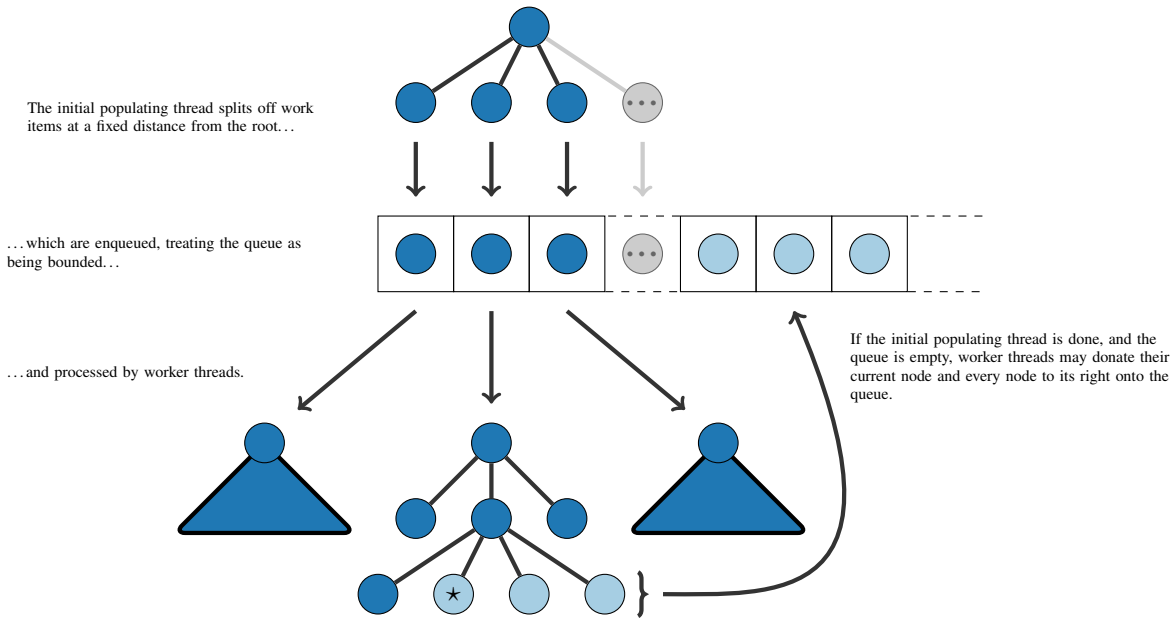
A common variation on this approach is to use task rather than thread decomposition, and avoid an explicit queue [MRR12]. This is conceptually simpler, and can be easier to implement [Lee06]. It does require efficient task cancellation, which can be problematic—using a queue and a fixed number of workers sidesteps this issue. More significantly, such an approach is incompatible with the conditions for avoiding a slowdown that we shall discuss later in this chapter. Thus we will not take this simplification.

Our primary difficulty is that our sequential algorithm typically makes a huge number of recursive calls, each node of which is reasonably cheap to process—we expect to be exceeding 100,000 nodes per second on modern hardware. We cannot expect competitive performance if we simply replace every single call to **expand** with a queue operation. Nor is it possible to pre-generate the entire tree, since the number of nodes is exponential in order of the graph. We must therefore decide what level of granularity we desire, and how the queue is to be populated.

In previous work [MP12] the search tree was split at distance 2 from the root, and then some subtrees were joined together to avoid having too many work items. The problem with such a static partition is that different subtrees can be of massively different sizes. Furthermore, there is generally no way to estimate up-front where the large subtrees are. It was shown that such a static distribution can sometimes lead to processors sitting idle. Although idle processors are not *a priori* a problem, it is easy to envision circumstances where effectively all of the work ends up on a single processor, resulting in no speedup at all.

Splitting work entirely dynamically would avoid this, but as we have many small nodes, the overheads would make this approach useless. We could instead split statically and use work stealing [BS81, KZ93, BL99], to allow idle processors to take subproblems from other threads. But this would still require each thread to make its active subproblem public. An alternative is presented by Clausen [CLT91] where overly busy processors can pass work onto other processors—this is done regardless of whether or not other processors are idle. In Tzeng et. al. [TPO10] something similar is done on a GPU, with the aim of allowing fixed size queues. We opt for a middle ground: we split statically to begin with, and then if there are idle processors later on, we advertise this fact. Other processes may then choose to "donate" parts of their work back onto the global queue.

Figure 3.1: Work splitting and queueing mechanism for **tmcsa**. Nodes correspond to a call to **expand**. We illustrate splitting at a distance $d = 1$ from the root, and work donation occurring once when the donating worker's position is at the node marked $\star$.



We assume at this stage that a single global queue will not be an excessive source of contention; if, when evaluating performance, it turns out that this assumption is overly optimistic, then we may switch to a multiple-queue method without requiring non-local design changes.

We must decide how to populate the queue initially. Although static splitting at a fixed distance from the root *sometimes* leaves processors idle, our previous work shows that it is none-the-less usually quite reasonable. Thus we retain this approach, and rely upon work donation to cover the awkward cases. We do, however, introduce a refinement. Rather than pre-populating the queue before starting worker threads, we make an additional thread that does the initial population. This allows us to start processing subtrees before population has finished. If we treat the queue as being bounded during the initial population (but not for work donation), we may reduce the amount of work done in cases where a good bound is found quickly. This also avoids the potential problem of having n^d items on the queue when splitting at distance d from the root. It remains to decide at what this value of d should be; we defer this decision until Section 3.4. We provide a general illustration of the queueing mechanism in Figure 3.1.

We must consider what data may be shared or should be shared between workers, and how if at all workers communicate the discovery of a better value of b to each other. For now we assume that b is globally visible and may safely be updated by any thread without the possibility of a race; we address this properly in Chapter 4.

This gives us Algorithm 4 on page 14, which we call **tmcsa** (for "threaded **mcsa**"). The variable b at ❶ is shared between all threads. (In practice, we share only $|b|$, and take the full value of b from the appropriate thread before joining.) At ❹ we must ensure that we are reading a valid value, which may or may not require synchronisation. At ❺ the test and update must be synchronised, to avoid the possibility of a data race resulting in a higher value of $|b|$ being overwritten by a lower value.

The condition at ❷ is subtle: the loop must continue whilst it is possible that the queue will later not be empty. Simply checking whether the queue is currently empty is insufficient, since the populating thread may have more work to produce, or other threads may later choose to donate work.

Work donation is handled by points 3 and 6. We note that once we have decided to donate something, we also donate every non-eliminated node to the right at the current level. For the condition at 6, we should decide to start donating work only if the populating thread is done, and if the queue is currently empty.

The enqueue operation at 7 should be blocking and bounded for the populating thread, but non-blocking and unbounded for work donation.

We may perform the same modifications we did to create **tmcsa** from **mcsa** to obtain a threaded version of **bmcsa**. We shall call this algorithm **tbmcsa**.

3.3 Potential Speedup

Intuitively, we may expect that doubling the number of cores available could at best halve the runtime of the algorithm—that is, the speedup we obtain could at best be linear. However for branch and bound algorithms, and backtracking search in general, this is not the case, and super-linear speedup is possible—we may be able to obtain a speedup of greater than n using n processors [LS84, MG85, LW86, Spe89, HM90, CHH91, BKT95, Sut08]. In particular, Amdahl's law [Amd67] is not a limiting factor, since it operates under the assumption that the amount of work to be performed is fixed.

We may analyse the potential for speedup in more detail. Again, we view our search as operating over a tree, where each node is a call to **expand**. We say a node is *eliminable* if it can be eliminated by a bound, if a sufficiently large incumbent has been found, and *ineliminable* if it cannot be eliminated by the bound regardless of the incumbent. Assuming neither the bound function nor the ordering is changed, *proving optimality* of a solution requires exploring at least all ineliminable nodes. Thus, once a maximum clique has been found, the remaining amount of work is fixed, and the best speedup we can hope for is linear, minus overheads.

Before a maximum has been found, matters are more complicated. If additional workers are exploring portions of the search tree, they may be wasting their time visiting parts of the tree that, in the sequential run, would be eliminated by virtue of a better incumbent having been found. Conversely, it is possible that an additional worker may find a good incumbent much more quickly than in the sequential run, eliminating large portions of the tree that would otherwise have to be explored. In this case the speedup gained may be super-linear; we illustrate this (and other possibilities) in Figure 3.2 on page 16.

Given certain reasonable assumptions, a bound on the best speedup we could hope to achieve for a given problem can be calculated. We assume that nodes and runtime are roughly interchangeable, and that there are no overheads and no sequential portion of the algorithm. Furthermore, we assume that the tree is "wide" rather than "deep", so we do not worry about the cost of getting to any particular node. We may decompose the time spent on a sequential run T_{seq} into the time spent visiting ineliminable nodes T_{inelim}, the time spent visiting nodes which could be eliminated T_{wasted}, and the shortest possible time it takes to find but not prove optimality of a maximum clique T_{oracle} (if the heuristic were an oracle, which always made the best possible choice). Thus

$$T_{seq} = T_{inelim} + T_{wasted} + T_{oracle}.$$

We assume that T_{oracle} is effectively zero—this is a reasonable assumption, since for non-trivial problems the cost of ω calls to $expand$ is dwarfed by T_{inelim}. Thus

$$T_{seq} = T_{inelim} + T_{wasted}. \tag{3.1}$$

In the best possible case, one of the workers in the parallel algorithm will make choices close to those of the oracle heuristic. This leads to effectively zero wasted effort. Thus, assuming we can gain a perfect linear speedup

Algorithm 4: tmcsa, a threaded variation of **mcsa**.

```
1  tmcsa :: (Graph g) → Vertices
2  begin
3      b ← ∅                                                          ❶
4      launch the populating thread do
5          expand(g, ∅, V(g) in non-increasing degree order, b)
6      launch multiple worker threads do                              ❷
7          while there is work left do
8              (c, o) ← dequeue
9              expand(g, c, o, b)
10     join all threads
11     return b

                                  c: Candidate clique
                                  o: Vertices adjacent to everything in c, in order
                                  b: Incumbent (shared between threads)

12 expand :: (Graph g, Vertices c, Vertices o, Vertices b)
13 begin
14     enqueueing ← false
15     if we are the populating thread, and |c| equals the splitting distance then   ❸
16         enqueueing ← true
17     (ω, p) ← colourise(g, o)
18     for i ← |p| downto 1 do
19         if |c| + ω_i > |b| then                                    ❹
20             v ← p_i
21             c ← c ∪ {v}
22             o' ← o ∩ N(g, v)
23             if o' = ∅ then
24                 if |c| > |b| then b ← c                            ❺
25             else
26                 if we should start to donate then                  ❻
27                     enqueueing ← true
28                 if enqueueing then enqueue(c, o')                  ❼
29                 else expand(g, c, o', b)
30             c ← c \ {v}
31             p ← p \ {v}
32             o ← o \ {v}
33 colourise :: (Graph g, Vertices o) → (Array of Integer, Vertices)
```

as per Algorithm 2 on page 7

from n cores, the best possible parallel runtime T_{par} is

$$T_{par} = \frac{T_{inelim}}{n}$$

and so a limit to achievable speedup S is given by

$$\begin{aligned} S &= \frac{T_{seq}}{T_{par}} \\ &= \frac{T_{inelim} + T_{wasted}}{\left(\frac{T_{inelim}}{n}\right)} \\ &= n\left(1 + \frac{T_{wasted}}{T_{inelim}}\right). \end{aligned} \tag{3.2}$$

In particular, if the sequential run wastes no time ($T_{wasted} = 0$), we can at best gain a linear speedup ($S = n$), but if the sequential run is wasteful then the speedup may be super-linear ($S > n$).

We may reformulate S in terms of easily measured quantities. We rearrange (3.1) to be in terms of T_{wasted},

$$T_{wasted} = T_{seq} - T_{inelim},$$

and observe that we may measure both terms on the right. T_{seq} is simply a sequential run of the algorithm; to obtain T_{inelim}, we rerun the sequential algorithm, but with a preset incumbent size equal to what we know the maximum to be. Substituting into (3.2),

$$\begin{aligned} S &= n\left(1 + \frac{T_{seq} - T_{inelim}}{T_{inelim}}\right) \\ &= n\left(\frac{T_{seq}}{T_{inelim}}\right). \end{aligned} \tag{3.3}$$

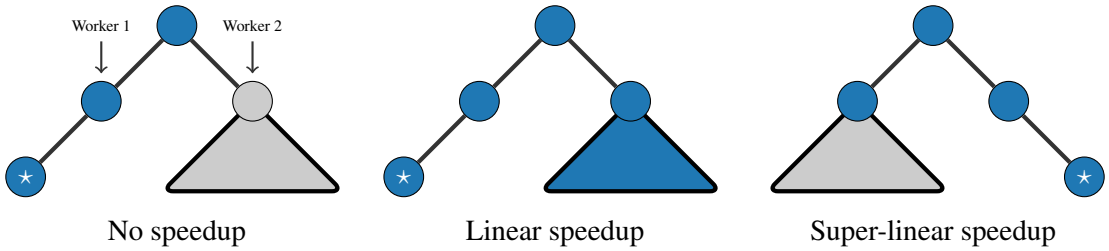
Again, we see that a linear speedup being the best possible corresponds to the sequential run taking the least possible time (that is, if there is no wasted effort).

Finally, we preempt the possible criticism that a super-linear speedup is only possible when the sequential implementation is defective [FLJ86]. Whilst it is true that a super-linear speedup corresponds to poor ordering choices being made by the sequential algorithm, if it were possible to always make good choices in polynomial time, we would be able to solve the decision problem in polynomial time too, and would thus have $\mathsf{P} = \mathsf{NP}$. We therefore cannot expect that the sequential algorithm will always perform the minimum amount of work required to obtain a solution—that is, we should not expect T_{wasted} to be small. In fact, even an *average* super-linear speedup is not an indication of a defect [Spe89].

3.3.1 Avoiding a Slowdown

Although in the best case speedups are possible, we must also be careful not to introduce the possibility of a slowdown (where $T_{seq} < T_{par}$). Ignoring overheads, this may be done by ensuring that one worker follows "the same path" (or a subset thereof) as the sequential version of the algorithm [BKT95] (but see Section 3.3.2 for complications). We may do this by ensuring that items placed into our work queue are evaluated in the order in which they appear in the sequential call tree; indeed, using a queue rather than a pool provides this guarantee (work donation may interfere, but this can be avoided by only donating when another worker is idle). Since

Figure 3.2: Possibilities for speedup in parallel branch and bound algorithms. Assume the algorithm traverses from left to right. For parallel execution consider two workers, initially splitting at distance 1 from the root and with work donation enabled. The triangles denote a large search space. We have a unique solution, marked $\star$, that is able to eliminate parts of the tree shown in light grey. On the left, we expect effectively no speedup: all the work done by the second worker is wasted. In the middle, we expect a linear speedup: the solution does not allow us to eliminate any nodes, so we are simply dividing a fixed amount of work up. On the right, super-linear speedup: if the second worker finds the solution quickly, the first worker may eliminate a large amount of the search space that would be explored in the sequential case.



the only information learned during search is the size of the incumbent (which we are assuming can be shared "instantly" between workers), we do not have to worry about losing additional knowledge by eliminating parts of the search tree that cannot contain a maximum clique. With this in mind, we obtain a range of possibilities, which we illustrate in Figure 3.2:

- We get a slowdown. If this happens, it is due either to overheads or to matters discussed in the following section.
- We get no speedup. This happens if the sequential algorithm quickly finds the largest clique, and if all remaining nodes can be eliminated by this bound. In other words, all the work done by additional threads ends up being wasted effort.
- We approach a linear speedup. This happens if the sequential algorithm quickly finds the largest clique, but proving optimality requires more work, which can be distributed evenly between threads.
- We get a super-linear speedup. This happens if the sequential algorithm takes a long time to find the largest clique, but one of the additional threads "gets lucky" and finds it quickly, allowing the bound to eliminate large portions of the tree that would have otherwise been explored.

We will return to these possibilities, and see that "gets lucky" actually happens in practice, in Section 5.6.

3.3.2 Complications from Hyper-Threading

Hyper-threading "makes a single physical processor appear as two logical processors; the physical execution resources are shared and the architecture state is duplicated for the two logical processors" [MBH+02]. The system we will be using in Chapter 5 is hyper-threaded; this causes complications.

Firstly, it means that when going from using one thread per physical processor to one thread per logical processor, we should not expect to be able to double our speedup. The cited Intel literature suggests a performance increase of "up to 30%"—this figure is derived from benchmarks (which show performance increases of "21%"

and "16 to 28%"), not theory. Taken at face value, this means that a speedup of around 15.6 on the 12 core, hyper-threaded system we describe later should be considered "linear".

Secondly, and more problematically, this means that if two (software) threads are running on the same physical processing core, each will run more slowly than if it had the core to itself [BP04]. Because we are not executing a fixed amount of work on each thread, this can lead to a slowdown anomaly—this is a variation of what Bruin et. al. describe as the "[danger of increasing] the processing power of a system by adding a less powerful processing element" [BKT95]. We will assume that so long as the number of threads we use is no greater than the number of physical processing cores, the operating system scheduler will be smart enough to allow us to ignore this issue. For larger numbers of threads, we proceed with the understanding that this could possibly make matters worse, not better[1].

3.4 Options for Splitting Distance

With our analysis of the potential for speedup, we may consider what depth we should use for splitting the search tree. If the splitting distance d is large (i.e. we split a long way from the root), we closely follow the steps of the sequential algorithm. On the other hand, if d is 1, the path taken by our additional workers diverges heavily.

If we expect the sequential algorithm to find a maximum clique almost immediately, we will be spending most of our time proving optimality. In this case, choice of splitting distance should make little difference with a sufficiently efficient implementation. If we are more pessimistic, we might expect the sequential algorithm to start off well but spend a long time deep down in the tree before it finds a maximum. A large splitting distance will allow our additional threads to speed up this process.

But in general a heuristic is weakest at the top of search—this is the basis of Harvey and Ginsberg's limited discrepancy search [HG95]. Splitting at distance 1 could be seen as an alternative way of hedging our bets against a bad initial heuristic choice. If this intuition is correct (and Prosser and Unsworth provide evidence in its favour, at least for hard problems [PU11]), this approach will maximise our opportunities for super-linear speedups: we would expect that often the sequential algorithm would waste considerable time exploring parts of the tree that could have been eliminated, if we had made better decisions early on.

We consider *sometimes* producing a super-linear speedup to be more desirable than *consistently* producing a near-linear speedup: following Gustafson's Law [Gus88], we wish to open up the possibility of tackling larger problems, rather than just tacking the same problems faster. Thus, noting that our potential speedup is highest when T_{wasted} is large, we conjecture that a splitting distance of 1 is the best choice. We measure the effects of using different splitting distances to verify this in Sections 5.3.2 and 5.6.

[1] The same problem arises if we use more worker threads than can be executed simultaneously. This, however, is directly under our control.

Chapter 4

Implementation

In this chapter we discuss an implementation of the algorithms developed in Chapters 2 and 3.

4.1 Choice of Environment

We chose to program our implementation in C++. Our justification is threefold.

Firstly, a claimed speedup is more convincing if our sequential runtimes are comparable to state of the art implementations. The implementation of Carmo and Züge [CZ12], which uses Python, comes with the note that "implementations geared toward maximum efficiency are reported to run the same algorithms discussed here in times orders of magnitude smaller"; we wish to avoid the necessity of such a disclaimer. The two sequential implementations [TSH+10, SMRLH11] upon which our algorithm is based use C and C++ respectively.

Secondly, the new C++11 standard [ISO12] provides extensive language level support for threading [Wil12], including support for atomics which would not be available via C with POSIX threads [IEE95]. (Atomics are supported in C11 [ISO11]; however, compiler support was not available at the start of this project. Further, we find C++11 threads to be considerably less verbose and simpler to work with.)

In particular, we will *not* be using MPI [MPI94]. Although MPI has commonly been used for parallel branch and bound, we are targeting shared memory systems, and so message passing is not our only option. We will also not be using OpenMP [Ope11]: using C++11 threads directly presents us with considerably more flexibility when it comes to sharing of data, and we are willing to deal with a possible increase in programming difficulty. We note the comparison of MPI and OpenMP by Barreto and Bauer [BB10], and aim for more favourable results than either method.

Finally, interesting phenomena tend to emerge as we consider larger problems. An efficient implementation allows us to consider a wider range of graphs, giving us a more powerful research environment [Mor02].

Due to hardware availability, we operate exclusively in Linux. There is nothing inherently unportable in our code; however, we do not deliberately avoid C++11 features that are not implemented in compilers other than GCC 4.7.

4.2 Graph Data Structures

Let $G = (V, E)$ be a graph with $|G| = n$. We may assume without loss of generality that V is of the form $\{0, 1, \ldots, n-1\}$. Thus we need only store n to determine the vertex set. For the edge set E, we have a choice of two simple representations. The first is to use an $n \times n$ adjacency matrix with boolean entries representing adjacency. This requires $O(n^2)$ memory and allows adjacency tests to be performed in constant time. Alternatively, for each vertex we may store a list of adjacent vertices. This requires less memory for sparse graphs, but random adjacency tests are no longer constant time. (Use of a more complex data structure in place of a list may circumvent this issue.) In both cases, we have the option of exploiting symmetry to halve the number of entries that must be stored, at the expense of a (constant) slower lookup time.

We are not restricting ourselves to sparse graphs, so we use an adjacency matrix. For **bmcsa**, to allow the use of bit operations, each row *must* be laid out as a sequence of adjacent bits in memory. For **mcsa** we have slightly more freedom, and may use either representation of E.

4.3 Data Sharing

Since we are using shared memory parallelism, and since the graph data structure is not modified during execution, we may share the graph between threads to save memory (and improve cache performance). C++ provides us with strong enough guarantees to do this without locking [Wil12].

Sharing the incumbent requires more attention. Our performance guarantees require that updates to the incumbent are made available "immediately", and McCreesh and Prosser [MP12] provide examples where not doing so leads to considerable performance problems. To simplify the problem, we note that we only need to share the size of the incumbent, not what its members are. Thus we need only share a single integer, and can retrieve the actual maximal clique when joining threads.

The standard approach is to use a mutex to guarantee exclusive access to the integer. However, this is a stronger form of exclusivity than is required: we expect that most of the time, we will be reading the integer, but not updating it. There is no facility for dealing with this directly in C++11, but the Boost Thread libraries [WE12] provide a shared (read-write) mutex. We could alternatively use an atomic to avoid the need for locking altogether. We cannot be sure upfront which solution is best, so we implement all three and compare the performance in Section 5.3.1.

4.4 Number of Threads

The number of threads to use is left as a configuration parameter—this allows experiments to be performed evaluating scalability. For a sensible default, C++11 provides a library function[1] which gives us a hint as to how many threads to use. The standard does not define how this value is calculated, but in practice, on hyper-threaded platforms, hardware threads are counted, not processor cores. As discussed in Section 3.3.2, this could potentially cause a slowdown, and we must evaluate whether this is a problem. (Specifying more threads than can be executed simultaneously leads to the same issue.)

We do not count the initial populating thread towards the thread count—it is expected that the amount of work performed for population will be a very small part of the overall work. We also do not count the main program thread, which just spawns then joins child threads.

[1] **std::thread::hardware_concurrency**

Chapter 5

Experimental Evaluation

In this chapter we begin by showing that our sequential implementation performs competitively with published results. We then present experimental results comparing various possible implementation choices for our threaded algorithms, and justifying our choice of splitting distance. With these choices made, we run our implementation on a variety of standard and random benchmarks and show that we can consistently obtain close-to-linear speedups on non-trivial problems, with super-linear speedups being common.

5.1 Experimental Data and Methodology

We work with three sets of experimental data. The first set, from the DIMACS Implementation Challenges [DIM], contains a smörgåsbord of random and real-world problems of varying difficulty—some can be solved in a few milliseconds, whilst others have not been solved at all. We report results for all of these graphs.

The second set consists of "Benchmarks with Hidden Optimum Solutions for Graph Problems" [BHO]. Each of these contains a maximum clique of known size that has been hidden in a way intended to make it computationally very hard to find. We present results only for the smaller problems.

Finally, we work with Erdős-Réyni random graphs, which have a chosen number of vertices, and an edge between each (unordered) pair of vertices with a given independent probability. We denote by $G(n, p)$ such a graph with n vertices and edge probability p.

For timing results, following standard practice we exclude the time taken to read in the graph from the file. We include the time to do preprocessing on the graph (this is not entirely standard, but we consider it the more realistic approach). We measure the wallclock time until completion of the algorithm. In the case of threaded algorithms, we include the time taken to launch and join threads and to accumulate results as part of the runtime. When giving speedups, we compare threaded runtimes against the sequential algorithm, not against the threaded algorithm running with a single thread, and all experiments have actually been performed on real hardware and are not simulations. We also spend the following section verifying that our implementation of the sequential algorithm is competitive with published results. In other words, our speedup figures measure what we can genuinely gain over a state of the art implementation [Bai09].

Except where otherwise noted, experiments are performed on a computer with two Intel Xeon E5645 ("Westmere-EP") processors running at 2.4GHz. Each of these processors has six cores, and hyper-threading is enabled, giving a total of twelve "real" cores, or twenty-four hardware threads. To get a better view of scalability we report results using four, eight, twelve and twenty-four worker threads. As discussed in Section 3.3.2, we should *not*

expect speedup to double when going from twelve to twenty-four threads, and a lower speedup-per-thread here (or even a slowdown) is not a sign of scalability issues.

5.2 Comparison of Sequential Algorithm to Published Results

The Java implementation by Prosser [Pro12] allows for easy validation of the sequential implementation. By copying the node counting mechanism used, and ensuring that tie-breaking in sorting by degree was performed in the same way (via vertex number), we are able to ensure that our implementation carries out *exactly* the same steps on sample problems. In addition, running both implementations on the same problems on the same machine and comparing runtimes allows us to be confident that we have not introduced substantial slowdowns due to poor programming (for example, the wrong choice of representation for the candidate and adjacency sets can easily double runtimes).

We use the "brock400" instances from DIMACS and obtain the results in Table 5.1. It was verified that the node counts were identical in each case. We observe that our implementation runs significantly faster (but by a more or less constant factor for **mcsa**, and a different more or less constant factor for **bmcsa**). This is not unexpected: our implementation is written in C++ rather than Java, and we have coded with performance in mind rather than modularity.

Table 5.1: Comparison of sequential runtimes (in seconds) and nodes with the implementation of Prosser [Pro12]. Our **mcsa** is the same algorithm as Prosser's **MCSa1**, and **bmcsa** is the same as **BBMC1**.

Problem	**mcsa**		**MCSa1**		**bmcsa**		**BBMC1**	
brock400_1	1510s	198,359,829	2566s	198,359,829	275s	198,359,829	716s	198,359,829
brock400_2	1112s	145,597,994	1860s	145,597,994	201s	145,597,994	500s	145,597,994
brock400_3	864s	120,230,513	1440s	120,230,513	159s	120,230,513	396s	120,230,513
brock400_4	428s	54,440,888	721s	54,440,888	78s	54,440,888	196s	54,440,888

The source code for the implementation by San Segundo [SMRLH11] is not publicly available. However, we obtained access to a machine with the same CPU model as was used to produce the published results. Although our algorithm is not identical due to differences in initial vertex ordering, we see from Table 5.2 that runtimes are comparable. Results in the penultimate column were provided by Pablo San Segundo for a more optimised implementation on the same machine. Note that the computer used to produce this comparison is not the same as the one used for other experiments in this report.

Table 5.2: Comparison of runtimes (in seconds) for **bmcsa** with San Segundo's published and improved results for **BBMCI** [SMRLH11] (which differs slightly from our algorithm), and with runtimes using Prosser's **BBMC1** [Pro12] (which is the same as our algorithm). The system used to produce these results has the same model CPU as was used by San Segundo.

Problem	**bmcsa**	**BBMCI**	San Segundo	**BBMC1**
brock400_1	198s	341s	270s	507s
brock400_2	144s	144s	113s	371s
brock400_3	114s	229s	180s	294s
brock400_4	56s	133s	107s	146s

5.3 Analysis of Implementation Choices for the Threaded Algorithm

In Chapters 3 and 4 we left certain implementation choices open: we did not decide how the incumbent would be shared, and we did not commit to a particular splitting distance. Here we evaluate our options experimentally.

5.3.1 Locking Mechanism for Sharing the Incumbent

In Section 4.3 we discussed three possible ways of sharing the incumbent: by using a mutex, by using a shared mutex, or by using atomics. All three methods were implemented for **tmcsa**, and benchmarks were performed using a varying number of threads over 500 instances of $G(200, 0.9)$. The average runtimes and speedups (i.e. the sum of the sequential runtimes over the sum of the parallel runtimes [BW05]) are presented in Table 5.3. These figures use splitting at distance 1 from the root and no work donation.

Table 5.3: Average runtime (in seconds) and speedup over **mcsa** for **tmcsa** with 500 instances of $G(200, 0.9)$ using different locking mechanisms for sharing the incumbent, and 4 to 24 threads on a 12 core hyper-threaded system.

Method	**tmcsa**							
	4		8		12		24	
Mutex	66.6s	4.0	34.1s	7.8	25.5s	10.4	26.3s	10.1
Shared mutex	75.5s	3.5	48.1s	5.5	44.0s	6.1	45.6s	5.8
Atomics	65.7s	4.1	33.0s	8.1	23.6s	11.3	22.8s	11.7

We see that shared mutexes consistently give *worse* runtimes than simple, exclusive mutexes, and that the disadvantage increases as the number of threads increases. There is less of a difference between mutexes and atomics, but using atomics does provide a measurable advantage. It is also interesting to note that only with atomics does performance continue to improve when going from 12 to 24 threads. With this in mind, we opt to use atomics despite the increased programming difficulty.

5.3.2 Splitting Distance and Work Donation

In Section 3.4 we suggested that splitting closer to the root may increase the possibility for super-linear speedup. We put this to the test by benchmarking the effects of splitting at different distances from the root over 500 instances of $G(200, 0.9)$. We also measure the effect of work donation.

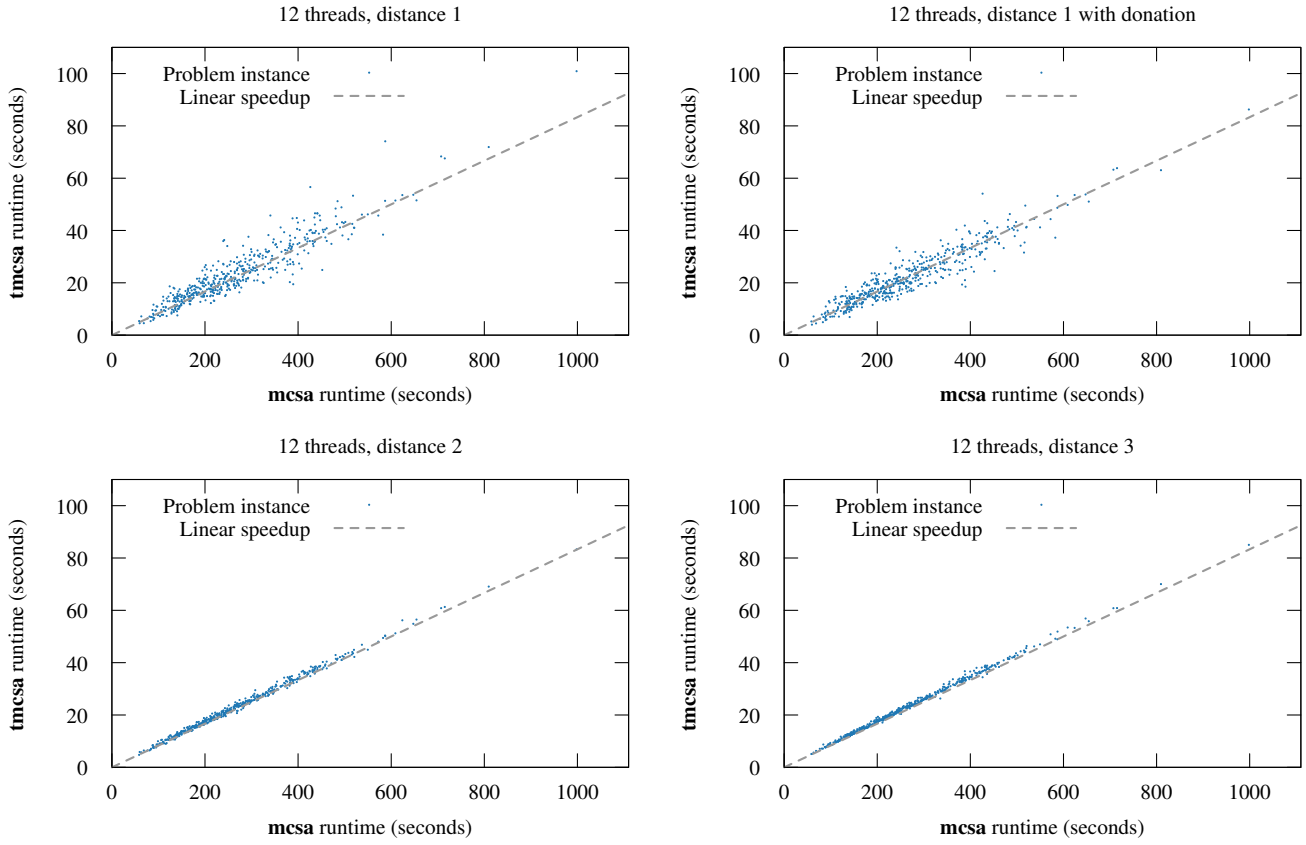
The results are presented in Table 5.4 on the next page. We see that *on average*, the splitting distance has little effect on runtimes. Work donation appears to provide a slight benefit in some cases, and a slight penalty in others. However, the table does not tell the full story: we wish to know whether splitting distance affects the possibility of super-linear speedups. For this we refer to Figure 5.1 on the following page, which shows threaded versus sequential runtimes for each instance. We see that for splitting at distance 1 with or without work donation, we get a variety of speedups, whereas for distances 2 and 3 the speedups in each case are always close to the average.

We argued in Section 3.4 that increasing the possibility of a super-linear speedup is more desirable than gaining a consistent speedup. With that in mind, we opt for splitting at distance 1, and using work donation: we will be hedging our bets against bad early heuristic choices, rather than assisting a nearly-sequential search. (We see in Section 5.6 that for some problems, the extent of the super-linear speedup goes considerably beyond what we have observed from $G(200, 0.9)$.)

Table 5.4: Average runtime (in seconds) and speedup over **mcsa** for **tmcsa** with 500 instances of $G(200, 0.9)$ using different splitting mechanisms, and 4 to 24 threads on a 12 core hyper-threaded system. A more detailed picture is presented in Figure 5.1.

Method	**tmcsa**							
	4		8		12		24	
Depth 1	65.7s	4.1	33.0s	8.1	23.6s	11.3	22.8s	11.7
Depth 1 with donation	65.9s	4.0	32.5s	8.2	22.3s	11.9	18.4s	14.5
Depth 2	67.3s	4.0	33.6s	7.9	23.0s	11.6	18.1s	14.7
Depth 2 with donation	67.4s	4.0	33.6s	7.9	23.2s	11.5	18.1s	14.7
Depth 3	67.5s	3.9	33.8s	7.9	23.3s	11.5	18.5s	14.4
Depth 3 with donation	67.6s	3.9	33.9s	7.9	23.5s	11.3	18.5s	14.4

Figure 5.1: Runtimes for **mcsa** vs **tmcsa**, 500 instances of $G(200, 0.9)$ with different splitting distances using 12 threads on a 12 core hyper-threaded system. Each point is a problem instance. Points below the line represent super-linear speedups. We see that splitting at distance 1, with or without donation, increases the possibility of super-linearity at the expense of consistency.



5.4 Threaded Experimental Results on Standard and Random Benchmarks

Experimental results on graphs from DIMACS and BHOSLIB are presented in Table 5.5 on the next page, and on Erdős-Réyni random graphs in Table 5.6 on page 26. The DIMACS benchmarks include a wide variety of problem sizes and difficulties. Problems which took under one second to solve with **mcsa** are shown with a grey background. We attempted every DIMACS instance at least with **tbmcsa** with 24 threads. Some problems took over a day to solve, and for these we indicate the largest clique we found in that time. Blank entries in the table indicate problems that were not attempted with other algorithms or numbers of threads due to resource limitations—as per Gustafson's law [Gus88], we are using parallelism in these cases to tackle larger problems rather than to produce a speedup.

When the sequential execution takes less than one quarter of a second, we sometimes get a speedup, and sometimes get a slowdown. Runtime measurements in this area are not consistently reproducible. This is not unexpected: we must still launch and join all our worker threads, and this overhead can only be ignored for non-trivial problems. There is also the sequential pre-processing of the graph to overcome. We do not attempt to address this issue: other sequential algorithms are already better for "easy" graphs.

For sequential runtimes above one quarter of a second, we always produce a speedup. On the DIMACS benchmarks, with 4, 8 and 12 threads this speedup is usually at least close to linear and super-linear speedups are common. Over the instances where the **mcsa** runtime is at least one second (those not shown in grey in Table 5.5), our worst speedups are 2.9 from 4 threads, 3.7 from 8 threads, and 3.6 from 12 threads with **tmcsa**, and 3.2 from 4 threads, 4.5 from 8 threads, and 4.2 from 12 threads with **tbmcsa**. These all correspond to problems where the threaded runtimes end up below one second. In the best case, we get speedups of 18.1 from 4 threads (and of 139.8 on one of the trivial instances), 126.3 from 8 threads and 109.1 from 12 threads using **tmcsa**, and from **tbmcsa**, 19.5 from 4 threads, 75.3 from 8 threads and 102.2 from 12 threads. We get super-linear speedups for in between one third and one half of these cases.

For the BHOSLIB benchmarks, which are designed to be hard to solve, our results are particularly consistent: our speedups are nearly always super-linear, with a speedup of between 11.7 and 15.0 from 12 threads.

On random graphs we still produce a speedup for every non-trivial data set, although the speedups are lower. This is not surprising: most of our random graphs are relatively large and sparse, and have an expensive initial sequential portion.

When using 24 threads we must worry about hyper-threading; nonetheless, our speedups typically (but not always) improve over those for 12 threads, and even when they do not, a speedup is still obtained.

We do not see any evidence of scalability problems when making use of all the cores available to us. A single shared queue does not appear to be a source of contention, and the use of atomics for sharing the incumbent directly is successful. We cannot say when this design will no longer be adequate, but it is at a point beyond the number of cores commonly available in current systems.

Table 5.5: Experimental results for DIMACS and BHOSLIB instances. Shown first are runtimes for **mcsa**, and runtimes with speedups over **mcsa** for **tmcsa** using 4 to 24 threads on a 12 core hyper-threaded system. Next are runtimes for **bmcsa**, and runtimes with speedups over **bmcsa** for **tbmcsa**. Super-linear speedups are shown in bold; problems where **mcsa** takes under one second have a grey background, and blanks indicate unattempted problems.

Problem	ω	**mcsa**	**tmcsa**								**bmcsa**	**tbmcsa**							
			4		8		12		24			4		8		12		24	
brock200_1	21	2.5s	631ms	3.9	293ms	**8.5**	255ms	9.7	164ms	15.2	386ms	98ms	3.9	57ms	6.8	37ms	10.4	34ms	11.4
brock200_2	12	16ms	7ms	2.3	16ms	1.0	18ms	0.9	30ms	0.5	2ms	3ms	0.7	6ms	0.3	6ms	0.3	7ms	0.3
brock200_3	15	71ms	22ms	3.2	19ms	3.7	24ms	3.0	26ms	2.7	10ms	6ms	1.7	8ms	1.2	8ms	1.2	9ms	1.1
brock200_4	17	261ms	72ms	3.6	49ms	5.3	49ms	5.3	41ms	6.4	40ms	12ms	3.3	12ms	3.3	9ms	4.4	10ms	4.0
brock400_1	27	1,510.3s	386.6s	3.9	198.6s	7.6	137.7s	11.0	69.9s	21.6	274.9s	69.3s	**4.0**	36.0s	7.6	25.3s	10.9	11.7s	23.4
brock400_2	29	1,112.2s	285.0s	3.9	126.2s	**8.8**	94.4s	11.8	50.0s	22.2	200.8s	50.7s	**4.0**	22.2s	**9.0**	16.6s	**12.1**	8.9s	22.5
brock400_3	31	864.1s	207.7s	**4.2**	95.5s	**9.0**	59.0s	**14.7**	31.6s	**27.4**	159.4s	38.6s	**4.1**	17.4s	**9.1**	10.6s	**15.1**	5.7s	**28.0**
brock400_4	33	428.4s	101.0s	**4.2**	47.9s	**8.9**	12.7s	**33.7**	10.1s	**42.6**	77.5s	17.9s	**4.3**	8.5s	**9.1**	1.9s	**40.4**	1.7s	**45.5**
brock800_1	23	20,781.5s	5,170.0s	**4.0**	2,511.8s	**8.3**	1,714.2s	**12.1**	1,122.8s	18.5	4,969.8s	1,216.5s	**4.1**	587.1s	**8.5**	405.6s	**12.3**	269.9s	18.4
brock800_2	24	20,714.6s	5,211.3s	**4.0**	2,536.1s	**8.2**	1,629.9s	**12.7**	1,109.3s	18.7	4,958.2s	1,237.8s	**4.0**	584.8s	**8.5**	386.0s	**12.8**	266.7s	18.6
brock800_3	25	19,036.9s	4,753.8s	**4.0**	2,241.6s	**8.5**	1,457.5s	**13.1**	964.6s	19.7	4,590.7s	1,125.2s	**4.1**	533.2s	**8.6**	347.8s	**13.2**	222.2s	20.7
brock800_4	26	7,559.4s	1,802.3s	**4.2**	965.7s	7.8	656.0s	11.5	548.4s	13.8	1,733.0s	408.7s	**4.2**	220.3s	7.9	152.3s	11.4	131.5s	13.2
c-fat200-1	12	0ms	7ms	0.0	10ms	0.0	15ms	0.0	24ms	0.0	0ms	2ms	0.0	6ms	0.0	6ms	0.0	4ms	0.0
c-fat200-2	22	0ms	3ms	0.0	9ms	0.0	11ms	0.0	18ms	0.0	0ms	1ms	0.0	5ms	0.0	4ms	0.0	8ms	0.0
c-fat200-5	58	1ms	7ms	0.1	26ms	0.0	13ms	0.1	24ms	0.0	0ms	2ms	0.0	6ms	0.0	6ms	0.0	8ms	0.0
c-fat500-1	14	1ms	18ms	0.1	151ms	0.0	75ms	0.0	110ms	0.0	1ms	4ms	0.2	7ms	0.1	8ms	0.1	12ms	0.1
c-fat500-10	124	3ms	21ms	0.1	62ms	0.0	85ms	0.0	133ms	0.0	2ms	4ms	0.5	8ms	0.2	12ms	0.2	13ms	0.2
c-fat500-2	26	1ms	21ms	0.0	51ms	0.0	77ms	0.0	206ms	0.0	1ms	3ms	0.3	8ms	0.1	10ms	0.1	15ms	0.1
c-fat500-5	63	1ms	23ms	0.0	53ms	0.0	88ms	0.0	143ms	0.0	2ms	3ms	0.7	9ms	0.2	10ms	0.2	13ms	0.2
C125.9	34	318ms	102ms	3.1	45ms	7.1	47ms	6.8	41ms	7.8	43ms	13ms	3.3	11ms	3.9	15ms	2.9	21ms	2.0
C250.9	44	11,311.4s	2,902.6s	3.9	1,588.8s	7.1	974.3s	11.6	844.2s	13.4	1,606.8s	411.2s	3.9	228.1s	7.0	147.8s	10.9	149.0s	10.8
C500.9	$\geq$54																	>1 day	
C1000.9	$\geq$58																	>1 day	
C2000.5	16																	4,347.9s	
C2000.9	$\geq$65																	>1 day	
C4000.5	$\geq$18																	>1 day	
DSJC500_5	13	6.0s	1.8s	3.3	883ms	6.8	727ms	8.2	535ms	11.2	1.0s	266ms	3.9	152ms	6.7	130ms	7.9	89ms	11.5
DSJC1000_5	15	611.4s	155.9s	3.9	78.8s	7.8	53.3s	11.5	44.7s	13.7	135.7s	34.7s	3.9	17.4s	7.8	11.7s	11.6	9.1s	15.0
gen200_p0.9_44	44	16.7s	4.5s	3.7	903ms	**18.4**	792ms	**21.0**	329ms	**50.6**	2.5s	654ms	3.9	109ms	**23.2**	100ms	**25.3**	95ms	**26.6**
gen200_p0.9_55	55	1.4s	233ms	**6.0**	18ms	**77.1**	31ms	**44.8**	43ms	**32.3**	212ms	39ms	**5.4**	8ms	**26.5**	11ms	**19.3**	17ms	12.5
gen400_p0.9_55	$\geq$55																	>1 day	
gen400_p0.9_65	65																	17,773.9s	
gen400_p0.9_75	75																	3,799.6s	
hamming6-2	32	0ms	2ms	0.0	5ms	0.0	7ms	0.0	10ms	0.0	0ms	1ms	0.0	5ms	0.0	5ms	0.0	8ms	0.0
hamming6-4	4	0ms	2ms	0.0	5ms	0.0	5ms	0.0	8ms	0.0	0ms	2ms	0.0	4ms	0.0	2ms	0.0	9ms	0.0
hamming8-2	126	2ms	12ms	0.2	23ms	0.1	31ms	0.1	45ms	0.0	1ms	3ms	0.3	7ms	0.1	10ms	0.1	10ms	0.1
hamming8-4	16	272ms	77ms	3.5	56ms	4.9	47ms	5.8	51ms	5.3	43ms	13ms	3.3	13ms	3.3	12ms	3.6	10ms	4.3
hamming10-2	511	133ms	218ms	0.6	398ms	0.3	511ms	0.3	1.3s	0.1	35ms	32ms	1.1	38ms	0.9	44ms	0.8	44ms	0.8
hamming10-4	$\geq$40																	>1 day	
johnson8-2-4	4	0ms	2ms	0.0	4ms	0.0	3ms	0.0	5ms	0.0	0ms	1ms	0.0	2ms	0.0	5ms	0.0	5ms	0.0
johnson8-4-4	14	0ms	2ms	0.0	4ms	0.0	4ms	0.0	8ms	0.0	0ms	1ms	0.0	2ms	0.0	3ms	0.0	7ms	0.0
johnson16-2-4	8	320ms	88ms	3.6	59ms	5.4	39ms	8.2	60ms	5.3	50ms	15ms	3.3	12ms	4.2	11ms	4.5	24ms	2.1
johnson32-2-4	$\geq$16																	>1 day	
keller4	11	52ms	15ms	3.5	14ms	3.7	12ms	4.3	16ms	3.2	8ms	5ms	1.6	6ms	1.3	7ms	1.1	9ms	0.9
keller5	27																	10,241.3s	
keller6	$\geq$55																	>1 day	
MANN_a9	16	0ms	1ms	0.0	5ms	0.0	3ms	0.0	6ms	0.0	0ms	1ms	0.0	3ms	0.0	5ms	0.0	5ms	0.0
MANN_a27	126	2.9s	787ms	3.7	392ms	7.5	310ms	9.5	296ms	9.9	262ms	71ms	3.7	62ms	4.2	63ms	4.2	116ms	2.3
MANN_a45	345	1,726.9s	432.4s	**4.0**	211.6s	**8.2**	127.6s	**13.5**	107.1s	16.1	224.8s	56.3s	**4.0**	27.1s	**8.3**	18.2s	**12.3**	12.5s	17.9
MANN_a81	$\geq$1100																	>1 day	
p_hat300-1	8	5ms	10ms	0.5	28ms	0.2	37ms	0.1	58ms	0.1	1ms	4ms	0.2	9ms	0.1	9ms	0.1	10ms	0.1
p_hat300-2	25	35ms	18ms	1.9	26ms	1.3	40ms	0.9	53ms	0.7	7ms	5ms	1.4	6ms	1.2	9ms	0.8	12ms	0.6
p_hat300-3	36	6.0s	1.6s	3.7	918ms	6.6	668ms	9.0	531ms	11.4	1.1s	291ms	3.7	156ms	7.0	129ms	8.4	103ms	10.5
p_hat500-1	9	49ms	34ms	1.4	74ms	0.7	76ms	0.6	151ms	0.3	10ms	9ms	1.1	13ms	0.8	13ms	0.8	11ms	0.9
p_hat500-2	36	1.5s	411ms	3.6	290ms	5.1	247ms	6.0	248ms	6.0	251ms	69ms	3.6	44ms	5.7	42ms	6.0	40ms	6.3
p_hat500-3	50	646.3s	171.4s	3.8	89.4s	7.2	62.8s	10.3	51.1s	12.6	108.7s	29.5s	3.7	15.1s	7.2	10.8s	10.1	8.1s	13.4
p_hat700-1	11	170ms	94ms	1.8	149ms	1.1	146ms	1.2	251ms	0.7	60ms	19ms	3.2	19ms	3.2	22ms	2.7	20ms	3.0
p_hat700-2	44	13.5s	3.5s	3.9	1.8s	7.4	1.8s	7.6	1.4s	10.0	3.1s	946ms	3.3	402ms	7.7	403ms	7.7	270ms	11.5
p_hat700-3	62	7,446.4s	1,886.9s	3.9	1,009.3s	7.4	700.8s	10.6	603.3s	12.3	1,627.6s	419.9s	3.9	223.9s	7.3	156.8s	10.4	120.4s	13.5
p_hat1000-1	10	1.4s	365ms	**4.0**	395ms	3.7	398ms	3.6	563ms	2.6	232ms	72ms	3.2	51ms	4.5	44ms	5.3	48ms	4.8

Continued on next page

Table 5.5 – continued from previous page

Problem	ω	**mcsa**	**tmcsa**								**bmcsa**	**tbmcsa**							
			4		8		12		24			4		8		12		24	
p_hat1000-2	46	699.8s	179.3s	3.9	91.6s	7.6	62.3s	11.2	52.0s	13.5	159.2s	40.5s	3.9	20.4s	7.8	14.3s	11.1	11.7s	13.7
p_hat1000-3	68																	53,424.6s	
p_hat1500-1	12	12.1s	3.0s	**4.1**	1.7s	7.1	1.5s	8.1	1.5s	8.1	3.2s	821ms	3.9	433ms	7.4	341ms	9.4	259ms	12.4
p_hat1500-2	65	72,775.5s	18,389.1s	**4.0**	9,261.7s	7.9	6,327.2s	11.5	5,682.0s	12.8	24,338.5s	6,117.3s	**4.0**	3,089.0s	7.9	2,094.6s	11.6	1,789.1s	13.6
p_hat1500-3	$\geq$81																	$>$1 day	
san200_0.7_1	30	74ms	6ms	**12.3**	13ms	5.7	18ms	4.1	22ms	3.4	15ms	3ms	**5.0**	7ms	2.1	9ms	1.7	12ms	1.2
san200_0.7_2	18	2ms	5ms	0.4	10ms	0.2	11ms	0.2	23ms	0.1	1ms	2ms	0.5	6ms	0.2	7ms	0.1	9ms	0.1
san200_0.9_1	70	559ms	4ms	**139.8**	12ms	**46.6**	16ms	**34.9**	20ms	**28.0**	92ms	2ms	**46.0**	6ms	**15.3**	11ms	8.4	9ms	10.2
san200_0.9_2	60	2.4s	318ms	**7.5**	19ms	**126.3**	22ms	**109.1**	47ms	**51.1**	348ms	48ms	**7.2**	8ms	**43.5**	8ms	**43.5**	11ms	**31.6**
san200_0.9_3	44	54.7s	3.0s	**18.1**	2.1s	**25.7**	956ms	**57.3**	993ms	**55.1**	8.5s	439ms	**19.5**	319ms	**26.8**	177ms	**48.3**	271ms	**31.5**
san400_0.5_1	13	28ms	23ms	1.2	103ms	0.3	53ms	0.5	93ms	0.3	8ms	5ms	1.6	8ms	1.0	12ms	0.7	11ms	0.7
san400_0.7_1	40	1.1s	381ms	2.9	113ms	**9.8**	111ms	10.0	145ms	7.6	224ms	68ms	3.3	24ms	**9.3**	21ms	10.7	19ms	11.8
san400_0.7_2	30	9.7s	2.3s	**4.1**	1.7s	5.7	909ms	10.7	394ms	**24.6**	2.0s	590ms	3.4	298ms	6.7	176ms	11.4	76ms	**26.3**
san400_0.7_3	22	5.9s	441ms	**13.5**	311ms	**19.1**	295ms	**20.1**	281ms	21.1	1.3s	84ms	**15.0**	62ms	**20.3**	54ms	**23.4**	58ms	21.7
san400_0.9_1	100	201.9s	37.2s	**5.4**	2.6s	**77.9**	1.9s	**108.7**	1.5s	**136.6**	23.5s	5.3s	**4.4**	312ms	**75.3**	230ms	**102.2**	191ms	**123.0**
san1000	15	5.6s	1.5s	3.8	985ms	5.7	791ms	7.1	879ms	6.4	1.9s	488ms	3.9	281ms	6.8	173ms	11.1	108ms	17.7
sanr200_0.7	18	695ms	184ms	3.8	94ms	7.4	79ms	8.8	80ms	8.7	106ms	35ms	3.0	19ms	5.6	16ms	6.6	14ms	7.6
sanr200_0.9	42	138.3s	37.7s	3.7	19.5s	7.1	14.3s	9.7	14.4s	9.6	19.4s	5.3s	3.7	2.8s	6.8	2.2s	9.0	3.0s	6.4
sanr400_0.5	13	1.5s	389ms	3.7	240ms	6.1	194ms	7.5	186ms	7.8	253ms	80ms	3.2	44ms	5.8	36ms	7.0	27ms	9.4
sanr400_0.7	21	393.9s	100.9s	3.9	50.6s	7.8	33.9s	11.6	25.3s	15.6	72.1s	18.1s	**4.0**	9.1s	7.9	6.2s	11.7	4.6s	15.7
frb30-15-1	30	3,421.7s	851.1s	**4.0**	408.1s	**8.4**	234.7s	**14.6**	181.1s	18.9	657.1s	160.2s	**4.1**	76.6s	**8.6**	43.9s	**15.0**	35.5s	18.5
frb30-15-2	30	6,058.3s	1,509.6s	**4.0**	742.2s	**8.2**	481.5s	**12.6**	310.2s	19.5	1,183.1s	287.7s	**4.1**	141.7s	**8.3**	93.6s	**12.6**	65.8s	18.0
frb30-15-3	30	1,879.0s	431.8s	**4.4**	207.1s	**9.1**	133.1s	**14.1**	96.0s	19.6	356.7s	80.8s	**4.4**	38.8s	**9.2**	25.3s	**14.1**	19.5s	18.3
frb30-15-4	30	10,173.2s	2,612.6s	3.9	1,293.7s	7.9	870.8s	11.7	662.5s	15.4	1,963.2s	496.0s	**4.0**	246.1s	**8.0**	166.0s	11.8	124.4s	15.8
frb30-15-5	30	3,118.3s	705.2s	**4.4**	370.4s	**8.4**	229.6s	**13.6**	203.9s	15.3	577.1s	129.2s	**4.5**	68.4s	**8.4**	44.4s	**13.0**	42.1s	13.7
frb35-17-1	35	198,847.8s	46,934.9s	**4.2**	23,018.9s	**8.6**	14,760.7s	**13.5**	8,385.8s	23.7	51,481.7s	12,072.8s	**4.3**	5,949.7s	**8.7**	3,800.8s	**13.5**	2,532.0s	20.3
frb35-17-2	35	351,034.7s	85,171.0s	**4.1**	42,558.0s	**8.2**	27,787.8s	**12.6**	20,867.1s	16.8	91,275.0s	21,867.3s	**4.2**	10,959.2s	**8.3**	7,175.1s	**12.7**	5,677.3s	16.1
frb35-17-3	35	126,010.7s	31,243.8s	**4.0**	15,230.6s	**8.3**	10,578.8s	11.9	8,544.3s	14.7	33,852.1s	8,278.8s	**4.1**	4,063.2s	**8.3**	2,813.6s	**12.0**	2,349.3s	14.4
frb35-17-4	35	141,670.0s	35,293.6s	**4.0**	17,226.5s	**8.2**	10,051.0s	**14.1**	8,090.6s	17.5	37,629.2s	9,319.5s	**4.0**	4,522.7s	**8.3**	2,638.6s	**14.3**	2,196.1s	17.1
frb35-17-5	35	776,775.2s	191,165.9s	**4.1**	95,960.1s	**8.1**	62,805.0s	**12.4**	37,306.0s	20.8	205,356.0s	49,901.9s	**4.1**	25,130.3s	**8.2**	16,365.4s	**12.5**	10,363.4s	19.8

Table 5.6: Experimental results for random graph instances. Shown first are runtimes for **mcsa**, and runtimes with speedups over **mcsa** for **tmcsa** using 4 to 24 threads on a 12 core hyper-threaded system. Next are runtimes for **bmcsa**, and runtimes with speedups over **bmcsa** for **tbmcsa**. Super-linear speedups are shown in bold; problems where **mcsa** takes under one second have a grey background.

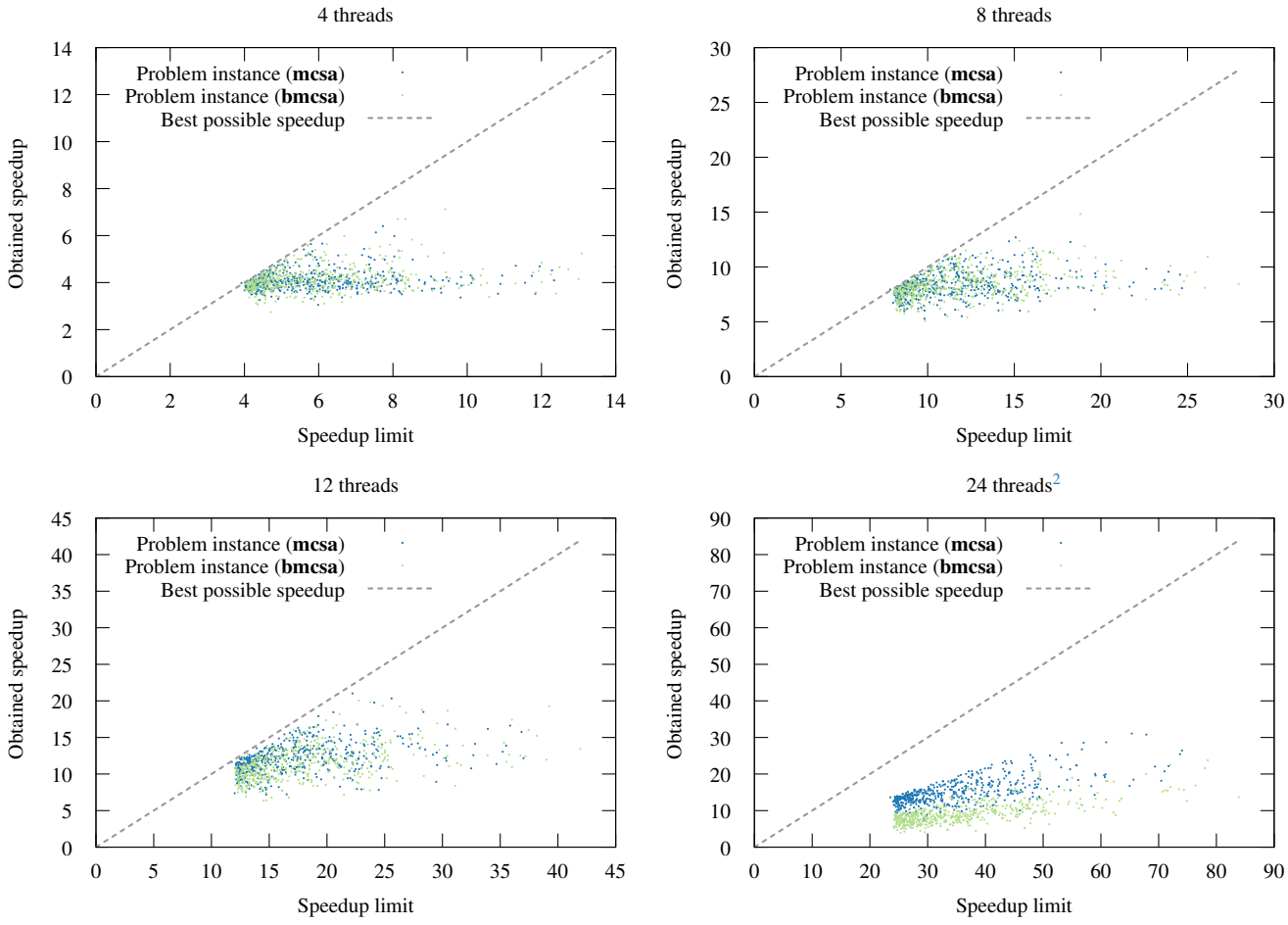
Problem	Sample size	**mcsa**	**tmcsa**								**bmcsa**	**tbmcsa**							
			4		8		12		24			4		8		12		24	
$G(200, 0.9)$	500	266.6s	65.9s	**4.0**	32.5s	**8.2**	22.3s	11.9	18.4s	14.5	39.2s	9.6s	**4.1**	4.8s	**8.2**	3.5s	11.2	4.6s	8.5
$G(1000, 0.1)$	10	31ms	95ms	0.3	211ms	0.1	328ms	0.1	575ms	0.1	18ms	18ms	1.0	23ms	0.8	26ms	0.7	33ms	0.5
$G(1000, 0.2)$	10	262ms	159ms	1.6	244ms	1.1	326ms	0.8	559ms	0.5	65ms	30ms	2.2	30ms	2.2	32ms	2.0	36ms	1.8
$G(1000, 0.3)$	10	2.5s	791ms	3.2	556ms	4.6	538ms	4.7	706ms	3.6	532ms	151ms	3.5	100ms	5.3	83ms	6.4	76ms	7.0
$G(1000, 0.4)$	10	28.5s	7.3s	3.9	3.9s	7.3	3.1s	9.3	2.4s	11.8	6.1s	1.6s	3.9	860ms	7.1	585ms	10.5	439ms	13.9
$G(1000, 0.5)$	10	629.6s	158.3s	**4.0**	80.2s	7.8	54.3s	11.6	42.0s	15.0	138.6s	34.9s	**4.0**	17.6s	7.9	12.2s	11.3	8.9s	15.6
$G(3000, 0.1)$	10	1.9s	769ms	2.5	877ms	2.2	1.1s	1.8	1.8s	1.1	640ms	220ms	2.9	171ms	3.7	156ms	4.1	168ms	3.8
$G(3000, 0.2)$	10	37.7s	9.8s	3.9	5.4s	6.9	4.4s	8.5	4.4s	8.6	11.9s	3.1s	3.9	1.7s	7.0	1.2s	10.1	900ms	13.3
$G(3000, 0.3)$	10	1,042.1s	274.1s	3.8	132.2s	7.9	89.4s	11.7	76.6s	13.6	358.5s	90.3s	**4.0**	45.6s	7.9	30.7s	11.7	23.2s	15.4
$G(10000, 0.1)$	10	214.5s	57.6s	3.7	36.3s	5.9	31.5s	6.8	50.9s	4.2	84.6s	21.8s	3.9	11.5s	7.3	8.5s	9.9	7.3s	11.6
$G(15000, 0.1)$	10	1,417.3s	375.5s	3.8	281.2s	5.0	282.4s	5.0	299.0s	4.7	403.5s	102.8s	3.9	53.8s	7.5	38.1s	10.6	33.2s	12.2

5.5 Comparison of Threaded Results with Theoretical Limits

For the 500 $G(200, 0.9)$ instances, Figure 5.2 plots obtained speedups versus the speedup limit discussed in Section 3.3. For the x position of each graph instance, we perform a sequential run and note the runtime and the size of the maximum clique (i.e. ω and T_{seq}). We then re-run the sequential implementation, priming the incumbent with what we now know to be the size of a maximum clique, to find the time spent visiting ineliminable nodes T_{inelim}. The speedup limit from Equation 3.3 on page 15 is then the number of threads, multiplied by the ratio of the sequential runtime to the primed sequential runtime. For the y position, we take the sequential runtime divided by the threaded runtime to get the actual speedup.

The straight line plots $y = x$. Points below this line are "acceptable", in that they do not exceed our theoretical limit. We see that in some cases we get very close to the line, but we do not pass it[1]. This is empirical evidence that our theory and implementation are in agreement.

Figure 5.2: Achieved speedup vs speedup limit with **tmcsa** and **tbmcsa**, $G(200, 0.9)$, on a 12 core hyper-threaded system. We expect all points to be below the line.



[1]There is one point that is ever so slightly over the line in the graphs for 4 and 8 threads. This is not evidence of a bug: we made a number of simplifying assumptions, such as that every node took the same amount of time to evaluate, and the point is well within the margins of measurement error.

[2]The difference in behaviour seen in the graph for 24 threads, where speedups from **tbmcsa** are consistently lower than those of **tmcsa**, is due to hyper-threading. The benefit obtainable from hyper-threading depends upon patterns of memory accesses and computations, which differ between the two encodings.

5.6 Analysis of Super-Linear Speedups

The behaviour we saw from the DIMACS instance "san400_0.9_1" in Table 5.5 is particularly interesting: we go from a slightly better than linear speedup with four threads, to a speedup of greater than 100 when using twelve or twenty four threads. A more detailed examination of this behaviour is instructive[3]. We plot the total CPU time spent (i.e. runtimes multiplied by the number of threads) for varying numbers of threads in Figure 5.3 on the next page. We also show the total CPU time taken to find a maximum clique but not to prove its optimality (we rerun the program, telling it to exit as soon as it finds a clique of the size we now know to be the maximum), and the total CPU time to find and prove optimality if we split at distance 3 rather than distance 1.

Total CPU time spent, as opposed to runtime, is perhaps not an obvious choice of y-axis. To understand why this information is useful, we should take a moment to consider what we would *expect* to see here for the total CPU times. We refer back to Figure 3.2 on page 16. In the "no speedup" case, we would expect runtimes to remain constant, and so total CPU time would rise linearly with the number of threads. In the "linear speedup" case, we would expect roughly horizontal lines (this is similar to saying that our algorithm is work efficient—we can think of the y-axis as being "work done"). But as we are getting a super-linear speedup, we expect a decrease in the overall CPU time (our algorithm is better than work efficient, so we are doing less total work) as the number of threads increases. That is indeed what happens.

But we can discover more from the graph by looking at the times taken to find but not prove optimality. With a single thread, we see that most of the time spent is on finding a maximum clique, and that proving optimality once such a clique has been found is a relatively simple affair. With eight or more threads, a maximum clique is found almost immediately, and almost all the CPU time is spent proving optimality. Furthermore, with this maximum found, a large portion of the search space that would have be explored in a sequential run is eliminated.

Going beyond eight threads, the total amount of work done is more or less constant. We still gain an increasing speedup by splitting this work between threads, but this is now effectively linear (i.e. the graph is horizontal).

Finally, we see that the line for splitting at distance 3 from the root are much closer to being horizontal, and do not show the same degree of super-linear speedup. This strongly justifies our choice in Section 3.4 of splitting closer to the root: we are right not to commit heavily to an early heuristic choice.

This kind of behaviour is not unique to "san400_0.9_1". A similar effect is observed with several members of the "brock400" family of graphs and the "gen200" instances (we are unable to say whether the "gen400" instances exhibit the same behaviour—we were unable to obtain sequential runtimes at all for these graphs). The second graph plots "brock400_4": we see that the ninth thread finds a maximum reasonably quickly. This matches up with the data in Table 5.5, where we see a speedup of around 9 from 8 threads, but between 30 and 40 for 12 threads. The rising slope of the distance 3 line again validates our decision on splitting distance.

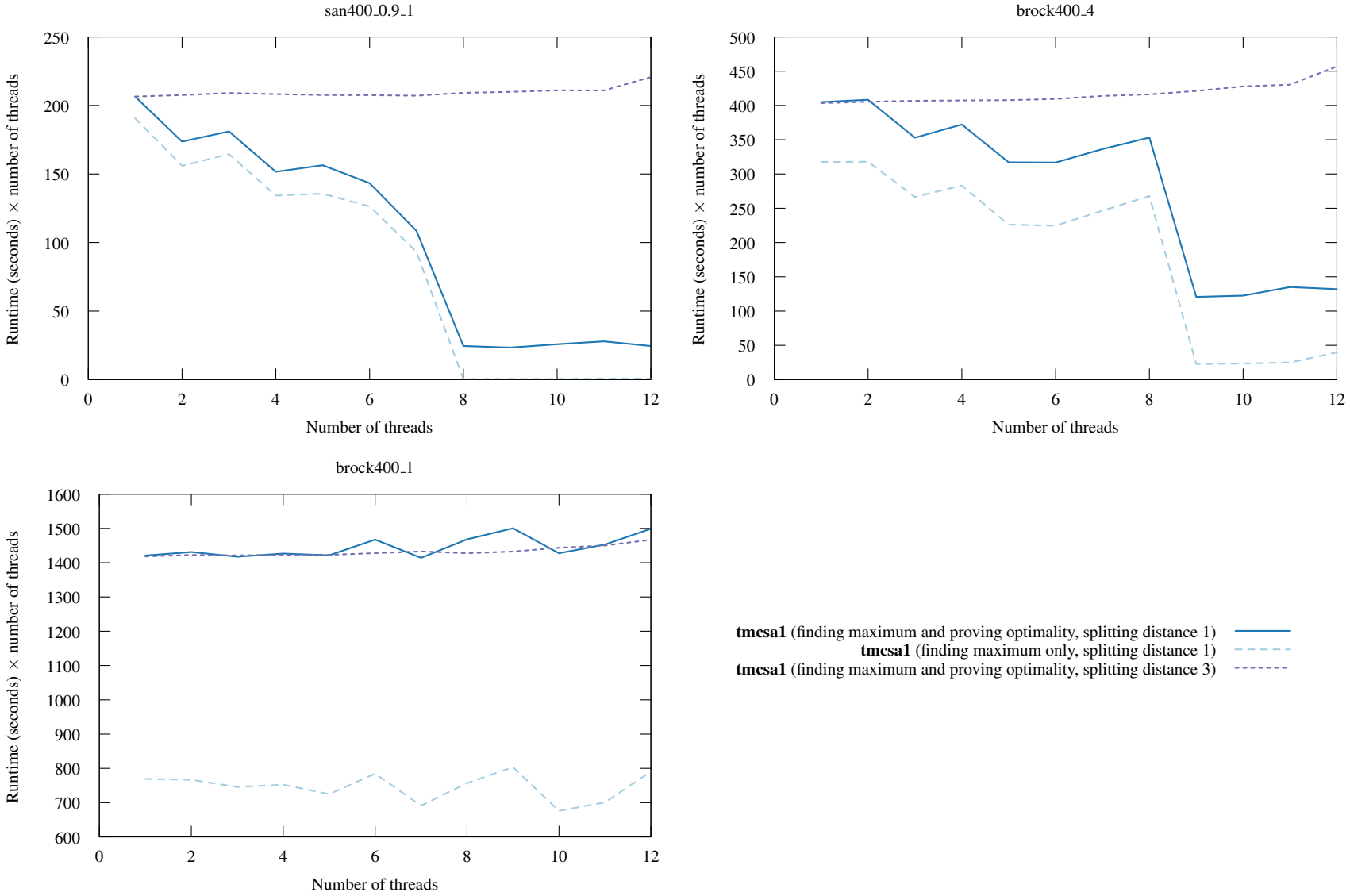
For comparison, we also present "brock400_1", where our speedup is a little below linear. There are two factors at work here. We see that we do not quickly find a maximum clique. Further experiments show that, if we were using at least 20 threads instead of 12, we *would* find a maximum clique quickly—this explains the speedup of over 20 from 24 threads, despite hyper-threading[4]. Even if a maximum is found immediately, though, less than half of the search space can be eliminated, which reduces the possibilities for super-linearity.

These results provide a compelling case in favour of the kind of global parallelism we have introduced. Even in non-ideal cases, we still produce a good speedup. But as predicted, we do not just gain a speedup from exploiting multi-core parallelism; we also make certain problems much easier to solve.

[3] Here we look only at **tmcsa** for clarity. The same pattern is observed with **tbmcsa**.

[4] One may ask why we do not then use more threads than we have cores, to make this more likely to happen. Such an approach would sometimes be of benefit, but only if we are prepared to accept the possibility of a considerable slowdown in other cases.

Figure 5.3: Total CPU time spent (i.e. runtimes multiplied by number of threads) for "san400_0.9_1", "brock400_4" and "brock400_1" from DIMACS with varying numbers of threads. Also shown is the total CPU time taken to find a maximum clique, but not prove its optimality, and the total CPU time to find and prove optimality if we split at distance 3 instead of distance 1. **Horizontal lines correspond to linear speedups**, and downwards sloping lines are super-linear.



Chapter 6

Conclusion

We discussed the maximum clique problem and presented two variations of a state of the art maximum clique algorithm. We discussed standard techniques for parallelising branch and bound algorithms, and introduced new threaded versions of the maximum clique algorithms, together with an analysis of the potential for speedup. We implemented both the sequential algorithms and our threaded versions, and showed that the performance of our sequential implementation is competitive with published results. We evaluated the threaded implementation experimentally on a variety of standard and random benchmarks, and showed that near-linear speedups can consistently be obtained on non-trivial problems, and that super-linear speedups are common.

This is important for two reasons. Firstly, existing work on local parallelism using bitset encodings has produced a speedup of between two and twenty over the basic algorithm. We have shown that a further speedup of around the same magnitude is possible by making use of the resources offered by today's multi-core processors.

Secondly, we have shown that super-linearity can happen in practice, and not just as a rare event. Furthermore, when it does happen, the effects can be extremely significant. This super-linearity occurs only because we designed for it, by using parallelism to offset bad early heuristic choices. Had we not used this approach, we would be repeating Lai and Sahni's claim that "our experimental results indicate that such anomalous behaviour will be rarely witnessed in practice" [LS84].

Our success contradicts some of the more pessimistic claims that have been made in the literature regarding the suitability of sequential maximum clique algorithms for parallelisation: being unable to split the problem into genuinely independent parts is not an insurmountable problem. Previous attempts at parallelising maximum clique solvers either restricted themselves to sparse graphs, or were not taken beyond a very small number of processors. By taking a threaded approach rather than using MPI or OpenMP, and by moving beyond a simple dependency-free work splitting mechanism, we have shown that making use of multi-core parallelism for hard clique problems is possible and worthwhile.

This work could be extended in several directions in the future. The results obtained so far suggest we might expect similar kinds of speedups within families of graphs, and that hard instances may be more amenable to super-linear speedups than easy ones—is this really the case, and if so, why? Or in a different direction, the implementation could be adapted to use a hybrid threaded and MPI approach, to target clusters and allow for even larger problems to be tackled. This is not just an engineering problem: with sufficiently many cores, there may be better work splitting mechanisms available. Finally, it would be interesting to repeat the entire approach with a different problem—say, graph colouring or SAT—and see whether the techniques and results generalise. If they do, we will not just have a way of solving hard problems faster; thanks to super-linear speedups, we can also make some hard problem instances easy.

References

[Amd67] Gene M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, Proceedings of the April 18-20, 1967, spring joint computer conference (New York, NY, USA), AFIPS '67 (Spring), ACM, 1967, pp. 483–485.

[Bai09] David H. Bailey, *Misleading Performance Claims in Parallel Computations*, 2009.

[BB10] Lucio Barreto and Michael Bauer, *Parallel Branch and Bound Algorithm - A comparison between serial, OpenMP and MPI implementations*, Journal of Physics: Conference Series **256** (2010), no. 1, 012018.

[BBPP99] Immanuel M. Bomze, Marco Budinich, Panos M. Pardalos, and Marcello Pelillo, *The maximum clique problem*, Handbook of Combinatorial Optimization (Supplement Volume A) **4** (1999), 1–74.

[BHO] *BHOSLIB: Benchmarks with Hidden Optimum Solutions for Graph Problems*, http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/graph-benchmarks.htm.

[BHP04] David A. Bader, William E. Hart, and Cynthia A. Phillips, *Tutorials on Emerging Methodologies and Applications in Operations Research: Presented at INFORMS 2004, Denver, CO*, International Series in Operations Research & Management Science, ch. Parallel Algorithm Design for Branch and Bound, Springer, 2004.

[BKT95] Arie de Bruin, Gerard A. P. Kindervater, and H. W. J. M. Trienekens, *Asynchronous Parallel Branch and Bound and Anomalies*, Proceedings of the Second International Workshop on Parallel Algorithms for Irregularly Structured Problems (London, UK, UK), IRREGULAR '95, Springer-Verlag, 1995, pp. 363–377.

[BL99] Robert D. Blumofe and Charles E. Leiserson, *Scheduling multithreaded computations by work stealing*, J. ACM **46** (1999), no. 5, 720–748.

[BP04] J.R. Bulpin and I.A. Pratt, *Multiprogramming performance of the Pentium 4 with Hyper-Threading*, Workshop on Duplicating, Deconstructing, and Debunking (WDDD04), 2004.

[Bré79] Daniel Brélaz, *New methods to color the vertices of a graph*, Commun. ACM **22** (1979), no. 4, 251–256.

[BS81] F. Warren Burton and M. Ronan Sleep, *Executing functional programs on a virtual tree of processors*, Proceedings of the 1981 conference on Functional programming languages and computer architecture (New York, NY, USA), FPCA '81, ACM, 1981, pp. 187–194.

[BW05] Holger Bast and Ingmar Weber, *Don't compare averages*, Proceedings of the 4th international conference on Experimental and Efficient Algorithms (Berlin, Heidelberg), WEA'05, Springer-Verlag, 2005, pp. 67–76.

[BW06] Sergiy Butenko and Wilbert E. Wilhelm, *Clique-detection models in computational biochemistry and genomics*, European Journal of Operational Research **173** (2006), no. 1, 1–17.

[CHH91] Scott H. Clearwater, Bernardo A. Huberman, and Tad Hogg, *Cooperative Solution of Constraint Satisfaction Problems*, Science **254** (1991), no. 5035, 1181–1183.

[CKT91] Peter Cheeseman, Bob Kanefsky, and William M. Taylor, *Where the really hard problems are*, Morgan Kaufmann, 1991, pp. 331–337.

[CLT91] Jens Clausen and Jesper Larsson Träff, *Implementation of parallel branch-and-bound algorithms experiences with the graph partitioning problem*, Annals of Operations Research **33** (1991), 329–349 (English).

[CP90] Randy Carraghan and Panos M. Pardalos, *An exact algorithm for the maximum clique problem*, Operations Research Letters **9** (1990), 375–382.

[CZ12] Renato Carmo and Alexandre P. Züge, *Branch and bound algorithms for the maximum clique problem under a unified framework*, J. Braz. Comp. Soc. (2012), 137–151.

[DIM] *DIMACS Implementation Challenges*, http://dimacs.rutgers.edu/Challenges/.

[FLJ86] V. Faber, Olaf M. Lubeck, and Andrew B. White Jr., *Superlinear speedup of an efficient sequential algorithm is not possible*, Parallel Computing **3** (1986), no. 3, 259 – 260.

[Fre86] Karen A. Frenkel, *Complexity and parallel processing: an interview with Richard Karp*, Commun. ACM **29** (1986), no. 2, 112–117, Interviewee-Karp, Richard.

[GC94] Bernard Gendron and Teodor Gabriel Crainic, *Parallel Branch-and-Branch Algorithms: Survey and Synthesis*, Operations Research **42** (November/December 1994), no. 6, 1042–1066.

[GJ90] Michael R. Garey and David S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, NY, USA, 1990.

[Gro95] UEA Calma Group, *Parallelism in Combinatorial Optimisation*, 1995.

[Gus88] John L. Gustafson, *Reevaluating Amdahl's law*, Commun. ACM **31** (1988), no. 5, 532–533.

[HBK06] Jim Held, Jerry Bautista, and Sean Koehl, *From a Few Cores to Many: A Tera-scale Computing Research Overview*, Tech. report, 2006.

[HG95] W.D. Harvey and M.L. Ginsberg, *Limited discrepancy search*, International Joint Conference on Artificial Intelligence, vol. 14, LAWRENCE ERLBAUM ASSOCIATES LTD, 1995, pp. 607–615.

[HHM08] J. Harris, J.L. Hirst, and M. Mossinghoff, *Combinatorics and Graph Theory*, Undergraduate Texts in Mathematics, Springer, 2008.

[HM90] D. P. Helmbold and C. E. McDowell, *Modeling Speedup (n) Greater than n*, IEEE Trans. Parallel Distrib. Syst. **1** (1990), no. 2, 250–256.

[IEE95] *Information Technology — Portable Operating Systems Interface (POSIX) — Part: System Application Program Interface (API) — Amendment 2: Threads Extension [C Language]*, IEEE Standard 1003.1c–1995, Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1995.

[ISO11] ISO, *ISO/IEC 9899:2011 Information technology — Programming languages — C*, International Organization for Standardization, Geneva, Switzerland, December 2011.

[ISO12] ———, *ISO/IEC 14882:2011 Information technology — Programming languages — C++*, International Organization for Standardization, Geneva, Switzerland, February 2012.

[Kar72] Richard M. Karp, *Reducibility Among Combinatorial Problems.*, Complexity of Computer Computations (Raymond E. Miller and James W. Thatcher, eds.), The IBM Research Symposia Series, Plenum Press, New York, 1972, pp. 85–103.

[KZ93] Richard M. Karp and Yanjun Zhang, *Randomized parallel algorithms for backtrack search and branch-and-bound computation*, J. ACM **40** (1993), no. 3, 765–789.

[Lee06] Edward A. Lee, *The problem with threads*, COMPUTER **39** (2006), 2006.

[LS84] Ten-Hwang Lai and Sartaj Sahni, *Anomalies in parallel branch-and-bound algorithms*, Commun. ACM **27** (1984), no. 6, 594–602.

[LW86] Guo-Jie Li and Benjamin W. Wah, *Coping with anomalies in parallel branch-and-bound algorithms*, IEEE Trans. Comput. **35** (1986), no. 6, 568–573.

[MBH+02] D.T. Marr, F. Binns, D.L. Hill, G. Hinton, D.A. Koufaty, J.A. Miller, and M. Upton, *Hyper-threading technology architecture and microarchitecture*, Intel Technology Journal **6** (2002), no. 1, 4–15.

[MG85] Ravi Mehrotra and Edward F. Gehringer, *Superlinear speedup through randomized algorithms*, ICPP'85, 1985, pp. 291–300.

[MM11] Cristopher Moore and Stephan Mertens, *The Nature of Computation*, Oxford University Press, USA, July 2011.

[Mor02] Bernard ME Moret, *Towards a discipline of experimental algorithmics*, Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges, vol. 59, AMS, 2002, pp. 197–213.

[MP12] Ciaran McCreesh and Patrick Prosser, *Distributing an Exact Algorithm for Maximum Clique: maximising the costup*, ArXiv e-prints (2012).

[MPI94] *MPI: A Message-Passing Interface Standard*, Tech. Report UT-CS-94-230, University of Tennessee, Knoxville, TN, USA, May 1994.

[MRR12] M. McCool, J. Reinders, and A. Robison, *Structured parallel programming: Patterns for efficient computation*, Elsevier Science, 2012.

[Ope11] *OpenMP Application Program Interface Version 3.1*, Tech. report, 2011.

[PMB11] Wayne Pullan, Franco Mascia, and Mauro Brunato, *Cooperating local search for the maximum clique problem*, Journal of Heuristics **17** (2011), 181–199 (English).

[PPG+12] B. Pattabiraman, M. M. A. Patwary, A. H. Gebremedhin, W.-k. Liao, and A. Choudhary, *Fast Algorithms for the Maximum Clique Problem on Massive Sparse Graphs*, ArXiv e-prints (2012).

[PR90] P.M. Pardalos and Rodgers, *Parallel branch and bound algorithms for quadratic zeroone programs on the hypercube architecture*, Annals of Operations Research **22** (1990), 271–292 (English).

[Pro12] Patrick Prosser, *Exact Algorithms for Maximum Clique: A Computational Study*, Algorithms **5** (2012), no. 4, 545–587.

[PRR98] P.M. Pardalos, J. Rappe, and M.G.C. Resende, *An exact parallel algorithm for the maximum clique problem*, High performance algorithms and software in nonlinear optimization. Kluwer Academic Publishers (1998).

[PU11] P. Prosser and C. Unsworth, *Limited discrepancy search revisited*, Journal of Experimental Algorithmics **16** (2011), Article 1.6.

[Rou87] Catherine Roucairol, *A parallel branch and bound algorithm for the quadratic assignment problem*, Discrete Applied Mathematics **18** (1987), no. 2, 211 – 225.

[SL05] Herb Sutter and James Larus, *Software and the Concurrency Revolution*, Queue **3** (2005), no. 7, 54–62.

[SMRLH11] Pablo San Segundo, Fernando Matia, Diego Rodríguez-Losada, and Miguel Hernando, *An improved bit parallel exact maximum clique algorithm*, Optimization Letters (2011).

[Spe89] Ewald Speckenmeyer, *Is average superlinear speedup possible?*, CSL '88 (Egon Börger, HansKleine Büning, and Michael M. Richter, eds.), Lecture Notes in Computer Science, vol. 385, Springer Berlin Heidelberg, 1989, pp. 301–312.

[SSRLJ11] Pablo San Segundo, Diego Rodríguez-Losada, and Agustín Jiménez, *An exact bit-parallel algorithm for the maximum clique problem*, Comput. Oper. Res. **38** (2011), no. 2, 571–581.

[SSTP09] Matthew C. Schmidt, Nagiza F. Samatova, Kevin Thomas, and Byung-Hoon Park, *A scalable, parallel algorithm for maximal clique enumeration*, Journal of Parallel and Distributed Computing **69** (2009), no. 4, 417 – 428.

[Sut05] Herb Sutter, *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*, Dr. Dobb's Journal **30** (2005), no. 3.

[Sut08] Herb Sutter, *Going Superlinear*, Dr. Dobb's Report (2008).

[Sza11] S Szabó, *Parallel algorithms for finding cliques in a graph*, Journal of Physics: Conference Series **268** (2011), no. 1, 012030.

[TK07] E. Tomita and T. Kameda, *An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments*, Journal of Global Optimization **37** (2007), no. 1, 95–111.

[TPO10] Stanley Tzeng, Anjul Patney, and John D. Owens, *Task management for irregular-parallel workloads on the GPU*, Proceedings of the Conference on High Performance Graphics (Aire-la-Ville, Switzerland, Switzerland), HPG '10, Eurographics Association, 2010, pp. 29–37.

[TS03] Etsuji Tomita and Tomokazu Seki, *An efficient branch-and-bound algorithm for finding a maximum clique*, Proceedings of the 4th international conference on Discrete mathematics and theoretical computer science (Berlin, Heidelberg), DMTCS'03, Springer-Verlag, 2003, pp. 278–289.

[TSH+10] E. Tomita, Y. Sutani, T. Higashi, S. Takahashi, and M. Wakatsuki, *A Simple and Faster Branch-and-Bound Algorithm for Finding a Maximum Clique*, WALCOM 2010, LNCS 5942, 2010, pp. 191–203.

[WE12] Anthony Williams and Vicente J. Botet Escriba, *Boost Thread Library, 3.1.0*, http://www.boost.org/doc/libs/1_52_0/doc/html/thread.html, November 2012.

[Wil12] Anthony Williams, *C++ Concurrency: Practical Multithreading*, Manning Pubs Co, 2012.

[WP67] 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** (1967), no. 1, 85–86.

[Zuc06] David Zuckerman, *Linear degree extractors and the inapproximability of max clique and chromatic number*, Proceedings of the thirty-eighth annual ACM symposium on Theory of computing (New York, NY, USA), STOC '06, ACM, 2006, pp. 681–690.