A Parallel implementation for the maximum clique problem

L.R. Thimm, D.L. Kreher, and P. Merkey
Department of Mathematical Sciences
Michigan Technological University
Houghton MI 49931-1295

Abstract

We discuss a parallel programming method for solving the maximum clique problem. We use the partitioned shared memory programing language, Unified Parallel C, for the parallel implementation. The problem of load balancing is discussed and the *stealstack* is used to solve this problem. Implementation details are provided.

1 Introduction

Let $G = (V, E)$ be an undirected graph with no repeated edges and no loops. A *clique* in G is a subgraph in which all vertices are pairwise adjacent. A clique is *maximal* if there is no vertex that can extend the clique, i.e., there are no other vertices adjacent to every vertex of the clique. A clique of largest possible size is a *maximum clique* and a graph may have more than one. If a subset $S \subseteq V$ of the vertices induces a clique we say that S forms a clique.

In this manuscript a parallel programming method for solving the maximum clique problem is discussed. Beginning with a serial algorithm for finding a maximum clique in a graph we develop a parallel algorithm to solve the same problem.

1.1 The Maximum Clique Algorithm

In their book [1] Kreher and Stinson describe a deterministic serial algorithm for finding a maximum clique in a simple graph. Pseudocode for their algorithm, MAXCLIQUE2(ℓ), is given as Algorithm 1.1. The parallel algorithm and implementation discussed in this paper are based on Algorithm 1.1. The MAXCLIQUE2(ℓ) algorithm performs a depth-first search using bounding functions to prune the state space tree and reduce search time.

Let $G = (V, E)$ be a graph on n vertices. Let $V = \{0, 1, \ldots n-1\}$ and let A_m be the set of vertices in G adjacent to vertex m. MAXCLIQUE2(ℓ) begins with a clique, X, of size ℓ with $X = [x_0, x_1, \ldots, x_{\ell-1}]$ and computes a set of vertices that can potentially extend X to a clique of size $\ell + 1$. A clique of size k can be written down in $k!$ ways, so a naive algorithm would find the same clique $k!$ times, slowing down runtime. To prevent this an ordering is imposed on the vertices. Different orderings may produce different sets of vertices as the first maximum clique found. We use the natural ordering $0 < 1 < \ldots < n-1$. The MAXCLIQUE2(ℓ) algorithm considers as possible vertices only those that are greater than the largest vertex in the current clique X. Proceeding in this manner will produce a clique X such that $x_0 < x_1 < \ldots < x_{\ell-1}$. Let B_m be the set of vertices greater than vertex m. The *extension set*, the set of vertices that can potentially extend the clique X of size ℓ to a clique of size $\ell + 1$, includes all vertices adjacent to every vertex of X that are also greater than $x_{\ell-1}$. Call this extension set C_ℓ. Then

$$C_\ell = A_{x_{\ell-1}} \cap B_{x_{\ell-1}} \cap C_{\ell-1}$$

is the set of possibilities for x_ℓ.

Our goal is to find a maximum clique, so there is no need to record every clique found by the algorithm. Instead, only the largest clique currently found is saved. Every clique found is compared to the current largest clique, `OptClique`, and if a larger clique is found it becomes the new `OptClique`. The variable `OptSize` is the size of the current optimal clique.

To reduce the number of nodes in the state space tree a bounding function, BOUND(X), can be included in the algorithm. The simplest bounding function adds the number of vertices in the extension set C_ℓ to the size of the current clique X of size ℓ. If this sum is less than or equal to the size of the current optimal clique, it is impossible to form a larger clique and there is no need to continue processing X. This is called the *size bound*. MAXCLIQUE2(ℓ), complete with bounding function, is given in Algorithm 1.1.

Algorithm 1.1: MAXCLIQUE2(ℓ)

```
external BOUND()
global $A_m, B_m, C_m$   $(m = 0, 1, \ldots, n-1)$
if $\ell >$ OptSize
  then { OptSize $\leftarrow \ell$
       { OptClique $\leftarrow [x_0, ..., x_{\ell-1}]$
if $\ell = 0$
  then $C_\ell \leftarrow V$
  else $C_\ell \leftarrow A_{x_{\ell-1}} \cap B_{x_{\ell-1}} \cap C_{\ell-1}$
$M \leftarrow$ B($[x_0, ..., x_{\ell-1}]$)
for each $x \in C_\ell$
  do { if $M \leq$ OptSize
     {   then return ()
     { $x_\ell \leftarrow x$
     { MAXCLIQUE2($\ell + 1$)
main
  OptSize $\leftarrow 0$
  MAXCLIQUE2(0)
  output (OptClique)
```

Other bounding functions produce better results than the simple size bound described above. Kreher and Stinson in [1] examined two additional bonding functions, the *greedy bound* and the *sampling bound*. Evaluating these three bounding functions Kreher and Stinson found the greedy bound to yield the best pruning of the state space tree. At each step the greedy bound greedily colors the vertices of the extension set C_ℓ such that no two adjacent vertices receive the same color. The number of colors needed gives an upper bound on the number of vertices that can be added to the current clique.

The third bound discussed by Kreher and Stinson is the sampling bound. It performs a greedy coloring of all the vertices in the graph before the search begins. At each step the number of colors used on the vertices of C_ℓ is added to the size of the current clique.

1.2 Example: MaxClique2

Consider the graph displayed in Figure 1. If the sampling bound were being used it would perform an initial greedy coloring of the graph. The vertices {0,2,4} would be given color 0, vertices {1,3} color 1, vertices {5,6} color 2, and vertex 7 would have color 3. At some point $X = [3]$ is the current clique with size $\ell = 1$. Then $A_3 = \{2, 4, 5, 6, 7\}$, $B_3 = \{4, 5, 6, 7\}$, and $C_2 = \{4,5,6,7\}$. We can compare the results of the size, greedy, and sampling bounds. Using the size bound we know that X can be extended into a clique of size at most $\ell + |C_\ell| = 5$. The greedy bound would color the vertices of C_2 as follows: vertex 4 is color 0, vertices 5 and 6 are color 1, and vertex 7 is color 2. The upper bound would therefore be four. The sampling bound used three colors to color the vertices in C_2, and would therefore also give a bound of four.

In the next step $X = [3,4]$ is the current clique with size $\ell = 2$. Then $A_4 = \{3,5,6,7\}$, $B_4 = \{5,6,7\}$ and $C_3 = A_4 \cap B_4 \cap C_2 = \{5,6,7\}$. The size bound still says that X can be extended into a clique of size at most five. Using the greedy bound, which would color the vertices of C_3 with two colors, we still have an upper limit of four vertices. The sampling bound colored the vertices in C_3 with two colors, giving an upper bound of four.

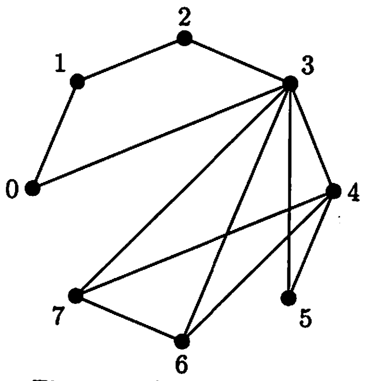

Figure 1: A sample graph

1.3 Parallel Computing

Traditionally computers have been composed of one central processing unit, CPU, simply called a processor, which can process one instruction at a time. This kind of machine is called a serial machine, and programs written for it are serial programs. Parallel computing, in contrast to serial computing, takes advantage of multiple processes working concurrently to solve the same problem. These processes can run on either a single or multiple processors, which in turn might be part of a single machine, may be located on many individual machines which have been joined into a cluster, or a combination of the two.

Consider 10 integers, $a_0, a_1, b_0, b_1, \ldots, e_0, e_1$, where every pair x_0, x_1 needs to be added. A serial machine would require five steps to complete the computation, one for each addition. If the addition were done on a parallel machine with five processes, where each process was assigned a pair of numbers, the computation would only take one step. During that one step each process would add the pair of numbers assigned to it.

One motivation for computing in parallel is the potential speed increase, as seen in the simple addition example. If one processor can complete a task in five minutes, then perhaps five processors working together can complete the same task in one minute. The speed increase, however, may not be proportional to the number of processors for many reasons. Communication between the processors creates overhead not present in a serial machine. If the problem is not easily broken into equal pieces there may be load balancing issues; some processors may finish early and sit idle while the others continue executing.

1.4 Partitioned Shared Memory Model

Before presenting the parallel algorithm for solving the maximum clique problem we will discuss the programming model used for the design and implementation. The partitioned shared memory model follows the single program, multiple data (SPMD) programming model. The user writes only one piece of code. During execution many copies of this single program are run simultaneously. Each instance of the program is called a thread. To manage the threads each is given a unique identifier, called MYTHREAD. The value of MYTHREAD is an integer from 0 to THREADS − 1, where THREADS is the total number of threads executing simultaneously. During execution each instance can use its value of MYTHREAD, as well as the value of THREADS, to make decisions within its own execution path.

As an example, consider the code below.

```
if MYTHREAD == 0
  then READFILE()
```

When each thread reaches this section of code it will consider the **if** statement. Only the thread with MYTHREAD equal to zero executes the function READFILE( ). All other threads skip that statement.

The partitioned shared memory model divides available memory among the threads. Each thread is said to have *affinity* to the memory associated with it. Available memory is, at the same time, divided into shared and private memory. Shared memory can be accessed by all threads, while private memory is available only to the thread with affinity to that memory. As a result of these two divisions each thread has affinity to some of the shared memory, as well as its own private memory. See Figure 2 for a picture of the partitioned memory. The memory above thread T_i represents the memory to which the thread has affinity.

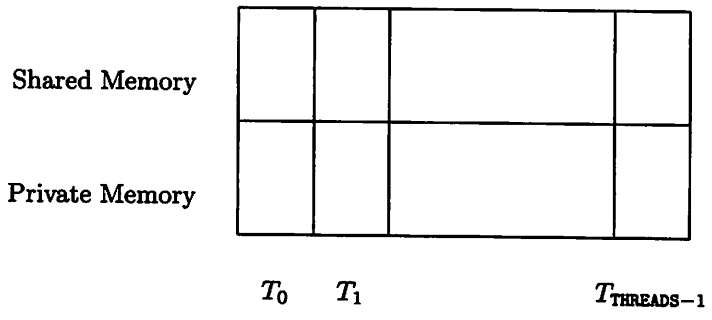

Figure 2: UPC Memory allocation

The partitioned shared memory model is architecture independent. Algorithms and programs written with this model in mind can be implemented on any machine, regardless of where the physical memory actually resides. As well as producing highly portable code, the partitioned shared memory model makes communication between threads simple for the user to deal with, improving ease of use.

Because all threads execute on their own clocks and take different execution paths through the program there is no notion of instruction-by-instruction synchronization. All synchronization must be done explicitly. There are several ways to do this, but the principle way is to use a barrier, or BARRIER( ). Once a thread reaches a barrier it cannot continue with execution until all other threads have reached the barrier. See Example 1.5 for an example of the effects of a BARRIER( ) in program execution.

One way to protect segments of code, and consequently access to shared variables, is to use locks. A *lock* is an object that can be held by only one thread at a time. Before accessing a certain area of shared memory a thread can be required to obtain a lock associated with that area. When the thread is finished it releases the lock. Until the lock is released any other thread attempting to acquire it must wait. Provided that all sections of code that read from or write to a specific area of memory are protected by the same lock, threads are prevented from reading memory that is concurrently being written to, and vice versa.

Unified Parallel C, or UPC, is an extension of the ANSI C programming language [5] that is based on the partitioned shared memory model. One of the main benefits of UPC as a parallel language is its ease of use. Because it is based on the C programming language, almost all syntax is familiar to C programmers. MPI, a library that handles explicit message passing,

```
0 likes bread and butter.
2 likes bread and butter.
1 likes bread and butter.
0 likes toast and jam.
2 likes toast and jam.
3 likes bread and butter.
3 likes toast and jam.
1 likes toast and jam.
```

Figure 3: Output without barrier

allows processes to send each other copies of the variables they have under their control. UPC, in contrast, achieves communication by references to shared variables, sometimes called implicit message passing. Users can write assignment statements without knowing which thread controls each variable.

1.5 Example: Barriers

Consider the simple code given below, compiled on four threads.

```
output (MYTHREAD, "likes bread and butter")
output (MYTHREAD, "likes toast and jam")
```

The output that results from running this code is given in Figure 3. Notice that although every thread executes its own set of instructions in order, stating a preference for bread and butter before stating a preference for toast and jam, there is no synchronization between the threads. The output depends entirely on the speed of each thread.

Now consider the modified code below and the resulting output given in Figure 4.

```
output (MYTHREAD, "likes bread and butter")
BARRIER()
output (MYTHREAD, "likes toast and jam")
```

When a thread reaches a barrier in its execution it cannot continue until all threads have reached the barrier. Although before and after the barrier the order of execution is based on the speed of each thread, no instructions before or after the barrier can be interchanged. In other words,

```
0 likes bread and butter.
2 likes bread and butter.
1 likes bread and butter.
3 likes bread and butter.
0 likes toast and jam.
2 likes toast and jam.
3 likes toast and jam.
1 likes toast and jam.
```

Figure 4: Output with barrier

all threads must make their fondness for bread and butter known before any can mention toast and jam.

1.6 Example: Variable Declaration

The following variable declarations, compiled to run on four threads, result in the placement shown in Figure 5. The keyword **shared** delcares a variable to be shared, and places one copy of it in shared memory. In the absence of the **shared** key word the variable is by default local, and each thread receives a copy. Notice that the array of stacks, `S[4]`, has been distributed among the threads' shared memory.

```
shared int p
shared stack S[4]
graph G
```

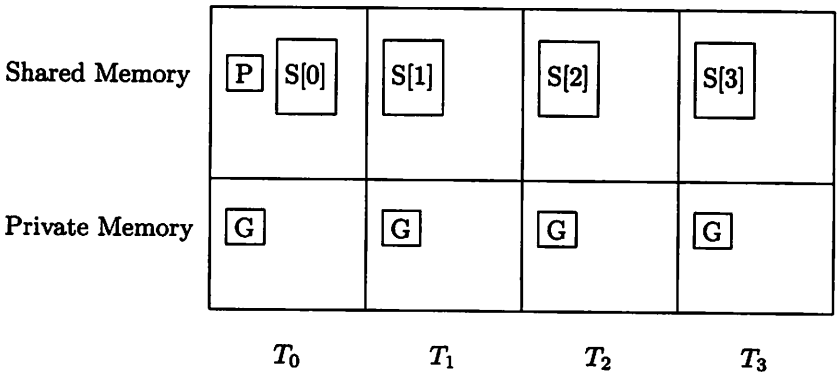

Figure 5: UPC Declarations

1.7 The State Space Tree and Load Balancing

To obtain good performance from parallel computation it only makes sense that all processors, or UPC threads, should be busy during the entire execution of the program. Idle threads contribute little to the speed increases we look for in a parallel implementation. Not all algorithms are capable of being broken into smaller pieces and are therefore not suitable for parallel computation. Other algorithms may be granular enough yet still suffer from load balancing problems for various reasons. To obtain an efficient algorithm the workload between the threads must be balanced.

To better understand the potential for uneven work distribution in a parallel implementation it helps to consider the *state space tree* of the backtrack algorithm. This tree is essentially a picture of the search conducted by the algorithm, with each vertex or node on the tree representing a clique considered and analyzed by the algorithm. By considering the state space tree of the complete graph on n vertices we can see that any state space tree is at most n nodes deep, and if only vertices of higher order are considered in the extension set, there are at exactly 2^n nodes in the tree. As an example consider Figure 6, the state space tree of the serial algorithm MAXCLIQUE2(ℓ) for the graph in Figure 1 that results if no bounding function is used.

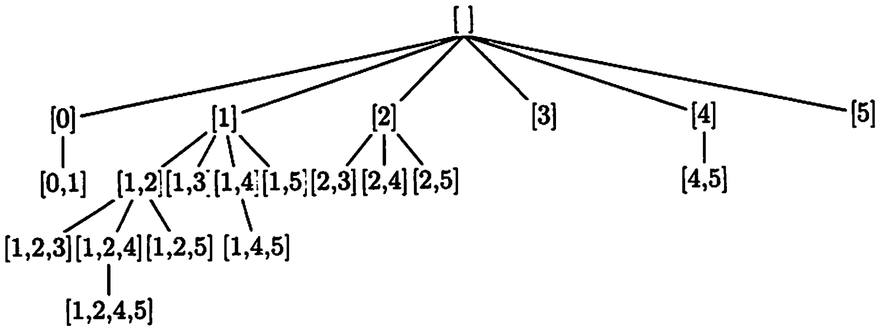

Figure 6: State space tree for the graph in Figure 1

A naive approach to distributing the work would be to assign the threads branches of the tree beginning with the cliques of size one. Consider again the graph in Figure 1 and the corresponding state space tree in Figure 6. If, for example, there were three threads searching we could assign the branches beginning with cliques [0] and [3] to thread T_0, branches beginning with cliques [1] and [4] to thread T_1, and branches with cliques [2] and [5] to thread T_2. Let each thread have a private set S such that S is the set of cliques of size one assigned to that thread. Each thread would execute the MAXCLIQUE2(ℓ) algorithm just as in the serial version, with a few small changes. Instead of making the first call to MAXCLIQUE2(ℓ) with the current clique X being the empty clique, each thread would make their first call with the current clique X set to one of the cliques of size one in S. Each thread would repeat this for every element of their set S. Pseudocode for this naive parallel approach is given as Algorithm 1.6, with the initial distribution of cliques done modulus the number of threads. Any other simple distribution could also be used. Note that only one thread needs to set the value of `OptSize` to zero. Note also that each thread must acquire a lock before accessing the shared variables `OptSize` and `OptClique`.

Processing the graph in Figure 1 using Algorithm 1.6 on three threads results in thread T_1 having potentially much more work than either of the other two threads. In general the branches toward the left of the state space tree will generate more work than branches toward the right of the tree, because only vertices of higher order are considered. The pruning, or bounding, functions make the unbalance even more extreme. Partial cliques toward the right of the tree are more likely to have smaller extension sets, and those branches will be pruned sooner.

Although it is easy to say that the branches on the left will usually provide more work, it is impossible to say exactly which branches will produce more work, and how much more. In Figure 6, for example, the branch of the state space tree beginning with the clique [0] terminates immediately, while the branch beginning with the clique [1] results in a maximum clique. This means that any initial division of work cannot guarantee that no thread will become idle while others continue processing. What is needed is a way to redistribute work while processing. We chose to redistribute the work using a stealstack.

```
Algorithm 1.6: PARMAXCLIQUE($\ell$)

external BOUND()
Shared OptClique, Optsize
global $A_m, B_m, C_m,\quad (m = 0, 1, \ldots, n-1)$
LOCK()
if $\ell >$ OptSize
  then { OptSize $\leftarrow \ell$
       { OptClique $\leftarrow [x_0, \ldots, x_{\ell-1}]$
UNLOCK()
if $\ell = 0$
  then $C_\ell \leftarrow V$
  else $C_\ell \leftarrow A_{x_{\ell-1}} \cap B_{x_{\ell-1}} \cap C_{\ell-1}$
$M \leftarrow$ BOUND($[x_0, \ldots, x_{\ell-1}]$)
for each $x \in C_\ell$
  do { if $M \leq$ OptSize
     {   then return ()
     { $x_\ell \leftarrow x$
     { PARMAXCLIQUE($\ell + 1$)
main
 if MYTHREAD == 0
   then OptSize $\leftarrow 0$
 for $s \leftarrow 0$ to $n - 1$
   do if $i$/THREADS == MYTHREAD
   then $S \leftarrow S \cup s$
 for each $s \in S$
   do { $x_0 \leftarrow s$
      { $C_0 \leftarrow A_s$
      { PARMAXCLIQUE(1)
 output (OptClique)
```

2 The Stealstack

2.1 Theoretical Overview

The *stealstack* was first implemented by Prins et al. as a means of dynamically balancing the work load of an unbalanced search tree, see [2]. Each thread has affinity to a stack and an associated lock. The stacks and their locks are declared in shared memory, allowing all threads access to each stack. The nodes on a stack are data structures that correspond to nodes in the state space tree. Each node contains enough information to start a

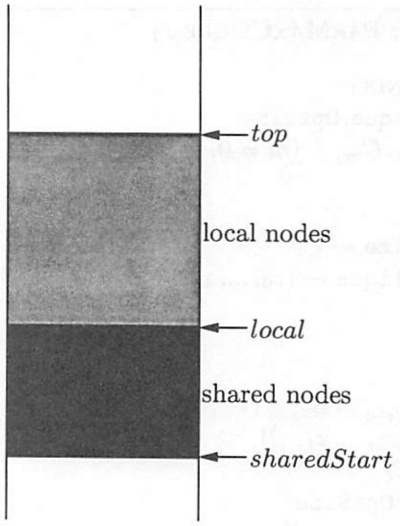

Figure 7: A stack

search as a root of a subtree of the complete state space tree. These nodes are divided into two categories. The nodes on the top of the stack are called the *local nodes*, and those on the bottom are the *shared nodes*. The local and shared portions of a stack should not be confused with private and shared memory in UPC. The local nodes are for processing only by the thread with affinity to the specific stack. Shared nodes are free to be stolen by any idle thread. The lock associated with a stack controls access to the shared nodes of the stack. See Figure 7.

When a thread anticipates that it will have too much work, a decision that is discussed later, it pushes jobs onto the top of the local portion of its stack. A thread can also release nodes from the local to shared portion of its stack, freeing them to be stolen. When the thread finishes processing its current work, it attempts to pop a job off the top of its local stack. If the local stack is empty the thread will attempt to move nodes from the shared portion of its stack to the (empty) local portion. The thread first acquires its stack lock, the lock associated with its particular stack. Once the lock has been acquired the thread checks to see that there are nodes in the shared portion, and if so moves some of them to its local portion and releases the lock. If the thread acquires the lock and discovers that its shared portion is also empty, it releases the lock and attempts to steal nodes from another thread.

The idle thread begins the stealing process by examining the other threads' stacks, searching for one with nodes in the shared portion of its stack. When the idle thread finds such a stack, it attempts to acquire the lock associated with it. Once the lock is acquired the idle thread first makes sure that there are still nodes available, and if so moves them to the local portion of its own stack, then releases the lock. Any number of polling methods can be used. The simplest way for thread T_i to find a thread with work to steal is to begin with thread T_{i+1} (mod THREADS) and loop through all threads sequentially. The polling need not be sequential, however. It could follow a random pattern, a predetermined permutation of the threads, or any cycle the user determines will produced better results. The number of times a thread polls is also open to optimization by the user.

Having the idle threads look for work is an advantageous feature of the stealstack, and why it was chosen to solve this problem. Some load-balancing models have busy threads pause their computation to find idle threads to give their work to.

Figure 8 summarizes the functions associated with the stealstack. For further implementation issues associated with stealstack functions see Prins [2].

2.2 MaxClique

We now incorporate the stealstack into the MAXCLIQUE2(ℓ) algorithm. Each node, or job, on the stack contains three elements: a clique, its size, and its extension set. Let *temp* be a node. We will denote node *temp*'s clique by *temp.X*, the size of the clique by *temp.L*, and the extension set by *temp.C*. The node is also denoted by $temp_{X,C,L}$.

At some point the thread needs to decide if it has too much work, and if so, what work it will put on its stack. Let TMW() be a function that returns **true** if the thread has too much work, and **false** otherwise. A particular implementation of TMW() is discussed Section 3.2.

In MAXCLIQUE2(ℓ) each element of the extension set C_ℓ is included, in turn, in the current clique, and the search continues from there. In the parallel version, Algorithm 2.1, if the thread determines that it has too much work it includes the first element of C_ℓ in the current clique. Before processing this new clique the algorithm includes each additional element of C_ℓ in the previous clique and pushes the job, the new clique and its size and extension set, onto its stack. At this point the thread also checks the number of jobs in the local portion of its stack. The function FULL() returns **true** if there are more than enough jobs in the local portion, and **false** otherwise. An implementation of FULL() is discussed in Section 3.2. If FULL()= **true** then the thread releases some of the nodes from the local to the shared portion of its stack, freeing them to be stolen. It then proceeds with the new current clique. See Example 2.3.

Function Name	Action Taken	Return Value
PUSH(&someNode)	the work in someNode is pushed on top of the stack	none
POP(&someNode)	the work from the node on the top of the stack is put into someNode	none
LOCALDEPTH()		number of nodes in local part of stack
RELEASE(k)	k nodes are moved from local to shared part of stack	none
ACQUIRE(k)	k nodes are moved from shared to local part of stack	TRUE if k nodes are moved from shared to local part of stack, else FALSE
STEAL(t, k)	k nodes moved from bottom of thread t's shared portion to top of local portion	TRUE if k nodes are taken from thread t's stack, else FALSE
FINDWORK(k)		number of a thread with at least k nodes in shared part, -1 if no such thread

Figure 8: Stealstack Functions

The algorithm must make an initial distribution of the work. In the serial version the algorithm essentially starts with nothing. It has no current clique, so its first action is to create C_0 as the set of all vertices. Each of these will at some point become the first element of a clique. In the parallel version it makes sense to begin by distributing these cliques of size one to all threads. Giving them all work to start on immediately saves time that would be lost if only one thread began working and all others had to wait for that thread to generate enough work for them to steal.

```
Algorithm 2.1: MAXCLIQUE($\ell$)

external { BOUND(), TMW()
         { PUSH($c$), FULL()
         { RELEASE($k$)
global $A_\ell, B_\ell, C_\ell$
LOCK()
if $\ell >$ OptSize
  then { OptSize $\leftarrow \ell$
       { OptClique $\leftarrow [x_0, \ldots, x_{\ell-1}]$
UNLOCK()
if $\ell = 0$
  then $C_\ell \leftarrow V$
  else $C_\ell \leftarrow A_{x_{\ell-1}} \cap B_{x_{\ell-1}} \cap C_{\ell-1}$
$M \leftarrow$ BOUND($[x_0, \ldots, x_{\ell-1}]$)
if TMW()
  then { if $M \leq$ OptSize
       {   then return ()
       { $x_\ell \leftarrow$ the first $x \in C_\ell$
       { for each additional $x \in C_\ell$
       {   do $push(x, C_\ell)$
       { if FULL()
       {   then RELEASE($k$)
       { MAXCLIQUE($\ell + 1$)
  else for each $x \in C_\ell$
    do { if $M \leq$ OptSize
       {   then return ()
       { $x_l \leftarrow x$
       { MAXCLIQUE($\ell + 1$)
```

Algorithm 2.1 is a modified version of Algorithm 1.1. It differs in that the conditional TMW(), the subsequent addition of nodes to the stack, and the conditional FULL() and following RELEASE(k) have been added. Although this now allows threads to add work to their stack, and even release work for stealing, it does not allow a thread to remove work from its own stack, or steal from other threads. An algorithm for enabling a thread to empty its stack will be addressed first. Consider Algorithm 2.2. The initial jobs, all cliques of length one, are distributed among the threads and placed on their local stacks. In the function DFS() a node is removed from the local stack and processed completely. During the processing other nodes may or may not be added to the local stack. When the thread finishes with that portion of the graph it begins on the next node in its local stack. When the local portion is empty the thread attempts to move nodes from its shared to local portion. The function ACQUIRE(a) returns **true** if a nodes were moved from the shared to local portion, and **false** if there were not a nodes to move. When DFS() completes the thread has no nodes in either its local or shared stack, and is out of work.

Algorithm 2.2: DFS()

```
external { MAXCLIQUE()
         { POP(c)
repeat
  while LOCALDEPTH() > 0
       { temp ← POP()
    do { X ← temp.X
       { $C_{\ell+1}$ ← temp.C
       { MAXCLIQUE(temp.L)
  moreWork ← ACQUIRE(a)
until moreWork = FALSE
main
  for each $x < n, x \equiv$ MYTHREAD( MOD $n$)
    do PUSH($c_{x,1,B_x}$)
  OptSize ← 0
  BARRIER()
  DFS()
  output (OptClique)
```

Now what is needed to balance the load is an algorithm that incorporates stealing. Consider Algorithm 2.3. The function DFS() empties both the local and shared portion of a thread's stack. Once that happens the thread needs to find work to steal, and begins searching. The function FINDWORK() polls the other threads using whatever method is decided upon, and returns the number of a thread with enough work to steal. If no other thread has enough nodes in its shared portion FINDWORK() returns -1 and the thread exits. Once a thread with potential work is found the nodes must be moved from its shared stack to the local stack of the searching thread. The function STEAL(victimID,k) returns **true** if k nodes are successfully stolen from thread $T_{victimId}$ and **false** otherwise. Because another thread may have moved the nodes to its own stack between the time the searching thread locates the busy thread and attempts to move the nodes, merely finding a thread with enough work is not a guarantee that a thread will be able to steal the work.

```
Algorithm 2.3: EXPLOREFOREST()

external  { DFS(), FINDWORK()
          { STEAL($t, k$)
DFS()
$victimId \leftarrow$ FINDWORK()
while $victimId != -1$
   do { if STEAL($victimId, k$)
      {   then DFS()
      { $victimId \leftarrow$ FINDWORK()
main
 for each $x < n, x \equiv$ MYTHREAD( MOD $n$)
   do PUSH($c_{x,\ell,B_x}$)
 OptSize $\leftarrow 0$
 BARRIER()
 EXPLOREFOREST()
 output (OptClique)
```

2.3 Example: Pushing nodes on the stack

Consider the graph displayed in Figure 9 Let TMW() return **true** if and only if $|C_\ell| > 3$ and let $X = [2,3]$ be the current clique. Then $\ell = 2$ and $C_3 = \{4,5,6,7\}$. The the new clique is then $X = [2,3,4]$ and the nodes

$\ell = 3$ $X = [2,3,5]$ $C = \{4,5,6,7\}$

$\ell = 3$ $X = [2,3,6]$ $C = \{4,5,6,7\}$

$\ell = 3$ $X = [2,3,7]$ $C = \{4,5,6,7\}$

are pushed on top of the stack.

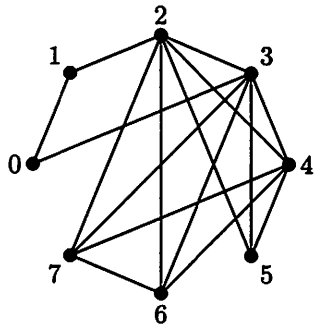

Figure 9: A sample graph

3 Implementation Details

3.1 Data Structures and Variable Declarations

The main data structure used in the program is a set, represented by a bit vector. Element i is in the set if and only if the ith bit is 1. The graph, then, is stored as an array of sets, $[A_j : j = 0, 1, \ldots, n-1]$. The jth set in the array represents the vertices in G adjacent to vertex j. Current cliques and the current optimal clique are also stored as sets. Using bit vectors to represent the sets results in a smaller amount of memory needed to store the data than if data structures liked linked lists or arrays of integers had been used. It also facilitates basic set operations; intersections and unions of sets are quick and simple to compute. Because UPC sometimes has difficulty dealing with dynamically allocated variables everything is declared statically before compilation. This means an upper bound on the number of vertices in the graph needs to be known beforehand.

Because the data structure used to store the graph information is relatively small, each thread makes a copy of the graph in its own private memory. The program references the graph frequently. Thus, placing a copy in memory that a thread has affinity to will speed up program execution.

Other than the stealstacks the only data placed in shared memory are the current optimal clique, `OptClique`, its size, `OptSize`, and the lock that regulates their access. Because all threads access these variables there is no benefit to giving them affinity to any particular thread, and they are declared simply as shared.

Distributing the initial cliques of length one, represented by the children of the root, modulus the thread number, divides more evenly the heavier branches among the threads than simply partitioning the branches. This way each of the threads will hopefully begin with one of the heavier branches from the left side of the tree.

3.2 Function Contents

The function TMW() determines whether or not the thread will push some of its jobs onto its stealstack. Because TMW() is called with each recursive call of MAXCLIQUE() it must run quickly. We chose to use the size of the extension set as the criteria for determining the amount of work the current clique will produce. A large extension set has the potential for creating much more work than a small set. Because the sets are all stored as bit vectors a simple mask and lookup algorithm for finding the size of the sets is not overly time-consuming.

The function FULL() determines if nodes will be released from the local to the shared portion of a thread's stealstack. As with the function TMW() the function FULL() is called with every recursive call of MAXCLIQUE() and must therefore run quickly. We chose to use the number of nodes in the local portion to determine if nodes should be released. If LOCALDEPTH(), a stealstack function, returns a value above a set limit, the thread will release the requisite number of nodes to its shared portion.

The number of nodes to release to the shared stack, acquire from the shared stack, and steal from another thread are all open to optimization. Stealing several nodes at a time cuts down on overhead. Without care, however, a problem may arise. The number of nodes released at a time, acquired at a time, and stolen at a time must be coordinated. If they are not it may happen that a thread has a number of nodes in the shared portion of its stealstack, but there are too few to steal and too few to acquire. In that case the nodes would go unprocessed and the program might return a clique that is not maximum. One way to ensure that no nodes are left in the shared stack is to make the number of nodes stolen at a time equal to the number of nodes released and the number acquired, say k. Then the number of nodes in the shared stack at any time would be a multiple of k and there would never be too few to be stolen or acquired. Another option is to acquire only one node at a time. With this option the number released and the number stolen at a time do not need to be equal. This allows the thread to release a number of nodes proportional to the number in its local stack.

3.3 Experimental Results

The motivation for a parallel implementation of the maximum clique finding algorithm was an increase in the speed of execution. To determine the validity of this presumption we compared the time of execution of the serial program to that of the parallel program. We first generated random graphs

Table 1: Density .20: Number of vertices vs number of threads: Time in seconds and total number of nodes processed

	Number of threads				
n	1	2	10	25	40
250	1.392 4,306	0.840 4,320	0.271 4,292	0.272 4,138	0.308 4,144
500	15.851 21,046	5.006 21,301	1.291 21,808	0.824 22,996	0.724 22,225
1000	368.160 302,596	76.359 302,726	16.279 303,252	7.259 302,810	5.082 302,956
2000		1654.030 5,066,860	337.225 5,084,782	136.812 5,062,776	88.303 5,062,985

on 250, 500, 1000, and 2000 vertices with densities of .20, .35, and .50. A density of k means that each possible edge is in the graph with probability k. This resulted in 12 random graphs. We first ran the serial program on each of these graphs. To obtain comparable results the serial program, originally a C program, was compiled as a UPC program, then run on one thread. Had the serial program been compiled as a C program, with a C compiler, the execution times may have been slightly faster. We then ran the parallel program, using the greedy bound, on the graphs, using varying numbers of threads: 2, 10, 25, and 40 threads.

The results are given in Tables 1, 2, and 3. The first number in each entry is the time in seconds for execution of the program. The second number is the total number of nodes processed during execution. The columns are labeled with the number of threads, with one thread representing the serial program. The rows are labeled with the number of vertices, n, in the graph.

The table in Figure 1, for example, gives the results for all trial graphs of density .20. Looking at any of the rows it is easy to see a dramatic increase in speed of execution as the number of threads increases. One might expect a speed increase directly proportional to the number of threads used. However, the tables show this to be false. Increasing the number of threads from one to two does not cut the time in half, nor does increasing from one to ten cut the time by a factor of 10. Managing the stealstacks, waiting at barriers and for locks, for example, all produce overhead not present in the serial version. This overhead is one of the factors that prevent the expected increase in speed.

Table 2: Density .35: Number of vertices vs number of threads: Time in seconds and total number of nodes processed

Number of threads

n	1	2	10	25	40
	7.593	5.015	1.184	0.650	0.590
250	25,670	25,939	27,309	27,819	29,329
	209.802	83.019	17.397	7.754	5.419
500	345,207	347,616	356,780	383,842	410,185
	13,368.565	3,505.992	699.551	277.631	178.432
1000	12,449,987	12,559,193	12,520,920	12,309,827	12,627,817
				13,316.92	8,488.096
2000				402823073	411,591,405

Table 3: Density .50: Number of vertices vs number of threads: Time in seconds and total number of nodes processed

Number of threads

n	1	2	10	25	40
	87.435	61.691	12.654	5.435	3.702
250	292,600	288,577	290,839	291,177	293,026
		3,488.164	688.334	276.494	173.756
500		12,924,435	12,731,349	12,751,478	12,771,695
					18,313.697
1000					1,008,807,821
2000					

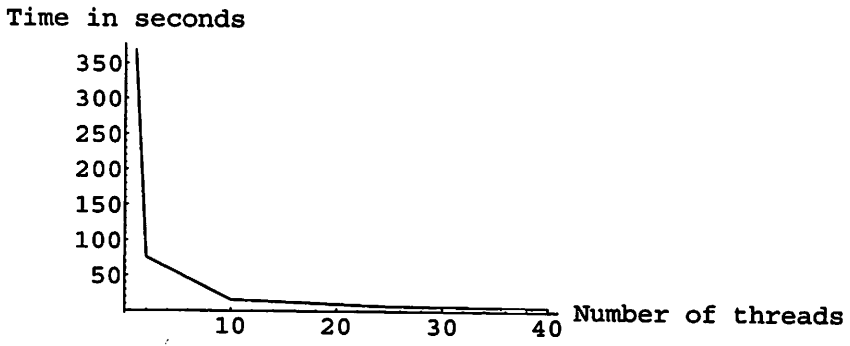

Figure 10: Density .20, 1000 vertices: Time in seconds vs Number of threads

To better see the speed increase brought about by a greater number of threads, consider the graphs in Figures 10, 11, and 12. Each graph shows the completion time, in seconds, for one of the randomly generated graphs, depending on the number of threads utilized. Note that although all three graphs have the same shape, the scales are quite different.

To illustrate that sharing was having a significant effect on the load balancing and efficiency of the program several of the random graphs were processed again, without allowing any stealing. The overhead of the steal-stacks was still present, but each thread worked only on the nodes in its own stack. The results of two of these trial runs, both on 500 vertices with density .35, are given in Tables 4 and 5. Each thread reported the number of nodes it processed, and from that the standard deviation was calculated. Looking at either table it is obvious that allowing threads to steal work from each other dramatically effects the load balance.

Also notable in Tables 1, 2, and 3 is the effect of the number of threads on the total number of nodes searched. It might be expected that the number of nodes searched would decrease, when it actually either remains about the same or even increases.

Several entries in Tables 1, 2, and 3, especially in Table 3 are empty. The time to completion for those graphs with the particular number of threads was too long.

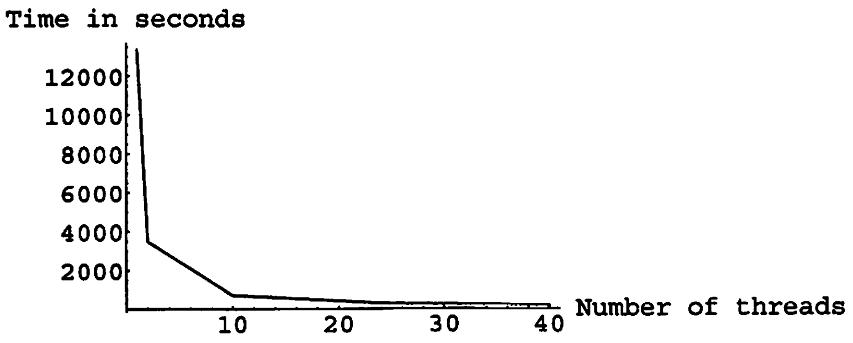

Figure 11: Density .35, 1000 vertices: Time in seconds vs Number of threads

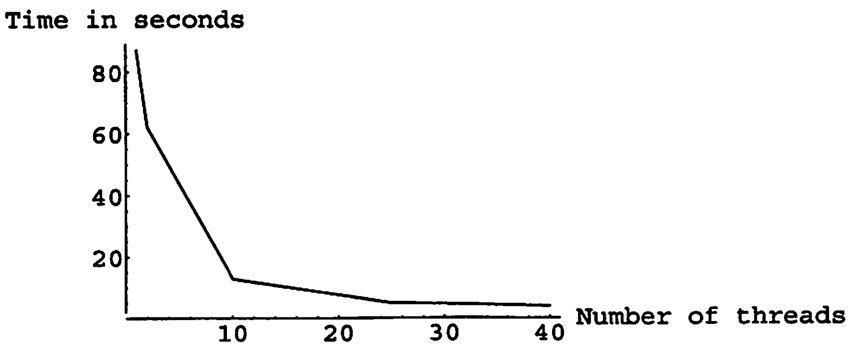

Figure 12: Density .50, 250 vertices: Time in seconds vs Number of threads

Table 4: 500 vertices, density .35, 10 threads

	total num nodes	std dev	time
sharing	356780	247	17 sec
no sharing	335061	1,843	78 sec

Table 5: 500 vertices, density .35, 25 threads

	total num nodes	std dev	time
sharing	383842	188	8 sec
no sharing	327145	1,604	37 sec

3.4 Conclusions

The ideas used to parallelize the maximum clique algorithm can be applied to other algorithms. Any backtracking algorithm can use stealstacks to balance the workload between the threads.

The implementation discussed in this paper utilizes the parallelism for moving through the state space tree. Any work done at a node on the tree is done by a single processor, and is therefore serial. One of the interesting side effects of this method is that, when compared to the serial version, the parallel program visits more nodes in the state space tree. There is a simple reason for what may at first seem counterintuitive. As the serial program traverses the state space tree it increases the size of the optimal clique. As it moves from left to right in the tree it prunes more branches because it has a larger optimal clique to prune against. In the parallel version several processors begin moving down the tree at the same time, and so for a while very little pruning happens because all processors are examining cliques of approximately the same size. The branches on the right side of the state space tree are not being compared to the ultimate largest clique on the left side, like with the serial program, but rather with a clique at about the same level. With less pruning happening more nodes are processed, meaning that the parallel version is actually doing more work. It is only because the work is being divided between several processors that the parallel program is able to complete sooner than the serial.

References

[1] D.L. Kreher and D.R. Stinson, *Combinatorial algorithms: generation, enumeration and search*, CRC Press, Boca Raton, 1990.

[2] J.Prins, Jun Huan, B. Bugh, Chau-Wen Tseng and P. Sadayappan, UPC *Implementation of an Unbalanced Tree Search Benchmark*, `http://www.cs.unc.edu/~huan/papers/03-034.pdf`, Univ. of North Carolina at Chapel Hill and Univ. of Maryland at College Park and Ohio State University, 2003.

[3] M. Flynn, Some Computer Organizations and Their Effectiveness, ITC, **C-21** (1972) 94.

[4] A. Grama, A. Gupta, G. Karypis and V. Kumar, *Introduction to Parallel Computing*, Pearson Education Limited, Harlow, England, 2003.

[5] S. Chauvin, P. Saha, F. Cantonnet, S. Annareddy and T. El-Ghazawi, *UPC Manual*, George Washington University, `http://www2.gwu.edu/~upc/documentation.html`, 2003.