A Parallel Optimization Algorithm for the Maximum Clique Problem

by

Mohamad A. Rizk

B.S., Computer Science, Lebanese American University, 2005

Thesis submitted in partial fulfillment of the requirements for the Degree of Master of Science in Computer Science

Division of Computer Science and Mathematics

LEBANESE AMERICAN UNIVERSITY

June, 2008

Lebanese American University
School of Arts and Sciences

Thesis Approval Form

Student Name: Mohamad Rizk_____ I.D. #: 200105318

Thesis Title :

A Parallel Optimization Algorithm for the Maximum Clique Problem

Program : M.S. in Computer Science

Division/Dept : Computer Science and Mathematics

School : Arts & Sciences - Beirut

Approved by :

[redacted]
Faisal Abu-Khzam, Ph.D. (Advisor)
Assistant Professor of Computer Science

[redacted]
Nashaat Mansour, Ph.D.
Professor of Computer Science

[redacted]
Rony Touma, Ph.D.
Assistant Professor of Mathematics

Date : June 19, 2008

Plagiarism Policy Compliance Statement

I certify that I have read and understood LAU's Plagiarism Policy. I understand that failure to comply with this Policy can lead to academic and disciplinary actions against me.

This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that by acknowledging its sources.

Name: Mohamad A. Rizk

Signature: 

Date: June 19, 2008

I grant to the LEBANESE AMERCIAN UNIVERSITY the right to use this work, irrespective of any copyright, for the University's own purpose without cost to the University or its students and employees. I further agree that the University may reproduce and provide single copies of the work to the public for the cost of reproduction.

"I can't understand why people are frightened by new ideas. I'm frightened of old ones."

John Cage

Acknowledgment

I would like to thank my advisor Dr. Faisal Abu-Khzam for his guidance throughout my Thesis work. A thanks is also to Dr. Nashaat Mansour and Dr. Rony Touma for being on my thesis committee.

I would like to express my sincere gratitude to the Lebanese American University whose financial support during my graduate studies made it all possible.

Finally, I would like to thank my friends and family for their long support.

Abstract

Recent advances in exact algorithm design and multi-processor industry have led to an increasing interest in exact (or optimal) solutions for hard problems. This interest was also motivated by the emergence of parametrized complexity theory as well as the the recent discouraging hardness of approximation results for most intractable problems. Coupling the best exact algorithms with scalable parallel implementations is a promising approach for dealing with computationally demanding problems. In this work, we introduce a parallel technique for solving the Maximum Clique problem using clusters of multi-core machines. Our algorithm employs a scalable load balancing strategy that is based on dynamic search-tree decomposition. We present experimental results that verify the scalability of our technique and its utility as a better alternative to approximation algorithms in many practical applications.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recent innovations in multiprocessor technology, encouraged algorithm designers to put more efforts in designing exact parallel algorithms to solve computationally demanding applications. This interest in exact parallel algorithms was influenced by the emergence of fixed-parameter tractability [13] and the tendency to avoid approximation algorithms. This tendency is justified by the the fact that some problems are hard to approximate with a guaranteed bounded error, unless some strongly believed complexity theories are dismissed or proved oppositely. Also, undesirable double inaccuracy or (what we call) two-fold approximation is another reason for discarding approximation algorithms. Two-fold approximation is the result of adopting simplifying assumptions to model a question by a certain problem and then adopting an approximate solution to that problem. Bringing together powerful platforms and the best exact algorithms seems to be a promising approach for bridging the gap between intractability and practicality, by solving practical instances of hard problems which are often judged as intractable. Such is the case of many graph problems like maximum clique, minimum vertex cover, maximum independent set, etc...

It is natural to choose the best exact algorithm and a highly scalable parallel implementation when seeking an accurate solution for a problem within reasonable time limits. An efficient scalable parallel implementation should guarantee fair load balancing among different processors, and the computation cost should not be overwhelmed by the cost of communication. In this work, we make use of clusters of multi-core

machines to solve the maximum clique problem by proposing a new parallel technique that realizes the above mentioned objectives.

Throughout this paper, we consider arbitrary, unweighted and simple graphs. A graph is considered simple if it is undirected and containing no self-loops or multiple edges. In other words, in a simple graph each vertex can not have an edge to itself or multiple edges to the same endpoint, and each edge does not have a direction. We denote a graph by $G = (V, E)$ where V is the set of all vertices and E is the set of edges. Two vertices are adjacent or neighbors if they have an edge between them, and the degree of a vertex indicates the number of edges incident to it (cardinality/size of its neighbors set). The maximum vertex degree in G will be the graph degree denoted by $\Delta(G)$. Graph $\bar{G} = (V, \bar{E})$ which is the complement of G is obtained by deleting the edges between the neighbors in G, while adding edges between non neighbors. $G' = (V', E')$ is a subgraph of G if $V'(G')$ and $E'(G')$ are subsets of $V(G)$ and $E(G)$ respectively. G' is an induced subgraph by $V'(G')$ in G if every pair of vertices in $V'(G')$ exhibit the same relationship as in G.

A graph is regular if all the vertices in it have the same degree. A complete graph is a regular graph of size n and degree $n - 1$. If an induced subgraph K in G is complete, then K is a clique of size $|V(K)|$. The neighbors or candidates set of a clique is the set of vertices that are adjacent to all vertices in the clique and therefore can extend the clique to a larger one. A clique K is considered maximal if it can not be contained as a subgraph in any larger clique. The maximum clique (henceforth MC) problem requires finding a maximal clique of maximum size in the given graph and is differentiated from the maximal clique enumeration (MCE) problem, which asks for generating all the maximal cliques in the graph.

Graph coloring is the process of assigning colors to the vertices of a graph such that no two adjacent vertices have the same color. The chromatic number of a graph is the minimum number of colors that can be used to color a graph. A color class is defined as the set of vertices that have the same color. It is obvious that no two vertices from the same color class can belong to the same clique. Therefore, the chromatic number or any number of color classes in a graph is an upper bound on the MC size in that graph.

An independent set I in a graph G is a set of vertices that induces an edgeless

subgraph in G. The maximum independent set problem seeks an independent set of maximum size in the given graph and is equivalent to the MC problem. This being the case because a clique in a graph G is an independent set in the graph $\bar{G}$.

A vertex cover in a graph G is a subset S of vertices such that each edge in G has at least one of its endpoints in S. Searching for a minimum vertex cover in $\bar{G}$ and hence solving the minimum vertex cover problem in $\bar{G}$ is also equivalent to solving the MC problem in G. The reason is that the vertices that do not belong to a vertex cover in $\bar{G}$ constitute an independent set in $\bar{G}$ and therefore a clique in G.

1.1 Applications

The importance of the maximum clique problem stems from its significant applications in a large number of diverse scientific fields such as Coding Theory [12], Image Processing [19], Bioinformatics [2, 3, 4], VLSI Design [22], Telecommunication [7], Fault Diagnosis [8], and Economics [9].

1.1.1 Protein Structural Alignment

Detecting similarities in proteins structures is of great importance in the field of Bioinformatics, since it is believed that proteins of similar structure have the same function and therefore can be classified in the same family. Some of the existing algorithms model the problem of matching d-dimensional proteins structures as the maximum common subgraph (MCS) problem which is one of the NP-complete problems. It has been found that the MCS problem can be solved by finding MC in an association graph that encodes the possible mappings between the two graphs being matched. A template/base algorithm for protein structural alignment works as follows: model the two input proteins as graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ where the vertices may represent the secondary structures (α-helix or β-strand) and the edges may denote connectivity between them. Then build the association graph $G_3 = (V_3, E_3)$ where $V_3 = V_1 \times V_2$ and any two vertices (u, v) and (u', v') $(u, u' \in V_1$ and $v, v' \in V_2)$ are connected either if $(u, u') \in E_1$ and $(v, v') \in E_2$ or $(u, u') \notin E_1$ and $(v, v') \notin E_2$.

The final step is to find MC in G_3 which will determine the maximum common subgraph between the two graphs and eventually the putative common structure between the proteins.

1.1.2 DNA Classification

One of the powerful methods used to classify DNA clones is Oligonucleotide fingerprinting. Given an array of DNA clones to be classified and a set of small DNA sequences (*i.e. oligonucleotide* of 8-50 bases) called probes, a probe is said to hybridize a clone if it occurs as a substring in it. After a series of hybridization experiments are carried on each clone with the set of probes, a fingerprint for each clone is created which is simply a sequence of hybridization intensity values for the clone with each probe. Those fingerprints are then transformed into normalized binary fingerprints using reference values from control DNA clones (a set of clones with known hybridization values with respect to the probes used in the experiments) where each hybridization value in the fingerprint is replaced either by a 1 (hybridization) or a 0 (no hybridization) or a N (unknown or missing value). Finally the binary fingerprints are clustered into the smallest possible number of clusters. In order to achieve the last step and to resolve the missing values, Figueroa *et al.* [15] greedily partition the graph into unique cliques in $O(p2^p n^2)$, where p is an upper-bound on the number of N's in any fingerprint and n is the number of fingerprints. They represent the fingerprints by graph nodes, where an edge is placed between two nodes/fingerprints if they do not differ at any position or at any position they differ, one of the values is N. After all maximal cliques are found and removed from the graph; it is searched for MC. Clearly, the maximal cliques together with the MC constitute the different clusters of the fingerprints.

1.1.3 Mining Market Data

Studying the behavior of the stock market by analyzing and visualizing the financial data is one of the essential problems in modern finance. However, the analysis becomes more complicated with the enormous data generated each day from the market. One

way to solve this problem is to model the financial data as a graph (called *market graph*) where the vertices are company stocks and two vertices are connected if they have a price correlation coefficient above a threshold θ (-1$\leq$ θ $\leq$1). Moreover, connected vertices will represent the stocks that have similar behavior over time. Clearly the larger θ is, the more the connected stocks behave similarly. Let $P_i(t)$ be the price of stock i on day t, $R_i(t) = \ln \frac{P_i(t)}{P_i(t-1)}$ be the logarithm of return for the stock i from day(t-1) to day t, and $< R_i >$ be the average of return of stock i over period N. Then the price correlation coefficient C_{ij} for two stocks i and j is calculated as:

$$C_{ij} = \frac{<R_i R_j> - <R_i><R_j>}{\sqrt{<R_i^2> - <R_i>^2><R_j^2> - <R_j>^2>}}$$

A maximum clique in the *market graph* denotes the largest number of stocks which are correlated together and the maximum independent set in it represents the diversified portfolios in the market. Boginski *et al.* [9] studied the behavior of stocks over the period 1998-2002 by solving the MC problem in the considered graph. They start by finding a lower bound l on the MC size using a greedy algorithm. Then, they reduce the graph size by removing the vertices whose degree is less than l, after that integer programming is used to find MC in the reduced graph. In order to find the maximum independent set, they find MC in the complement graph since it is more efficient. They also partition the studied stocks into different clusters using a greedy algorithm for partitioning the graph into unique cliques.

1.2 Literature Review

The MC problem is known to be NP-Hard in arbitrary graphs [17]. It was also shown that unless P=NP, there is no polynomial time algorithm that can approximate the MC problem within a factor of $n^{1-\epsilon}$ for any $\epsilon > 0$ [18].

One of the first algorithms for MC is due to Brone and Kerbosch (BK in what follows) [10]. BK solves MC by enumerating all cliques in the graph and then finds the one with maximum size among them. Recently, Tomita *et al.* [26] proved that BK runs in $O(3^{n/3})$ time. Another notable MC algorithm is due to Tarjan and Trojanowski [24], which solves the equivalent Maximum Independent Set Problem in $O(2^{n/3})$ time.

After the 1970's, the majority of MC algorithms are based on the branch-and-bound method. The key features in any MC branch-and-bound algorithm were identified as:

1. Lower-Bounding: How to find a lower bound on the MC size?
2. Upper-Bounding: How to find a tight upper bound on the MC size?
3. Divide-and-Conquer: How to break the problem into smaller subproblems?

Most of the algorithms, in the literature, select a vertex v and find the largest clique containing it. If not satisfied with any of the cliques that contains v, select another vertex and try to find a larger clique that does not include v. However, different algorithms follow different approaches regarding the selection of the vertices and answering the first two questions.

Balas and Yu [6] adopts a Lower-Bounding strategy by finding a maximum clique in a maximal triangulated[1] induced subgraph before starting the search. Balas and Xue [5] added an Upper-Bounding technique to the same algorithm by using fractional coloring. Woods [28] improved further the algorithm by coupling the same techniques for lower and upper bounding with selecting each time a vertex from the current largest color class to be added to a current clique.

Carraghan and Pardalos [11] did not implement any bounding methods. Their algorithm keeps track of the sum of the current clique and the number of the candidate vertices and it prunes the search when this sum drops below the current MC size. In addition to the pruning strategy, it also enhances the overall computational time by selecting the vertices in the ascending order of their degrees as advised by Fuji and Tomita [16]. Fahle [14] tested cost-based filtering techniques on Carraghan and Pardalos algorithm by adding two additional constraints.

Fahle [14] presented in his paper a taxonomy of upper bounds for the MC problem and succeeded in solving 12 DIMACS that were not solved before. During the same period, Ostergard [20] got also better results than Carraghan and Pardalos by ordering

[1] A simple graph is triangulated if every cycle of length greater than three has a chord in it. A chord is an edge between two vertices not adjacent in the cycle.

the vertices in the reverse order they used and coloring the vertices using greedy coloring. Recently, Tomita and Kameda [25] introduced a MC algorithm with remarkable improvements (see Chapter 2).

Pardalos *et al.* [21] implemented a parallel version of Garraghan and Pardalos algorithm using MPI (Message Passing Interface) and following the master-worker technique with a centralized load-balancing strategy. They tested their algorithm on random graphs of maximum size 500 using 2 and 4 processors. Shinano *et al.* [23] used PVM (Parallel Virtual Machine) and PUBB (Parallelization Utility for Branch-and-Bound algorithms) to implement a parallel MC algorithm. They succeeded in obtaining exact solutions for five unsolved DIMACS instances.

Chapter 2

The Tomita-Kameda Approach

Tomita and Kameda [25] introduced a MC branch-and-bound algorithm, with a great improvement over previously known methods. We refer to this algorithm by TK in what follows. The remarkable experimental results reported by the authors of the TK algorithm made it a natural choice for us to implement a parallel version of it. A search-tree based clique algorithm like TK can be viewed as a traversal of a virtual tree whose nodes are search states. Every search state consists of a set $CurrentK$ holding the current clique, a set $Clique\_Neighbors$ containing the neighbors of $CurrentK$, and the MC found so far, say $MaxK$.

In order to reduce the overall running time and search space, the TK algorithm employs a similar technique to Fuji and Tomita [16], and Carraghan and Pardalos [11]. Given a graph $G = (V, E)$, it starts by sorting the vertices in V in a decreasing order with respect to their degrees, such that the degrees of the unsorted neighbors of a vertex v are decremented after v is placed in the sorted list. During this process, if a draw occurs between two vertices or more in terms of their degrees, the one with minimum sum of neighbors' degrees is selected. However, if all remaining unsorted vertices have the same (minimum) degree, that is, the subgraph induced by those vertices is regular, the vertices of this regular subgraph G'' are kept unsorted. If the degree of G'' is $(|G''| - 1)$, then G'' constitutes a current $MaxK$ of size $|G''|$. The importance of this step is that it prunes the search tree in the branching/expansion phase later on by setting a lower bound on the size of any MC to be found.

In addition to the above pruning step, the TK algorithm prunes the search-tree further by finding an upper bound for any MC in a graph by taking benefit of the well known fact: the size of MC can not exceed the chromatic number of the graph. Moreover, it employs a greedy coloring at every state of its search. The assigned colors are represented by positive integers in such a way that no two adjacent vertices have the same color. The colored vertices are then sorted according to their colors such that any vertex v of color c must have at least $c-1$ adjacent vertices with $c-1$ different smaller colors. This sorting according to colors, helps in pruning/terminating the search at any point where $|MaxK|$ is greater than the sum of $|CurrentK|$ and the maximum assigned color. In the greedy-coloring algorithm described below, $Color$ will contain the colors of the vertices after coloring such that $Color[v]$ is equal to the color of vertex v.

Greedy-Coloring(S, $Color$)
Begin
for (each vertex v in S)
  $c = 1$
  while (any neighbor of v have color c)
    Increment c
  $Color[v] = c$
Sort vertices in S in ascending order according to their colors
End

The needed input for the TK branch function is the set of vertices sorted according to their colors in ascending order, and their colors. As a criteria for selecting a vertex for branching/expansion (adding it to $CurrentK$), the vertex with maximum color being the last vertex in the sorted list is considered. The selected vertex should also satisfy the condition that the sum of its color and $|CurrentK|$ is greater than $|MaxK|$. We refer to this vertex by $Candidate$ in the sequel.

Once $Candidate$ is determined, $Clique\_Neighbors$ (Initially is the whole set of vertices) is updated to become the intersection between itself and the neighbors of $Candidate$. If $Clique\_Neighbors$ is not empty after update, greedy coloring is applied to the new intersection set and the branching function is called again. However, if $Clique\_Neighbors$ is empty and the size of $CurrentK$ is greater than the size of

$MaxK$, the latter is then replaced by $CurrentK$, otherwise $CurrentK$ is ignored. As a last step, the current $Candidate$ is removed from CurrentK and another one is chosen from the sorted list. If no other candidate is found, the search backtracks. **Branch**, below, is a simple description of the TK branch function.

Branch($Clique\_Neighbors$, $Color$)
Begin
while ($Clique\_Neighbors$ is not empty)
  $Candidate$ = last vertex in $Clique\_Neighbors$
  Remove $Candidate$ from $Clique\_Neighbors$
  if ($Color[Candidate] + |CurrentK| > |MaxK|$)
    Add $Candidate$ to $CurrentK$
    $New\_Clique\_Neighbors = Clique\_Neighbors \cap$ neighbors of $Candidate$
    if ($New\_Clique\_Neighbors$ is not empty)
      Greedy-Coloring($New\_Clique\_Neighbors$, New_Color)
      Branch($New\_Clique\_Neighbors$, New_Color)
    else if ($|CurrentK| > |MaxK|$)
      $MaxK = CurrentK$
    Remove $Candidate$ from $CurrentK$
  else
    return
End

A requirement for the branch function is that the vertices should be sorted, in advance, in ascending order with respect to their colors. However, initially V is sorted only with respect to the degrees of vertices; therefore coloring should be applied on V. Yet, if greedy coloring is used, the order in V with respect to degrees is lost and the order according to colors is gained. In order to solve this complication, the TK algorithm applies an approximation coloring to the initially sorted set of vertices in V prior to branch.

It is clear that greedy coloring can be applied safely to the vertices of the regular subgraph (if any was found), since those vertices are kept unsorted as explained previously. Therefore, the vertices of the regular subgraph, say $V[0]$ to $V[i]$ ($0 \leq i \leq |G| - 1$) are colored and sorted using the greedy coloring method. As for the vertices

$V[i+1]$ to $V[|G|-1]$, they are colored using an approximation coloring according to the following criteria: let $RegC$ be the largest color used to color the vertices of the regular graph, then $V[i+1]$ receives color equal to the minimum of $(RegC+1, \Delta(G)+1)$, $V[i+2]$ receives color equal to the minimum of $(RegC+2, \Delta(G)+1)$,..., $V[i+k]$ receives color equal to the minimum of $(RegC+k, \Delta(G)+1)$. The theory behind the bounds $RegC+k$ and $\Delta(G)+1$ is that any MC to be found will have at minimum a size of $RegC+k$ where $k \geq 1$ and at maximum a size of $\Delta(G)+1$. It is obvious that after this step, the last vertex in the sorted list will have the minimum degree among other vertices and at the same time the highest color (equal to the chromatic number of G). The algorithm of initial sorting and coloring is listed below.

Initial-Sorting-Coloring()

Begin

$R = V$

$V = \phi$

$i = |G| - 1$

$RegC = 1$

while (R is not empty or does not induce a regular subgraph)

  v = vertex with minimum degree in R or vertex with minimum sum of neighbors' degrees among vertices with the same minimum degree in R

  $V[i] = v$

  Decrement i

  Decrement the degrees of v's neighbors

  Remove v from R

if (R induces a regular subgraph)

  Greedy-Coloring(R,$Color$)

  $RegC = Color[|R|-1]$ /*color of the last vertex in R*/

  for ($i = 0$ to $|R|-1$)

    $V[i] = R[i]$

  if (R is a complete subgraph)

    $MaxK = R$

for ($i = |R|$ to $|G|-1$)

  $Color[V[i]] = Min(RegC+i, \Delta(G)+1)$

End

Chapter 3

The Buffered Work-pool Approach

In this work, we couple messaging passing using MPI, threading and the methodology of dynamic search-tree decomposition, proposed by Abu-Khzam *et al.* [1], to present a new parallel load balancing technique. Our technique called the Buffered Work-pool approach (BWP in what follows) is an efficient scalable framework for implementing parallel versions of exact algorithms, specifically recursive backtracking ones such as the TK algorithm.

The BWP approach is a hybrid dynamic load-balancing technique that contains threading and message passing between the different processors of any cluster. It belongs to the category of master-worker techniques and similar to decentralized work-pool described by Wlinkson And Allen (see chapter 7 of [27]). The core concept of BWP is to permit workers to attain their own local shared work-pools (or task-buffers). Each worker have different local threads that exchange tasks from/to their process's work-pool and communicate tasks with other workers through the master process. Communication and synchronization overhead is reduced by assigning the role of grid middle-ware to the master process. Furthermore, the termination detection problem is solved by delegating the task of raising the termination flag to the master process.

Most of the exact algorithms for computationally demanding applications are search-tree based ones, where each computation is subdivided further into tasks and each task may generate a larger number of other tasks. Therefore, in BWP each task represents

a search-tree node which in turn corresponds to a certain state during computation. All state-related information should be encapsulated/encoded in the structure of its corresponding task. This dynamic nature of search-tree decomposition does not limit the objectives of BWP to best enhancement of execution time but also for excellent utilization of the distributed memory and computational power in use.

Figure 3.1 is a general overview of the BWP approach. We describe the possible scenarios in the following sections.

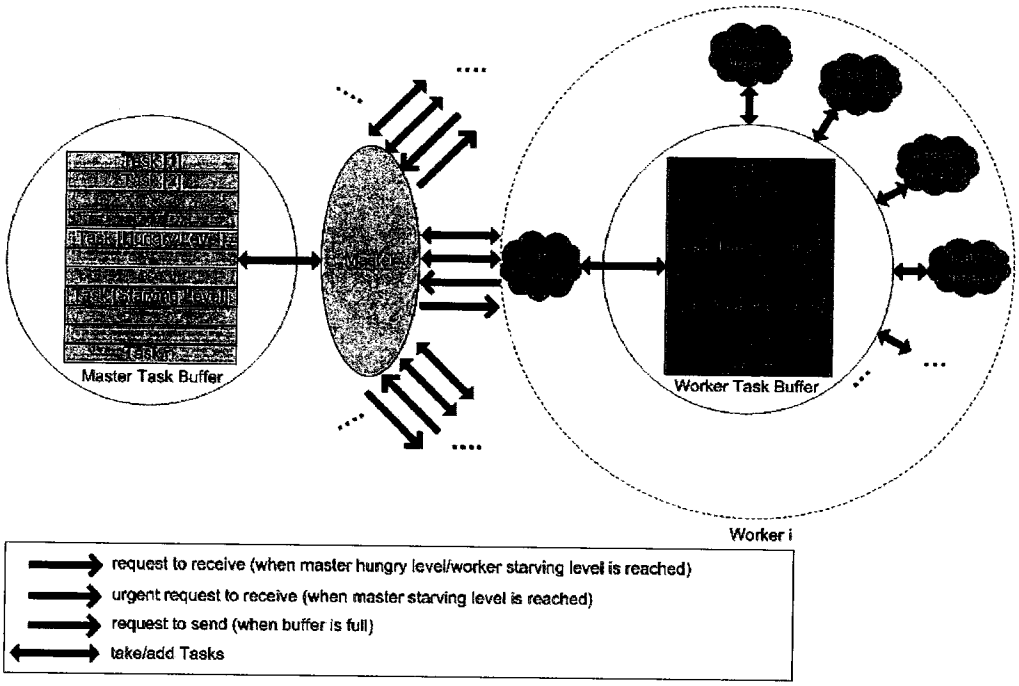

Figure 3.1: The General BWP Approach

3.1 The Master

The master process does not start immediately acting as a grid middle-ware. However, it starts the computation process by performing any problem specific pre-processing steps. Afterwards, it fills its task-buffer with an initial set of tasks and distributes, in a sequential manner, those tasks among different workers. Until then, the master's role

changes to communication with the involved workers. Throughout this communication role, the master process will be responsible for answering the different requests of the workers. It satisfies a worker requesting tasks by designating a number of tasks to be sent to this worker after settling an agreement with it. In addition, it realizes the wish of a worker to share part of its load with other workers by trying to accommodate part of its load. It will delegated this load to other workers upon request for tasks later on.

3.2 The Worker

After the initial set of tasks is received, each worker starts communicating with the master through one communication thread. Moreover, it fires a pre-defined number of threads whose role is to consume the tasks in its work-pool. During the consumption of tasks, each thread will have the ability to add new generated sub-tasks to its parent task-buffer. Yet, this ability is controlled by satisfying all of the following three conditions:

1. The number of new generated sub-tasks is greater than a user-defined parameter Add_Threshold.
2. The current computation level is a multiple of a predefined parameter called Level_Threshold.
3. The task-buffer is not full and not locked by another thread.

The first condition ensures that no thread wastes computation time by spending more time on adding tasks rather on expanding/branching tasks. The reason is that if the number of generated tasks is small, it will be more efficient to expand those tasks than to add them to the task-buffer. The second condition gives the user the ability to decide how coarse/fine (s)he wishes the computation to be. The third condition ensures that at most one thread can access the shared memory space at any time.

If any of the above conditions fails, the search thread proceeds as in the sequential version by expanding its tasks recursively.

3.3 Communication Scenarios

In BWP, the cost of exchanging messages is concealed by the computation cost since they occur simultaneously, where each worker has search threads to handle the computation part and one different thread to communicate with the master. The cost of communication is depreciated further by using asynchronous messages whenever possible.

Before any exchange of tasks occurs, the master and the involved worker should settle an agreement concerning the number of tasks to be exchanged. This agreement is established using synchronous communication. However, asynchronous communication is used whenever messages are not followed by an immediate exchange of tasks such as in the case of requests, negative replies, and announcements (example: announcing a new maximum clique size). We differentiate between the above mentioned types of messages by assigning a unique tag to each one according to its role. We should also note that announcements are usually problem-specific messages.

During the computation, the master tries to maintain a minimum number of tasks (which we refer to by Master_Starving_Level) in its work-pool in order not to reject any future requests for tasks. It achieves this will by sending a request for tasks, tagged as Urgent_Request message, to workers whenever the number of its tasks drops below Master_Starving_Level. When a worker receives an Urgent_Request message, it starts the negotiation with the master by sending an Urgent_Request_Answer message containing the number of tasks it is ready to send (a pre-defined percentage). In return, the master replies either positively by accepting to receive all/portion of the tasks designated by the worker, or negatively by rejecting to receive any tasks. The reply of the master depends on the current number of empty slots below Master_Starving_Level in its task-buffer. If a positive reply is received by the worker, the tasks in need are then sent.

As a precautious step to starving, the master process sends to workers a Normal_Request message indicating that it needs tasks, if the number of tasks in its task-buffer falls below its Master_Hungry_Level parameter. Each of the workers will not reply positively using tag Normal_Request_Answer unless it has tasks above the predefined value Worker_Starving_Level. The master process will reply to any positive

answer by sending the number of tasks it still needs to reach the Master_Hungry_Level. The exchange of tasks occurs if the last two communicated messages contained positive values. **Receive_Normal_Request_Answer** and **Send_Normal_Request_Answer**, below, are the algorithms for the above normal request scenario executed by the master and workers respectively. (ISend indicates non-blocking send in what follows)

Receive_Normal_Request_Answer()

Begin

worker_reply = Receive(Normal_Request_Answer, $worker$)

if(worker_reply > 0)

  if(number of tasks < Master_Hungry_Level)

    empty_slots = Master_Hungry_Level − number of tasks

    master_reply = Min(empty_slots, worker_reply)

    Send(Normal_Request_Answer, $worker$, master_reply)

    Receive_Tasks(Normal_Request_Answer, $worker$, master_reply)

  else

    ISend(Normal_Request_Answer, $worker$, 0)

End

Send_Normal_Request_Answer()

Begin

Receive_Message(Normal_Request, master)

if (number of tasks > Worker_Starving_Level)

  worker_reply = number of tasks − Worker_Starving_Level

  Send(Normal_Request_Answer, master, worker_reply)

  master_reply = Receive(Normal_Request_Answer, master)

  if (master_reply > 0)

    Send_Tasks(Normal_Request_Answer, master, master_reply)

else

  ISend_Message(Normal_Request_Answer, master, 0)

End

Categorizing requests for tasks sent by the master into urgent and normal, is another way of achieving fair distribution of tasks among workers. A loaded worker in terms of tasks will satisfy a large portion of the master's need upon receiving a normal

request. As a result, the master will not send an urgent request forcing workers to reply positively unless they have empty task-buffers. Moreover, the urgent request decreases the probability of a worker becoming idle during computation. This being the case because the master will maintain a minimum level of tasks whenever possible and thus it will feed starving workers with tasks upon request.

Two types of requests can be issued by any worker. A worker informs the master that it is ready to share some of its tasks specifically those above its Worker_Hungry_Level by sending a Delegation_Request message when its buffer is full. The master replies back by a Delegation_Request_Answer message containing the number of empty slots in its buffer. The flow of tasks will not take place, unless the worker still has a full buffer after receiving a positive answer from the master. The delegation scenario can be described by the following algorithms. (Check_Deadlock() is explained in section 3.5)

Send_Delegation_Request_Answer()

Begin

Receive(Delegation_Request, $worker$)

if(task-buffer is not full)

  master_reply = task-buffer size − number of tasks /*number of empty slots*/

  Send(Delegation_Request_Answer, $worker$, master_reply)

  Check_Deadlock(Delegation_Request_Answer,$worker$)

  worker_reply = Receive(Delegation_Request_Answer, $worker$)

  if (worker_reply > 0)

    Receive_Tasks(Normal_Request_Answer, $worker$, worker_reply)

else

  ISend(Delegation_Request_Answer, $worker$, 0)

End

Receive_Delegation_Request_Answer()

Begin

master_reply = Receive_Message(Delegation_Request, master)

if (task-buffer is full and master_reply > 0)

  worker_reply = task-buffer size − Worker_Hungry_Level

  worker_reply = Min(master_reply, worker_reply)

  Send(Delegation_Request_Answer, master, worker_reply)

  Send_Tasks(Delegation_Request_Answer, master, worker_reply)

else if(master_reply > 0)

  ISend(Delegation_Request_Answer, master, 0)

End

A Tasks_Request message is sent by a worker to indicate that the number of tasks in its buffer dropped below Worker_Starving_Level and thus starving. The master sends back the number of tasks it can delegate to this worker using Tasks_Request_Answer tag. Note that, the master reserves equal portions of its tasks for the different workers. Therefore, the master replies by sending the size of the portion for this starving worker. If the master's reply is positive, the worker responds by sending the number of tasks it needs to reach Worker_Starving_Level. The exchange of tasks occurs if the communicated values were positive as shown in the below algorithms.

Send_Tasks_Request_Answer()

Begin

Receive(Tasks_Request, $worker$)

if(number of tasks > 0)

  master_reply = number of tasks/number of workers

  Send(Tasks_Request_Answer, $worker$, master_reply)

  Check_Deadlock(Tasks_Request_Answer,$worker$)

  worker_reply = Receive(Tasks_Request_Answer, $worker$)

  if (worker_reply > 0)

    Send_Tasks(Tasks_Request_Answer, $worker$, worker_reply)

else

  ISend(Tasks_Request_Answer,$worker$, 0)

End

Receive_Tasks_Request_Answer()

Begin

master_reply = Receive_Message(Tasks_Request, master)

if (number of tasks < Worker_Starving_Level and master_reply > 0)

  worker_reply = Worker_Starving_Level − number of tasks

  worker_reply = Min(master_reply, worker_reply)

  Send(Tasks_Request_Answer, master, worker_reply)

  Receive_Tasks(Tasks_Request_Answer, master, worker_reply)

else if(master_reply > 0)

  ISend(Tasks_Request_Answer, master, 0)

End

To decrease the number of messages exchanged and to avoid a bottleneck state that may be caused by the master process, we put the following restrictions:

1. The master and worker should always send the master number of tasks they own in their requests. This helps in estimating the ability of the worker/master to delegate tasks.

2. A master/worker should not send a request, unless the other party has replied to previous requests and is expected to have tasks (except for Delegation_Request case).

The communication algorithms are shown below. (Note that New_Message() is true if the messages buffer has a new message.)

Master-Communication()

Begin

```
While (Terminate==FALSE)
  if(number of tasks ≤ Master_Hungry_Level)
    for (each worker)
      if(worker has tasks and has answered previous requests)
        ISend(Normal_Request, worker, number of tasks);
  if(number of tasks ≤ Master_Starving_Level)
    for (each worker)
      if(worker has tasks and has answered previous requests)
        ISend(Urgent_Request, worker, number of tasks);
  if(New_Message())
    if(new_message_tag == Normal_Request_Answer)
      Receive_Normal_Request_Answer()
    if(new_message_tag == Urgent_Request_Answer)
      Receive_Urgent_Request_Answer()
    if(new_message_tag == Tasks_Request)
      Send_Tasks_Request_Answer()
    if(new_message_tag == Delegation_Request)
      Send_Delegation_Request_Answer()
```

End

```
Worker-Communication()
Begin
While (Terminate==FALSE)
  if (number of tasks ≤ Worker_Starving_Level)
    if(master has tasks and has answered previous requests)
      ISend(Tasks_Request, master, number of tasks);
  if (task-buffer is full)
    if (master has answered previous requests)
      ISend(Delegation_Request, master, number of tasks);
  if(New_Message())
    if(new_message_tag==Normal_Request)
      Send_Normal_Request_Answer()
    if(new_message_tag==Urgent_Request_Answer)
      Send_Urgent_Request_Answer()
    if(new_message_tag==Tasks_Request_Answer)
      Receive_Tasks_Request_Answer()
    if(new_message_tag==Delegation_Request_Answer)
      Receive_Delegation_Request_Answer()
    if(new_message_tag==Termination)
      Terminate = TRUE
End
```

3.4 Termination Detection

To raise a termination flag, the master process should insure that:

1. Its task-buffer and the workers' buffers are empty.
2. All search threads of the workers are idle.
3. No messages are in transmission.

If the task-buffer of a worker is empty and it receives a zero-valued Urgent_Request message (*i.e.* the master's buffer is empty), it applies any of the following:

- If none of its search threads is active, and it is not waiting for a Tasks_Request_Answer message, it replies back by a negative value for the number of its tasks. The negative value indicates that it is ready to terminate.
- If not all its search threads are idle, or it has an unanswered Task_Request message, the worker sends back a zero number of tasks.

In its turn, when the master receives a negative value in the Urgent_Request_Answer message from a worker, it tags this worker as idle. After all workers are tagged as idle and the master's task-buffer is still empty, the master process announces the termination by broadcasting a termination signal.

The urgent request and termination scenario is described in the below algorithms.

Receive_Urgent_Request_Answer()
Begin
worker_reply = Receive(Urgent_Request_Answer, $worker$)
if(worker_reply > 0)
  if(number of tasks < Master_Starving_Level)
    empty_slots = Master_Starving_Level − number of tasks
    master_reply = Min(empty_slots, worker_reply)
    Send(Urgent_Request_Answer, $worker$, master_reply)
    Receive_Tasks(Urgent_Request_Answer, $worker$, master_reply)
  else
    ISend(Urgent_Request_Answer, $worker$, 0)
else if (number of tasks == 0 and worker_reply < 0)
  if (all workers are tagged idle)
    Terminate=TRUE
    Broadcast(Termination)
End

```
Send_Urgent_Request_Answer()
Begin
tasks_of_master=Receive_Message(Urgent_Request, master)
if (number of tasks > 0)
  worker_reply = a pre-defined percentage of the number of tasks
  Send(Urgent_Request_Answer, master, worker_reply)
  master_reply = Receive(Urgent_Request_Answer, master)
  if (master_reply > 0)
    Send_Tasks(Normal_Request_Answer, master, master_reply)
else if (tasks_of_master == 0 and number of tasks == 0)
  if (received reply for Task_Request message and all threads are idle)
    ISend(Urgent_Request_Answer, master, −1)
  else
    ISend(Urgent_Request_Answer, master, 0)
else
ISend(Urgent_Request_Answer, master, 0)
End
```

3.5 Avoiding Deadlocks

A deadlock occurs whenever two processes are involved in a synchronous communication and each one is waiting to receive a message from the other process to continue. In BWP, synchronous communication is used to settle agreements between the master process and any of the workers and asynchronous messages is used for requests. Therefore, it might happen that the master and a worker X are trying to settle different kinds of agreements at the same time. The result is that each one of them will be waiting to receive a different kind of message from the other and therefore a deadlock occurs. To avoid this, we delegate the task of escaping deadlocks to the master process. After sending a blocking send message and before waiting to receive the corresponding receive message, the master process probes its message buffer looking for blocking messages from X. If the master receives a blocking message from X whose tag does not correspond to the previously sent blocking message. In this case, the master gives the priority to serve the worker's message and then goes back into the blocking receive

state. In particular, the master checks for deadlock occurrence after it replies positively (using blocking send message) to a Delegation_Request or Tasks_Request of a worker X since there is no possibility for a deadlock to occur in other places. This is justified by the fact that these are the only cases where the master tries to initiate an agreement with the worker X (by sending the first blocking send message in the agreement procedure). A simple algorithm for avoiding deadlocks can be the following.

Check_Deadlock(Required_Tag, $worker$)
Begin
while (TRUE)
  if (master received a Required_Tag message from $worker$)
    return
  if (master received a blocking message other than Required_Tag message from $worker$)
    serve $worker$ according to its message
    return
End

Chapter 4

A Buffered Work-Pool Algorithm for Maximum Clique

4.1 Data Structures

Every search tree node (*a.k.a.* search state) in the search tree of the MC problem has a different $CurrentK$, and a graph state (*i.e.* a different set of active/candidate vertices). Therefore in BWP, each task has its own $CurrentK$ and list of candidate vertices. The colors of the candidate vertices should not be specified in the task structure except for the current last vertex($Candidate$) in the list which is sorted according to colors. As we will elaborate later, the parallel branching function uses the color of $Candidate$ to check if this vertex should be added to task's $CurrentK$ or not. If yes, the other vertices are assigned new colors. Moreover, we encode the task structure as a linear array called $data$ of size $n + 5$, where n is the number of vertices in the graph. Furthermore, slots $data[n]$ to $data[n + 4]$ will be used in the following way:

- $data[n]$ contains vertex $Candidate$.
- $data[n + 1]$ specifies the color of $data[n]$
- $data[n + 2]$ holds the size of $CurrentK$
- $data[n + 3]$ indicates the level at which this task was generated/created

- $data[n+4]$ holds the size of $Clique\_Neighbors$

On the other hand, the first n slots of $data$ contain $Clique\_Neighbors$ and $CurrentK$. During branching, if a vertex is added to $CurrentK$, it will be contained at $data[n-1-|CurrentK|]$. In addition, each task is considered as valuable if the sum of its $data[n+2]$ and $data[n+1]$ is larger than current $|MaxK|$ since it may lead to a larger MC.

During sequential branching, vertices are expanded in a descending order with respect to their colors at each level. Imitating this behavior in the parallel branching has a great impact on the size of the search tree by affecting the overall number of tasks being generated and consumed later on (see section 4.4). However, if tasks were to be added to or removed from the task-buffer in a random manner, their expanding/branching order will be lost; unless a kind a search for the "right" task is applied when a task to be consumed.

As a first step to mimicking the sequential version, we implemented the task-buffer of every process in a format similar to a hash table. We used a linear array of pointers called $levels$ (representing search-tree levels) where each pointer points to another array of pointers called $colors$ such that $colors[c]$ points to a doubly-linked list of tasks whose $Candidate$ vertex have color c. The size of $levels$ is equal to the chromatic number of the graph computed in the initial coloring stage. This is justified by the fact that the height of the search tree can not exceed the largest possible size of MC which is the chromatic number of the graph, say $MaxC$. Guided by the rule that no two vertices from the same color class can belong to the same clique, we record the following observation. At search-tree level i, only one vertex from each of the highest i color classes would have been added to $CurrentK$, with no possibility to use those color classes again in the chosen search path. Therefore it is safe to set the size of $colors$ at level 0 to $MaxC$, at 1 to $MaxC-1$,..., at i to $MaxC-i$.

Figure 4.1 illustrates the task-buffer in the BWP version of TK.

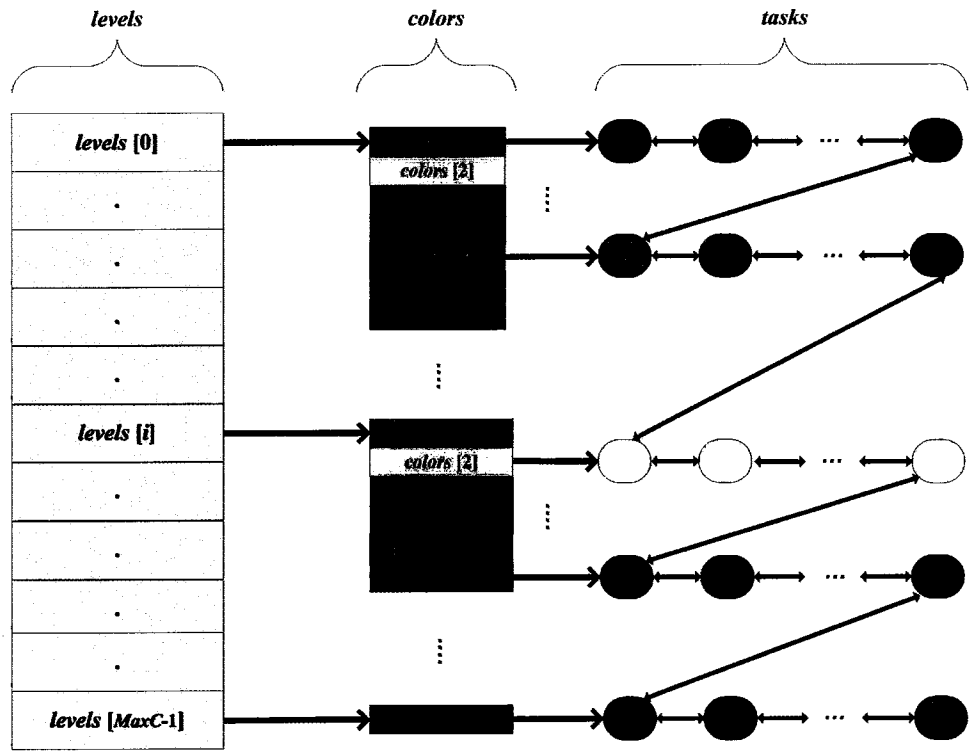

Figure 4.1: Task-buffer in the BWP version of TK

As shown in the above figure, tasks point to each other even if they are at different levels and belong to different color sets. This property reduces the time needed to add and remove tasks from the buffer.

4.2 The Master

The master process starts the computation by doing the pre-processing steps that include the initial sorting and coloring of vertices as described in the TK algorithm. After the pre-processing steps, the master process encodes/encapsulates each vertex v in the sorted list (except for the vertices of the regular subgraph R, if any was found during the pre-processing steps) as a task whose $data[n+2]$ and $data[n+3]$ are equal to zero, whereas its $data[n]$ and $data[n+1]$ contain v and its color respectively. Moreover,

every v in the sorted list represents a different search path which does not consider all the vertices whose index is larger than v's index. Therefore, the candidates list in each task is defined as the list of all vertices from index zero to index of v in the sorted list. Afterwards, the created tasks are inserted to the master's task-buffer. In order to minimize the search time, the master starts the search phase in a depth-first manner (starting by the task whose $Candidate$ has the highest color) and adds the tasks generated on each level to its buffer. Moreover, the master stops the search when the number of tasks generated, at search-tree levels greater than zero, is at least equal to the number of workers. After that, the master distributes equally the generated tasks among the processors. In the following, we explain the algorithm followed by the master process in case of MC.

Master()
Begin
Initial-Sorting-Coloring()
for ($i = |R|$ to $|G| - 1$)
  $task$ = New_Task()
  $task.data[n] = V[i]$
  $task.data[n+1] = Color[V[i]]$
  $task.data[n+2] = 0$
  $task.data[n+3] = 0$
  $task.data[n+4] = i$
  for ($j = 0$ to $i - 1$)
    $task.data[j] = V[j]$
  Insert $task$ to the task-buffer
while (number of tasks whose $data[n+3] > 0$ is less than number of workers)
  Branch and add the new tasks to task-buffer
Distribute the tasks whose $data[n+3] > 0$ equally among workers
Master-Communication()
End

4.3 Parallel Branching By Workers

After receiving the initial set of tasks, the worker sets off one communication thread and a pre-defined number of search threads. Each search thread takes one task from the buffer (if task-buffer is not empty) and starts recursively expanding it. Moreover, the search thread checks whether the sum of task's $Candidate$ color and the size of its $CurrentK$ (*i.e.* $data[n+1] + data[n+2]$) is larger than the current MC size. If so, $Candidate$ is added to task's $CurrentK$ whose size is thus incremented by one. Then, a new $Clique\_Neighbors$ set is created by determining the intersection between the neighbors of $Candidate$ and current $Clique\_Neighbors$ set.

If the new $Clique\_Neighbors$ set is empty and $|CurrentK|$ is less than $|MaxK|$, then the branch on the considered task is terminated and a new task is chosen by the search thread. However if $|CurrentK|$ is greater than $|MaxK|$, then $|CurrentK|$ is sent to the master process to replace $|MaxK|$. In its turn, the master broadcasts the received $|CurrentK|$ as the new $|MaxK|$[1]. On the other hand, if the intersection set is not empty and has a size s, greedy coloring is applied on it as in the sequential version to get a new ordering by color. The search thread follows a similar procedure to the master process by branching on a task in a depth-first manner and adding the generated tasks in a breadth-first manner. The search thread produces a new list of tasks of the colored and sorted intersection set according to the following: let v be a vertex of index $i(\leq i \leq s-1)$ and has color c in the sorted intersection list. The created task will have v, c, $|CurrentK|$ and current search-tree level contained in $data[n]$, $data[n+1]$, $data[n+2]$, and $data[n+3]$ respectively. The value for $data[n+4]$ will be i since the vertices between index 0 and $i-1$ will constitute the candidates list for the new task. If all the conditions in section 3.2 are satisfied, the produced tasks are added to the task-buffer in ascending order of their colors except for the last task (task of highest $Candidate$'s color) in the list which will be expanded recursively. On the other hand, if not all the conditions are satisfied or the task-buffer accommodated only a portion of the produced tasks, the current thread branch recursively the remaining tasks in descending order of their $Candidate$ color.

The BWP version of **Branch**() is shown below.

[1]The master process assures again that the received $|CurrentK|$ is larger than $|MaxK|$ before sending a broadcast message; if not, no action is taken.

Worker-Parallel-Branching($task$)

Begin

if ($task.data[n+1] + task.data[n+2] > |MaxK|$)

$Candidate = task.data[n]$

computation-level = $task.data[n+3]$

for ($i = 0$ to $data[n+4]$)

$Clique\_Neighbors[i] = task.data[i]$

for ($i = 0$ to $data[n+2]$)

$CurrentK[i] = task.data[n-1-i]$

Add $Candidate$ to $CurrentK$

$New\_Clique\_Neighbors = Clique\_Neighbors[i] \cap$ neighbors of $Candidate$

if ($New\_Clique\_Neighbors$ is not empty)

Greedy-Coloring($New\_Clique\_Neighbors$, $Color$)

$s = |New\_Clique\_Neighbors|$

new_tasks = New_Tasks(s) /*Allocate array of tasks and of size s*/

for ($i = s-1$ to 0)

$v = New\_Clique\_Neighbors[i]$

if ($Color[v] + |CurrentK| > |MaxK|$)

$new\_tasks[i].data[n] = v$

$new\_tasks[i].data[n+1] = Color[v]$

$new\_tasks[i].data[n+2] = |CurrentK|$

$new\_tasks[i].data[n+3]$ = computation-level + 1

$new\_tasks[i].data[n+4] = i$

for ($j = i-1$ to 0)

$new\_tasks[i].data[j] = New\_Clique\_Neighbors[j]$

for ($j = 0$ to $|CurrentK|$)

$new\_tasks[i].data[n-1-j] = CurrentK[j]$

else

break

If (task-buffer is not full and not locked and $s >$ Add_Threshold and

computation-level%Level_Threshold == 0)

if possible add the first $s-1$ tasks in new_tasks to tasks-buffer

while (new_tasks is not empty)

$task$ = last task in new_tasks

Worker-Parallel-Branching($task$)

Remove $task$ from new_tasks

else /*$New\_Clique\_Neighbors$ is empty*/

if ($|CurrentK| > |MaxK|$)

$MaxK = CurrentK$

ISend(New_MC, master,$MaxK$)

End

4.4 Search-Tree Pruning

In addition to the fact, that the parallel branch function will ignore any task whose $data[n+1]+data[n+2] \leq |MaxK|$ during addition of tasks and recursive branching. We further reduce the number of tasks being handled by deleting tasks that do not satisfy this condition during the exchange of tasks between the master and any of the workers. Deleting tasks during communication is possible since the the master and a worker X might have different values for $|MaxK|$ (synchronization of value did not occur yet). Furthermore, when a process receives a new value for $|MaxK|$, it deletes all tasks in its buffer which may be become worthless after the update of $|MaxK|$.

The criteria for selecting tasks for exchange and branching by the communication and search threads plays a major role in reducing the search space. There are mainly two alternatives when removing a task from the task buffer: either remove a task from the top of the search-tree or from its bottom, *i.e.* either in an ascending order of $data[n+3]$ or vice versa. In general, the choice is problem-specific since when the sequential version is brute-force in nature, it will not make a difference in what order tasks are chosen because eventually all of them will be expanded.

However, in the BWP version of TK, it is clear that the larger $MaxK$ is, the less the tasks and search-space will be. Moreover, priority is given to the tasks at the the bottom of the search-tree with highest color since those represent the shortest path to any new $MaxK$. Therefore, the master and the communication/search threads of the workers are forced to select the task with the highest level and $Candidate$'s color ($data[n+1]$ and $data[n+3]$) from the task-buffer during exchange and branching of

tasks. If there is more than one task that share the same values for $data[n+1]$ and $data[n+3]$, the one with minimum candidates list ($data[n+4]$) is chosen.

Chapter 5

Experimental Results

In order to assess the efficiency of the BWP algorithm for MC, we implemented the sequential version of the TK algorithm and our parallel version. We have used graphs from the DIMACS Benchmark (http://dimacs.rutgers.edu/Challenges) to compare both versions; moreover we chose the graphs that the sequential TK is known to take near to or more than one hour to solve. Since, the BWP algorithm employs threads and communication between the master and the different workers, the search-tree built and the computation time for a given instance may vary over multiple runs. Therefore, in Table 5.1 we report the average, maximum and minimum run times for 25 runs on each instance. The parameters' values which were used during the experiments are listed in appendix A.

During experiments, we have used different sets of parameters' values on each instance until we had an average super-linear/linear speedup. We can not claim that the reported speedups are the best nor the worst that one can ever get on the considered instances. Another set of parameters' values may result in greater or smaller speedup. We should note also that the amount of work done by the parallel version is more than that of the sequential since multiple search-spaces are explored at the same time(*i.e.* more tasks are generated and branched). However, the speedup is achieved since more processors are used and the search-tree is conquered in parallel. In other words, in the BWP version, larger values for the maximum clique size are found faster than the sequential version; therefore, the search tree is pruned at earlier stages than the

Graph	Sequential TK	BWP version of TK (processors = 6)			Speedup		
(V,E,MC)	User Time (Secs)	Average UserTime (Secs)	Max User Time (Secs)	Min User Time (Secs)	Average	Max	Min
p_hat1000-2 (1000, 244799,46)	5886 (1.63 hours)	557 (9.28 min)	771 (12.8 min)	454 (7.56 min)			
p_hat500-3 (500, 93800,50)	5691 (1.58 hours)	619 (10.13 min)	726 (12.1 min)	411 (6.85 min)			
brock400_1 (400,59723,27)	3861 (1.07 hours)	620 (10.3 min)	914 (15.2 min)	494 (8.24 min)			4.22
brock400_3 (800,59681,31)	2632 (43.87 min)	179 (2.98 min)	492 (8.2 min)	99 (1.65 min)			
brock800_1 (800,207505,23)	34174 (9.5 hours)	5669 (1.57 hour)	7100 (1.97 hours)	5032 (1.39 hours)			4.81
brock800_2 (800,208166,24)	30696 (8.52 hours)	4980 (1.83 hours)	7844 (2.17 hours)	3710 (1.03 hours)			3.91
brock800_4 (800,207643,26)	14478 (4 hours)	2171 (36.1 min)	3548 (59.14 min)	1734 (28.9 min)			4.08
MANN_a45 (1035,533115,345)	5400 (1.5 hours)	825 (13.75 min)	938 (15.64 min)	753 (12.55 min)			

Super-linear Speedup Linear speedup

Average Speedup = Seq. Time/Avg. Time; Minimum Speedup = Seq. Time/Max. Time; Maximum Speedup = Seq. Time/Min. Time

Table 5.1: Sequential TK Vs BWP version of TK

sequential version and hence the computational time is less.

We have used six processors (one master and five workers) in our experiments. Therefore, we consider a speedup in the range five to six as linear and as super-linear if it is greater than six. The BWP efficiency is shown by the super-linear speedup achieved in most of the cases. Also, the reported runtimes proves our previous claim which is, in BWP the communication and synchronization times are concealed. Since if otherwise, the the reported speedups should have been less.

In order to study the benefit of the dynamic search-tree decomposition and branching on the search-tree in a depth-breadth first manner by permitting the threads to add tasks. We did experiments (shown in Table 5.2) on two graphs p_hat1000-2 and brock800-1. During those experiments, we reduced the exchange of tasks to the minimum by setting the values of Hungry_Level and Starving_Level parameters for the master and the workers to 2 and 1 respectively. In addition, we used one search thread in each worker without giving it the permission to add tasks to the buffer by setting the Add_Threshold value to a very large value (100000). The speedup obtained was near to linear in the case of p_hat1000-2 and below linear in the case of brock800-1.

Graph (V,E,MC)	Sequential TK User Time (Secs)	BWP version of TK (processors = 6) User Time (Secs)	Speedup
p_hat1000-2 (1000, 244799,46)	5886 (1.63 hours)	1300 (21.6 min)	4.60
brock800_1 (800,207505,23)	34174 (9.5 hours)	8330 (2.31 hours)	4.10

Speedup = Seq. Time/BWP Time

Table 5.2: BWP version of TK without dynamic search-tree decomposition

Even with increasing the number of the threads, the speedup achieved was not much better. In most cases, the extra threads will be doing worthless work that does not help in pruning the search tree at early stages. This is the case because each thread will be using only depth-first search on a task (from the initial set of tasks) until this task can not lead to any larger MC and without adding any generated tasks during search. Permitting threads to add tasks at different levels (and hence adding the property of breadth-first search) helps by increasing the chance of reaching a larger MC size and reducing the search-tree size in a faster manner. However, we should note that this permission should be controlled by taking into consideration the size and density of the graph instance. In addition, a balance should be found between the number of threads, the frequency of adding tasks, and the number of tasks to add.

Chapter 6

Concluding Remarks

Many real life questions are modeled by the maximum clique problem (MC). Approximation algorithms may not be always convenient due to the complex nature of the corresponding question. This is often the case in Bioinformatics and Economics applications. In this work, we proposed an efficient parallel exact algorithm for MC. Our parallel algorithm employs the Buffered Work-pool approach (BWP), which is a dynamic load balancing technique that targets clusters of shared-memory multiprocessors. BWP is suitable for highly demanding computations where each task assigned to a process has the potential of producing a large number of sub-tasks. Another attractive feature of BWP is the fact that it reduces the communication latency by permitting computation and communication to occur simultaneously.

The BWP method applies well to many other combinatorial problems. Other members of our research team are currently using it in designing algorithms for many classical problems like SAT, Vertex Cover and Maximal Cliques enumeration.

Bibliography

[1] F. Abu-Khzam, M. Langston, P. Shanbhag, and C. Symons. Scalable parallel algorithms for fpt problems. *Algorithmica*, 45(3):269–284, 2006.

[2] D. Bahadur, T. Akutsu, E. Tomita, and T. Seki. Protein side-chain packing problem: a maximum edge-weight clique algorithmic approach. In *APBC '04: Proceedings of the second conference on Asia-Pacific bioinformatics*, pages 191–200, Darlinghurst, Australia, 2004. Australian Computer Society, Inc.

[3] D. Bahadur, T. Akutsu, E. Tomita, T. Seki, and A. Fujiyama. Point matching under non-uniform distortions and protein side chain packing based on efficient maximum clique algorithms. *IPSJ SIG Notes*, 2002(36):21–24, 20020510.

[4] D. Bahadur, E. Tomita, J. Suzuki, K. Horimoto, and T. Akutsu. Protein threading with profiles and distance constraints using clique based algorithms. *J. Bioinformatics and Computational Biology*, 4(1):19–42, 2006.

[5] E. Balas and J. Xue. Weighted and unweighted maximum clique algorithms with upper bounds from fractional coloring. *Algorithmica*, 15(5):397–412, 1996.

[6] E. Balas and C. Yu. Finding a maximum clique in an arbitrary graph. *SIAM Journal on Computing*, 15(4):1054–1068, 1986.

[7] B. Balasundaram and S. Butenko. Graph domination, coloring and cliques in telecommunications. In M. Resende and P. Pardalos, editors, *Handbook of Optimization in Telecommunications*, pages 865–890. Spinger Science + Business Media, New York, 2006.

[8] P. Berman and A. Pelc. Distributed fault diagnosis for multiprocessor systems. In *Proceedings of the 20th Annual International Symposium on Fault-Tolerant Computing*, pages 340–346, 1990.

[9] V. Boginski, S. Butenko, and P. Pardalos. Mining market data: a network approach. *Computers and Operations Research*, 33(11):3171–3184, 2006.

[10] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph. *Communications of the ACM*, 16:575–577, 1973.

[11] R. Carraghan and P. Pardalos. An exact algorithm for the maximum clique problem. *Operations Research Letters*, 9:375–382, 1990.

[12] I. Chuang, A. Cross, G. Smith, J. Smolin, and B. Zeng. Codeword stabilized quantum codes: algorithm and structure. *ArXiv e-prints*, 803, 2008.

[13] R. Downey and M. Fellows. *Parameterized Complexity*. Springer-Verlag, 1999.

[14] T. Fahle. Simple and fast: Improving a branch-and-bound algorithm for maximum clique. In *ESA '02: Proceedings of the 10th Annual European Symposium on Algorithms*, pages 485–498, London, UK, 2002. Springer-Verlag.

[15] A. Figueroa, J. Borneman, and T. Jiang. Clustering binary fingerprint vectors with missing values for dna array data analysis. *J. of Computational Biology*, 11(5):887–901, 2004.

[16] T. Fuji and E. Tomita. On efficient algorithms for finding a maximum clique. Technical report, IECE.

[17] M. Garey and D. Johnson. *Computers and Intractability*. W. H. Freeman, New York, 1979.

[18] J. Hastad. Clique is hard to approximate within $n^{1-\epsilon}$. In *FOCS '96: Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, page 627, Washington, DC, USA, 1996. IEEE Computer Society.

[19] K. Hotta, E. Tomita, and H. Takahashi. A view-invariant human face detection method based on maximum cliques. *Transactions of Information Processing Society of Japan*, 44(SIG14(TOM9)):57–70, 2003.

[20] P. Ostergard. A fast algorithm for the maximum clique problem. *Discrete Appl. Math.*, 120(1-3):197–207, 2002.

[21] J. Rappe P. Pardalos and M. Resende. An exact parallel algorithm for the maximum clique problem. pages 279–300. Kluwer Academic Publishers, 1998.

[22] R. Pal, S. Pal, and A. Pal. An algorithm for finding a non-trivial lower bound for channel routing. *Integr. VLSI J.*, 25(1):71–84, 1998.

[23] Y. Shinano, T. Fujie, Y. Ikebe, and R. Hirabayashi. Solving the maximum clique problem using pubb. In *IPPS '98: Proceedings of the 12th. International Parallel Processing Symposium on International Parallel Processing Symposium*, page 326, Washington, DC, USA, 1998. IEEE Computer Society.

[24] R. Tarjan and A. Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6(3):537–546, 1977.

[25] E. Tomita and T. Kameda. An efficient branch-and-bound algorithm for finding a maximum clique with computational experiments. *Journal of Global Optimization*, 37:95–111, 2007.

[26] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques. In *10th Int. Computing and Combinatorics Conf. (COCOON)*, 2004.

[27] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[28] D. Wood. An algorithm for finding a maximum clique in a graph. *Operations Research Letters*, 21(5):211–217, 1997.

Appendix A

The parameters' values that were used during the experiments are listed below:

- **p_hat1000-2**

  {

  Worker task-buffer size=1000; Worker_Hungry_Level=800; Worker_Starving_Level=250

  Master task-buffer size=1000; Master_Hungry_Level=75; Master_Starving_Level=30

  Number of search threads=5; Add_Threshold=50; Level_Threshold=5

  }

- **p_hat500-3**

  {

  Worker task-buffer size=500; Worker_Hungry_Level=300; Worker_Starving_Level=200

  Master task-buffer size=500; Master_Hungry_Level=100; Master_Starving_Level=50

  Number of search threads=5; Add_Threshold=25; Level_Threshold=5

  }

- **brock400-1**

  {

  Worker task-buffer size=1000; Worker_Hungry_Level=800; Worker_Starving_Level=250

  Master task-buffer size=1000; Master_Hungry_Level=50; Master_Starving_Level=20

  Number of search threads=5; Add_Threshold=25; Level_Threshold=5

  }

- **brock800-1**

  {

  Worker task-buffer size=1600; Worker_Hungry_Level=600; Worker_Starving_Level=300

  Master task-buffer size=1600; Master_Hungry_Level=75; Master_Starving_Level=30

  Number of search threads=15; Add_Threshold=20; Level_Threshold=3

  }

- **brock800-2**

  {

  Worker task-buffer size=1600; Worker_Hungry_Level=600; Worker_Starving_Level=300

  Master task-buffer size=1600; Master_Hungry_Level=75; Master_Starving_Level=30

  Number of search threads=10; Add_Threshold=15; Level_Threshold=4

  }

- **brock800-4**

  {

  Worker task-buffer size=1600; Worker_Hungry_Level=600; Worker_Starving_Level=300

  Master task-buffer size=1600; Master_Hungry_Level=75; Master_Starving_Level=30

  Number of search threads=10; Add_Threshold=20; Level_Threshold=4

  }

- **MANN_a45**

  {

  Worker task-buffer size=1400; Worker_Hungry_Level=800; Worker_Starving_Level=250

  Master task-buffer size=1400; Master_Hungry_Level=75; Master_Starving_Level=30

  Number of search threads=15; Add_Threshold=50; Level_Threshold=5

  }