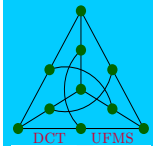


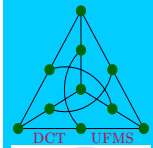
Efficient Implementation of the BSP/CGM Parallel Vertex Cover FPT Algorithm

E. J. Hanashiro DCT - Univ. Fed. Mato Grosso do Sul

H. Mongelli DCT - Univ. Fed. Mato Grosso do Sul

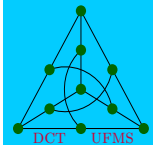
S. W. Song IME - Univ. São Paulo

Summary



Parameterized Complexity and Fixed Parameter Tractability

- Parameterized Complexity
- FPT - Fixed Parameter Tractability
- Techniques in FPT Algorithm Design

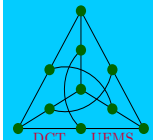





Back
End

Parameterized Complexity

- The input problem is divided into two parts:
  - the main part containing the data set
  - a parameter
- A problem whose input can be divided like this is said to be *parameterized*.

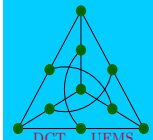





Back
End

FPT - Fixed Parameter Tractability

- A parameterized problem is said to be *fixed-parameter tractable* if there is an algorithm that solves the problem in $O(f(k)n^{\alpha})$ time, where α is a constant and f is an arbitrary function.
- The definition of FPT problems remains unchanged if we consider $O(f(k) + n^{\alpha})$ time.
- The main part of the input contributes polynomially on the total complexity of the problem.
- The parameter is responsible for the combinatorial explosion.
- This approach is feasible if the constant α is small and the parameter k is within a tight, but useful, interval.
- The fixed parameter tractable problems form a class of problems called *FPT*.



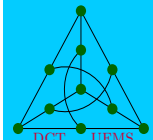




Back
End

Techniques in FPT Algorithm Design

- Two techniques are usually applied:
  - the reduction to problem kernel
    The goal is to reduce, in polynomial time, an instance I of the parameterizable problem into another equivalent instance I', whose size is limited by a function of the parameter k.
  - the bounded search tree
    This technique attempts to solve the problem through an exhaustive tree search, whose size is to be bounded by a function of the parameter k.

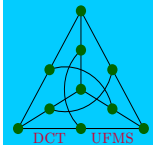






Back
End

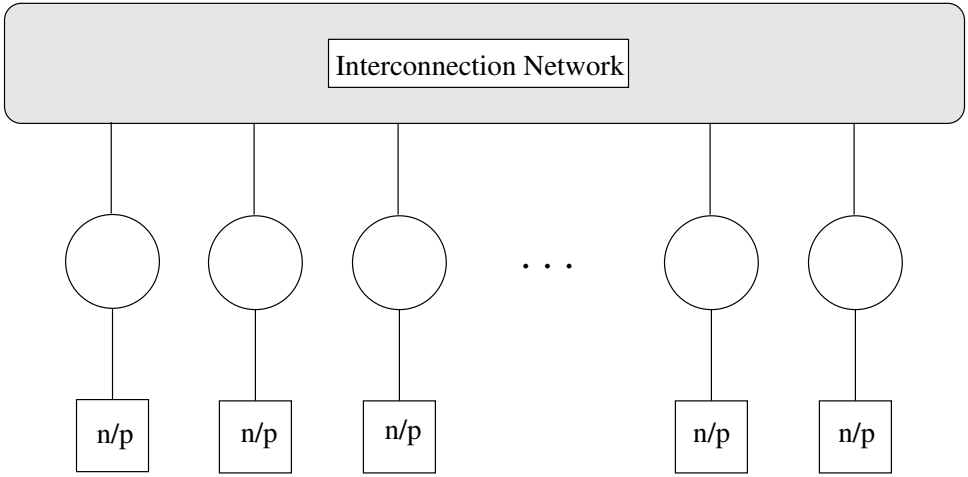
- These techniques can be combined to solve problems.
- The application of these methods, in this order, as an algorithm of two phases, is the basis of several FPT algorithms.
- FPT algorithms have been implemented and they constitute a promising approach to solve problems to get the exact solution.
- The exponential complexity on the parameter can still result in a prohibitive cost.





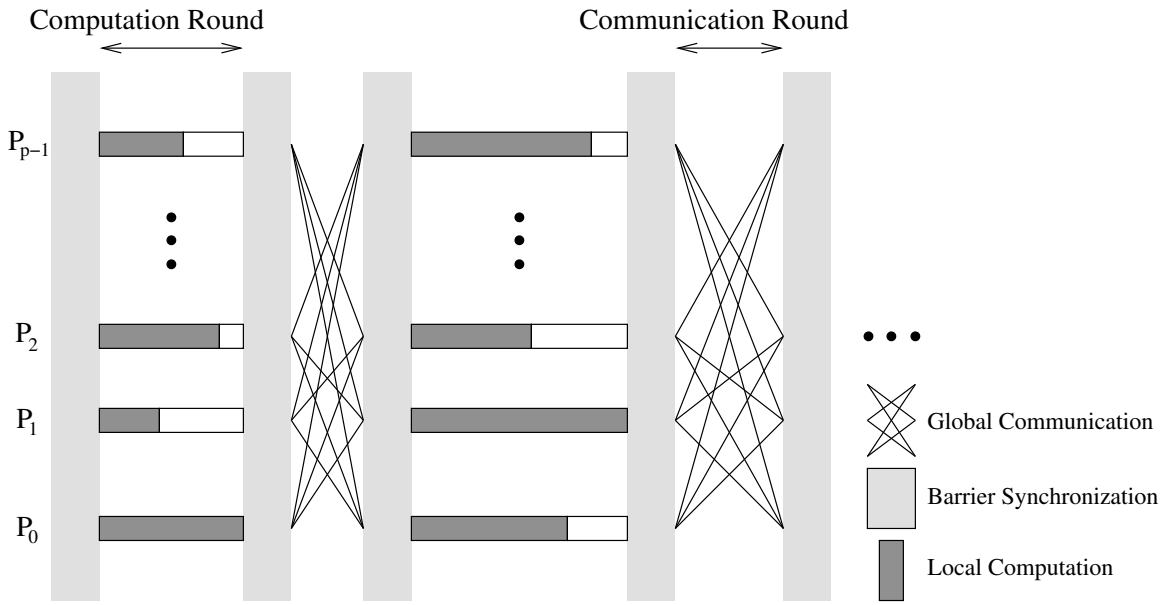

Back
End

CGM Parallel Model



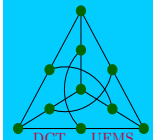
Processor

Local Memory


Computation Round
Communication Round
P_{p-1}
P_2
P_1
P_0
Global Communication
Barrier Synchronization
Local Computation

FPT Algorithms for the k-Vertex Cover Problem

- k-Vertex Cover Problem
- Algorithm of Buss
- Algorithms of Balasubramanian *et al.*
- BSP/CGM Algorithm of Cheetham *et al.*



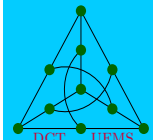



Back

End

k-Vertex Cover Problem

- We have a graph $G = (V, E)$ (the instance) and a non-negative integer k (the parameter).
- We want to answer the following question: "Is there a set $V' \subseteq V$ of vertices, whose maximum size is k, so that for every edge $(u, v) \in E$, $u \in V'$ or $v \in V'$?".
- An application of the vertex cover problem is the analysis of multiple sequences alignment.
- A trivial exact algorithm for this problem is to use brute force, and it is usually not feasible in practice.



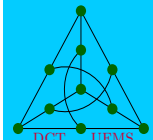




Back
End

Algorithm of Buss

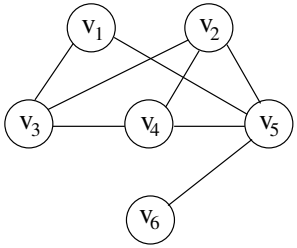
- The algorithm of Buss is based on the idea that all the vertices of degree greater than k belong to any vertex cover for graph G of size smaller or equal to k.
- Such vertices must be added to the partial cover and removed from the graph.
- If there are more than k vertices in this situation, there is no vertex cover of size smaller or equal to k for the graph G.
- Complexity time: $O(kn + 2^k k^{2k+2})$.

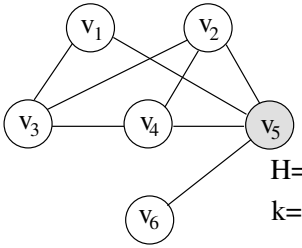






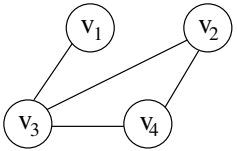
Back
End





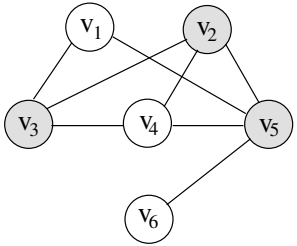
H={v5}

k=3



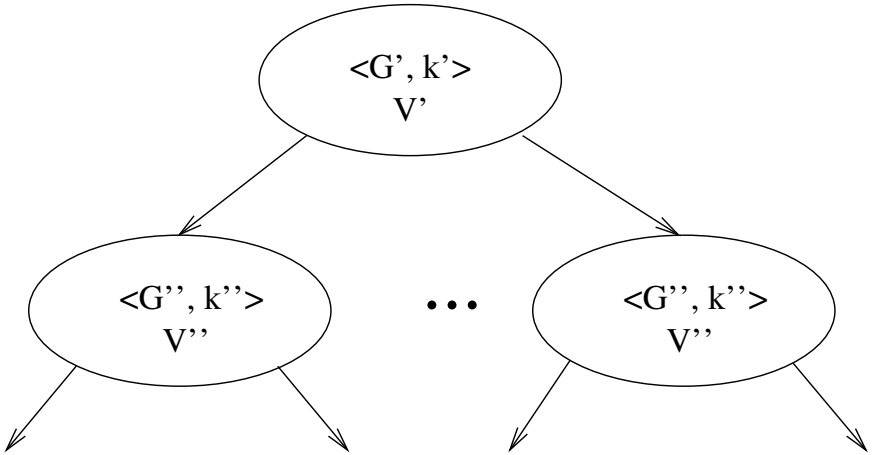
k'=3−1=2

{ } is not a V.C.
{v1} is not a V.C.
{v2} is not a V.C.
{v3} is not a V.C.
{v4} is not a V.C.
{v1,v2} is not a V.C.
{v1,v3} is not a V.C.
{v1,v4} is not a V.C.
{v2,v3} is a V.C.
{v2,v4}
{v3,v4}

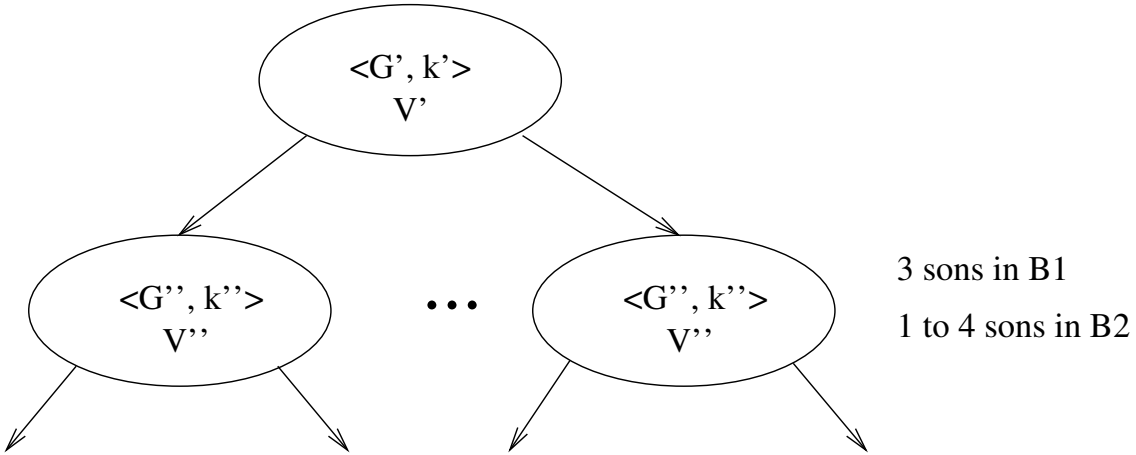


Algorithms of Balasubramanian *et al.*

- The algorithms of Balasubramanian *et al.* execute initially the phase of reduction to problem kernel based on the algorithm of Buss.
- In the second phase, a bounded search tree is generated.
- Balasubramanian *et al.* developed two algorithms to generate the bounded search tree:
  - Algorithm B1 (Complexity time: $O(kn + (\sqrt{3})^k k^2)$)
  - Algorithm B2 (Complexity time: $O(kn + 1.324718^k k^2)$)
- In both cases, we search the tree nodes exhaustively for a solution of the vertex cover problem, by depth first tree traversal.
- The difference between the two algorithms is the form we choose the vertices to be added to the partial cover and, consequently, the format of such a tree.



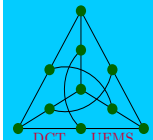
- Each node of the search tree stores a partial vertex cover and a reduced instance of the graph.
- The root of the search tree, for example, represents the graph situation after the method of reduction to problem kernel.



- The edges of the search tree represents the several possibilities of adding vertices to the existing partial cover.
- We actually do not generate all the nodes before the depth first tree traversal. We only generate a node of the bounded search tree when this node is visited.
- The growth of the search tree is interrupted when the node has a partial vertex cover of size smaller or equal to k or a resulting empty graph (case in which we find a valid vertex cover for graph G).

BSP/CGM Algorithm of Cheetham *et al.*

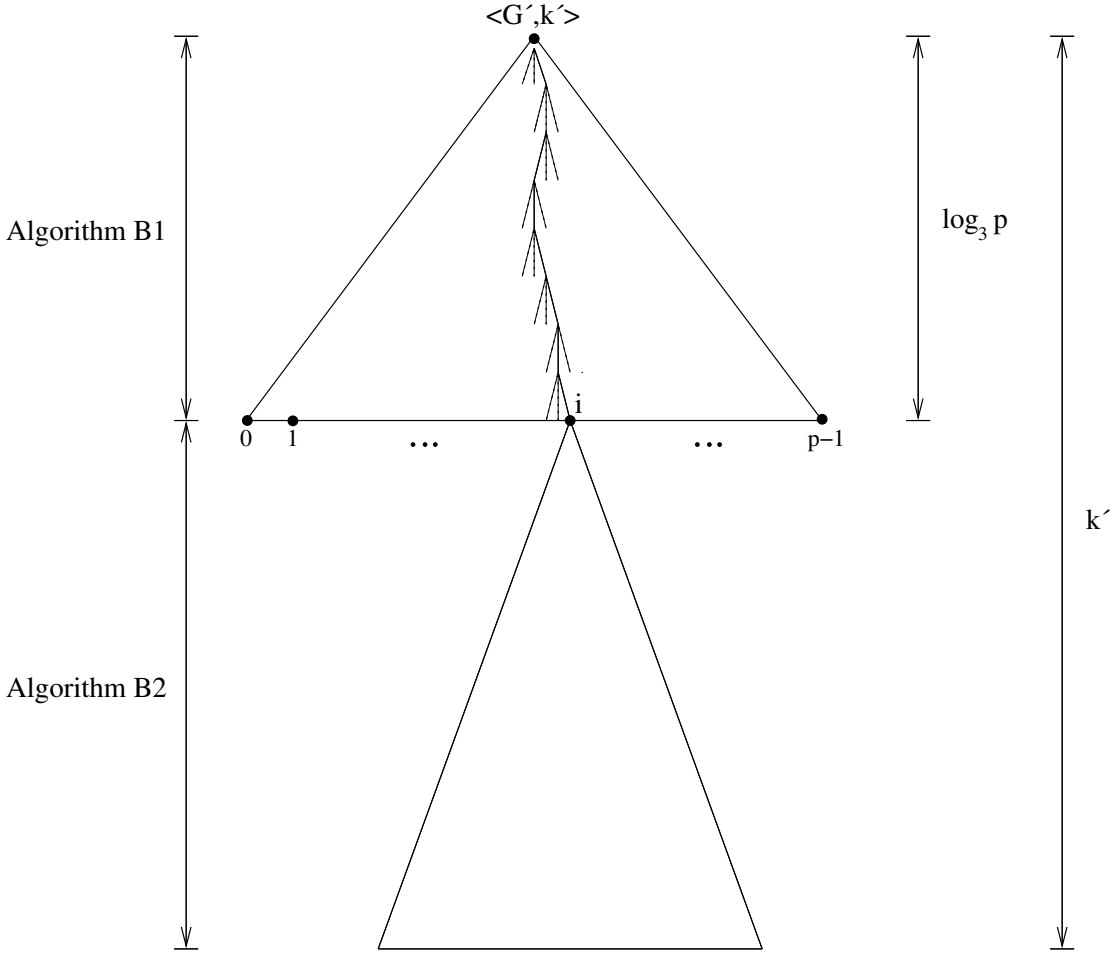
- This BSP/CGM algorithm parallelizes both phases of an FPT algorithm, reduction to problem kernel and bounded search tree.
- This algorithm solves even larger instances of the k-Vertex Cover problem than those solved by sequential FPT algorithms.
- The phase of reduction to problem kernel is parallelized through a parallel integer sorting.





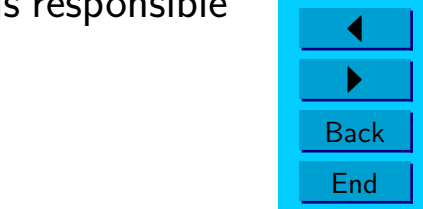


Back
End

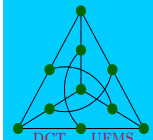

<G´,k´>
Algorithm B1
Algorithm B2
log3 p
k´
0
1
i
p−1

Implementation Details

- We used C/C++ and the MPI communication library.
- The input was a text file describing a graph G by its adjacency list and an integer k that determines the maximum size for the vertex cover desired.
- Let n be the number of vertices, m the number of edges of graph G and p the number of processors to run the program.
- At the beginning of the reduction to problem kernel phase, the input adjacency list of graph G is transformed into a list of corresponding edges and distributed among the p processors.
- Each processor P_i, $0 \leq i < p$, receives m/p edges and is responsible for controlling the degrees of n/p vertices.



- The p processors transform the list of edges corresponding to graph G' again into an adjacency list, that will be used in the bounded search tree phase.
- The resulting adjacency list from the reduction to problem kernel is implemented as a doubly linked list of vertices.
- Our program uses the *backtracking* technique.
- We need to store some information in a stack of pointers to removed vertices and edges, that enables us to go up the tree and recover a previous instance of the graph.
- The partial vertex cover is also a stack of pointers to vertices known to be part of the cover.



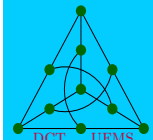




Back
End

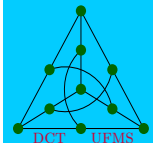
Experimental Results

- The parallel implementation is called **Par-Impl**.
- The sequential implementation of Algorithm B2 is called **Seq-Impl**.
- The sequential and parallel times were measured as wall clock times in seconds, including reading input data, data structures deallocation and writing output data.
- The parallel times were measured between the start of the first processor and termination of the last process.





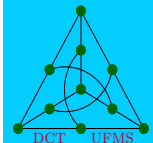

Back
End



- In our experiments we used conflict graphs that represent sequences of amino acid collected from the NCBI database:

Graph	$\|V\|$	$\|E\|$	k	k'
Kinase	647	113122	495	391
PHD	670	147054	601	600
SH2	730	95463	461	397
Somatostatin	559	33652	272	254
WW	425	40182	322	318

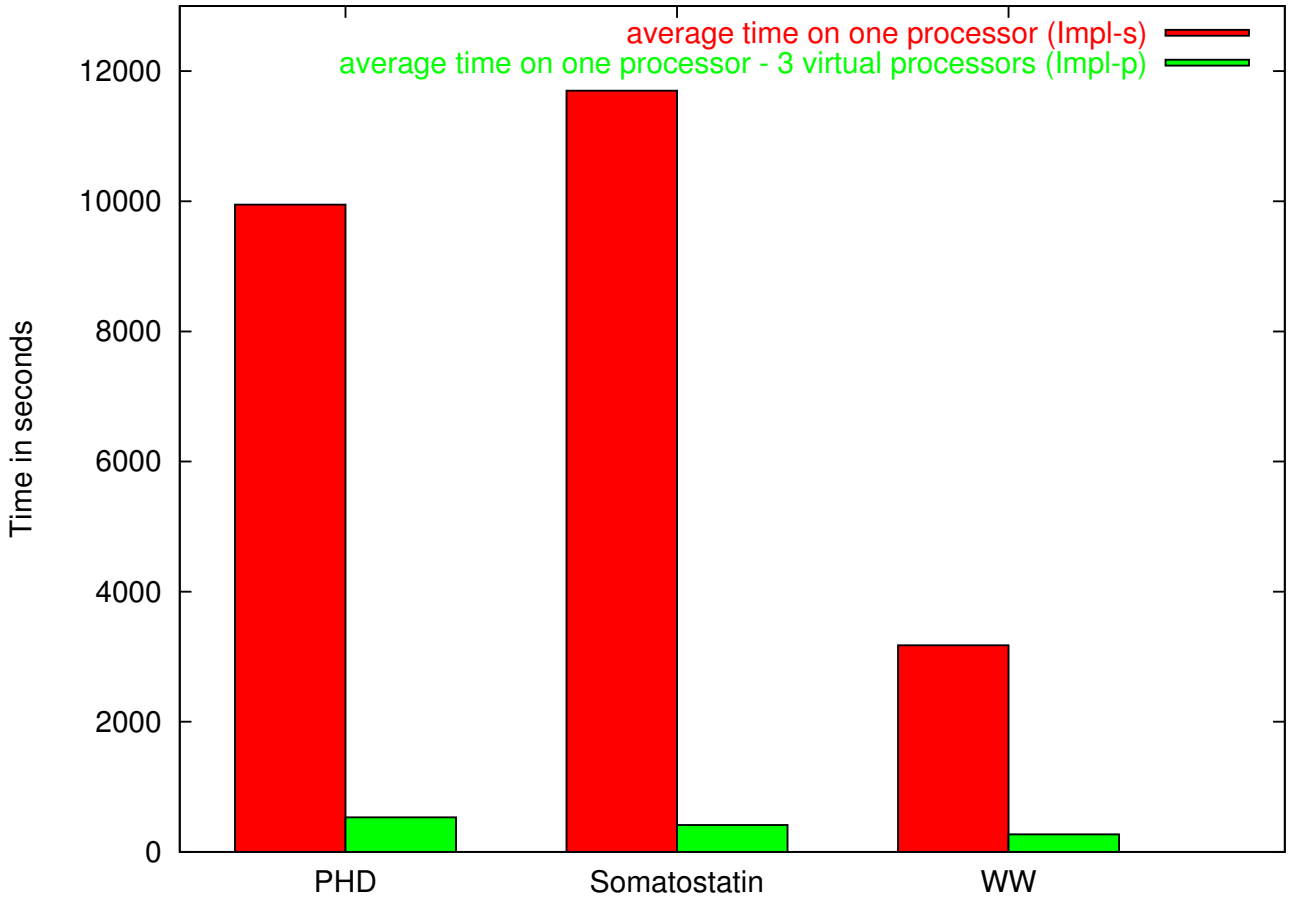
- The times were obtained by executing:
  - Seq-Impl in a single processor;
  - Par-Impl in a single processor (3 virtual processors); and
  - Par-Impl in 3, 9 and 27 processors.
- The time obtained by Par-Impl in a single processor is the sum of the wall clock times of the individual processes plus the overhead created by their communication.







Back
End


average time on one processor (Impl-s)
average time on one processor - 3 virtual processors (Impl-p)
Time in seconds
12000
10000
8000
6000
4000
2000
0
PHD
Somatostatin
WW

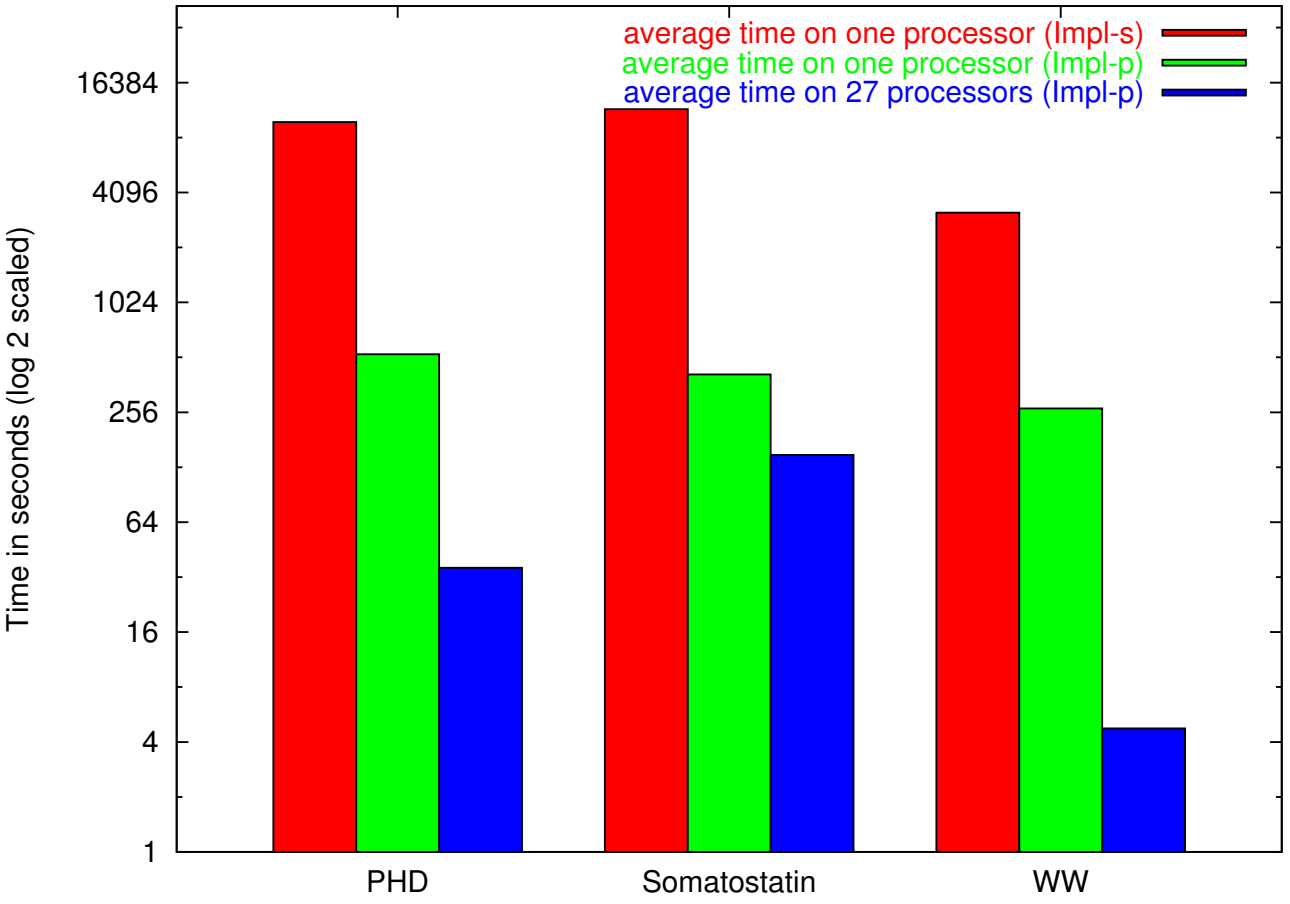
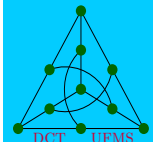


Figura 1: Average wall clock times on 3, 9 and 27 processors for PHD, Somatostatin and WW.

- Notice the increase in the number of processors does not necessarily imply a greater improvement on the average time, in spite of the always observed time reduction.
- Nevertheless, the use of more processors increases the chance of determining the cover more quickly, since we start the tree search in more points.
- It seems that the number of tree nodes with a solution also has some influence on the running times.
- As we do the depth first traversal in the bounded search tree, a wrong choice of a son to visit means that we have to traverse all the subtree of the son before choosing another son to visit.







Back
End

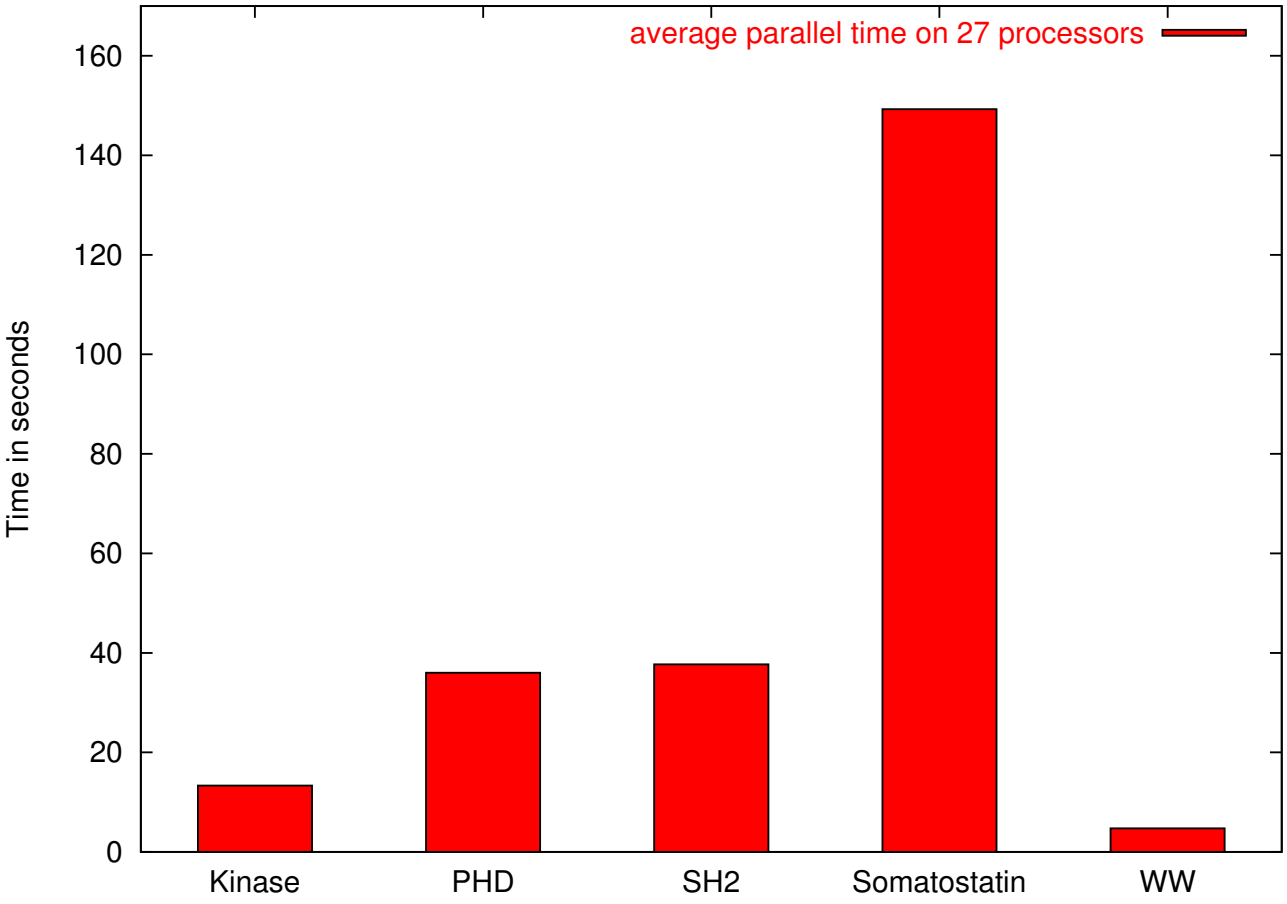
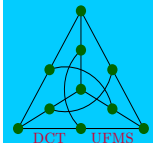


Figura 2: Average wall clock times for the data sets on 27 real processors.



- For the graphs PHD, SH2, Somatostatin and WW we could guarantee the non existence of covers smaller than that determined by the parallel algorithm, confirming the minimality of the values obtained.

Graph	Time
PHD	1.162,11 seg
SH2	4.374,14 seg
Somatostatin	272,102 seg
WW	664,25 seg




Back
End

Conclusions

- Our experiments are very relevant, since we used a computational platform that is much inferior than that used in Cheetham *et al.*.

Item	Our Environment	Cheetham's Environment
Number of processors	64	32
Processor	Pentium III	Xeon
Clock Speed	500 Mhz	1.8 GHz
Memory	256 Mb	512 Mb
Switch	Fast Ethernet	Gigabit Ethernet
C Language	g++ 2.96	g++ 2.95.3
MPI Version	MPI-LAM 6.5.6	MPI-LAM 6.5.6
OS Version	Linux Red Hat 7.3	Linux Red Hat 7.2

- The parallel times obtained in our experiments were better.

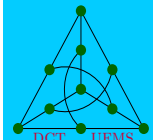
Graph	Cheetham*	Our Implementation	Improve Rate
Kinase	1550 seg	13,3273 seg	119,23
PHD	600 seg	36,0201 seg	16,65
SH2	4400 seg	37,7083 seg	116,68
Somatostatin	150 seg	149,2843 seg	1
WW	10 seg	4,7455 seg	2,1

* approximated values

- We considered that the choice of good data structures and use of the backtracking technique were essential to obtain our relevant results.




- Furthermore, the size of the covers obtained were smaller for the following graphs:
  - Kinase (from 497 to 495)
  - PHD (from 603 to 601)
  - Somatostatin (from 273 to 272).
- The reduction in the size of the cover implies the reduction on the universe of existing solutions in the bounded search tree, which in turn gives rise to an increase in the running time.

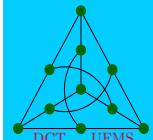






Back
End

- FPT algorithms constitute an alternative approach to solve NP-complete problems for which it is possible to fix a parameter that is responsible for the combinatorial explosion.
- The use of parallelism improve significantly the running time of the FPT algorithms, as in the case of the k-Vertex Cover problem.
- During the program design, we utilized several alternative data structures and their results were compared with those of Cheetham *et al.*.
- We chose the design that obtained the best performance.



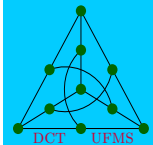




Back
End

Future Works

- Changing the algorithm used in the second phase of the parallel algorithm by a better one.
- Making practical tests with other real amino acid sequences, providing a parallel tool.
- Exploring parallelism for other FPT problems.







Back
End

Acknowledgments

- Prof. Frank Dehne (Carleton University) and Peter J. Taillon who kindly provided us the conflict graphs.
- Prof. Edson N. Cáceres (UFMS) for his assistance.
- the Institute of Computing/UNICAMP for giving the permission to use the machines.
- CNPq.
- CAPES.