# Parallel, Scalable, Memory-Efficient Backtracking for Combinatorial Modeling of Large-Scale Biological Systems[1]

Byung-Hoon Park[*], Matthew Schmidt[*,+], Kevin Thomas[#], Tatiana Karpinets[*], Nagiza F. Samatova[*,+,¥]
[*]*Computer Science and Mathematics Division,*
*Oak Ridge National Laboratory, Oak Ridge, TN 37831*
[+]*Computer Science Department, North Carolina State University, Raleigh, NC 27695*
[#]*Cray Inc.*
[¥]*Corresponding author: samatovan@ornl.gov*

## Abstract

*Data-driven modeling of biological systems such as protein-protein interaction networks is data-intensive and combinatorially challenging. Backtracking can constrain a combinatorial search space. Yet, its recursive nature, exacerbated by data-intensity, limits its applicability for large-scale systems. Parallel, scalable, and memory-efficient backtracking is a promising approach. Parallel backtracking suffers from unbalanced loads. Load rebalancing via synchronization and data movement is prohibitively expensive. Balancing these discrepancies, while minimizing end-to-end execution time and memory requirements, is desirable. This paper introduces such a framework. Its scalability and efficiency, demonstrated on the maximal clique enumeration problem, are attributed to the proposed: (a) representation of search tree decomposition to enable parallelization; (b) depth-first parallel search to minimize memory requirement; (c) least stringent synchronization to minimize data movement; and (d) on-demand work stealing with stack splitting to minimize processors' idle time. The applications of this framework to real biological problems related to bioethanol production are discussed.*

## 1. Introduction

Biological systems are inherently complex. This complexity arises from nonlinear interconnections of their functionally diverse components to produce a coherent behavior. While analytical tools that derive the components from high-throughput experimental data significantly reduce the amount of data to be dealt with, the challenge still remains of how to "connect the dots," that is, to construct predictive *in silico* models of these biological systems. The combinatorial space of feasible models is enormous, and advanced methods for constraining such a space and for efficient search of optimal solutions are in great demand.

Data-driven construction of predictive biological models is, thus, often considered as a combinatorial optimization problem, where a search for a particular object or enumeration of all the objects with given properties is being sought. The data-intensive nature of this problem, however, makes existing methods fail to meet the required scale of data size, heterogeneity, and dimensionality. High-end computing hardware and software have been well configured for running simulations. Fundamental differences exist, however, between running simulations and building data-driven biological models (Figure 1). Such differences necessitate novel algorithms with the right mix of memory, disk, and communication trade-offs. Requirements and trade-off strategies for real biological systems are poorly understood.

Exact combinatorial algorithms for biological systems frequently explore the search space recursively. Since input is typically huge (thousands or millions of nodes), it should not be copied indiscriminately in a recursive process. Storage demands are often enormous and memory management

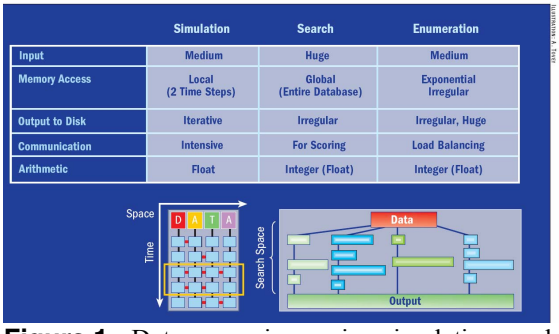


**Figure 1.** Data access in running simulations and in building biological models.

[1] This research has been supported by the "Exploratory Data Intensive Computing for Complex Biological Systems" project from U.S. Department of Energy (Office of Advanced Scientific Computing Research, Office of Science). The work of NFS was also sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory. Oak Ridge National Laboratory is managed by UT-Battelle for the LLC U.S. D.O.E. under contract no. DEAC05-00OR22725.

978-1-4244-1694-3/08/$25.00 ©2008 IEEE

is critical. Enumeration problems (such as maximal clique enumeration, MCE) can generate output exponential with the input, and may reach petabyte scale on modest-sized problems.

*Backtracking* [1] is a widely used recursive strategy. Unlike exhaustive search, backtracking avoids exploring unpromising paths by applying the specific property of the sought solution. In case of MCE, the completeness of the clique is the constraining property. All possible paths that backtracking can explore are represented as a tree, which we call *search tree*. A path (from the root) in the tree corresponds to a sought solution (e.g. clique), and the tree is expanded during traversal. Backtracking stops path expansion and backtracks to its previous level, if no further expansion in the current direction leads to feasible solutions (e.g. cliques). A search tree can be split into a number of disjoint sub-trees, which can be recursively split and independently explored.

Parallelization of backtracking requires special attention to load-balancing. Although some strategies (e.g. breadth-first traversal) may seem embarrassingly parallelizable, they inherently suffer from extremely unbalanced loads. A search tree may grow highly irregular and practically impossible to predict *a priori*. This prohibits adapting static allocation strategies with which many processors may finish exploring their search trees quickly, while very few, "unlucky" ones, would still be struggling to expand their search trees. Load rebalancing via synchronization and data movement sounds promising, but for data-intensive applications it is often prohibitively expensive. Therefore, a highly tailored strategy that minimizes the end-to-end execution time by balancing these discrepancies is particularly desirable.

Parallel backtracking can expand paths in search tree by either Breadth First Search (BFS) or Depth First Search (DFS). With BFS, coordinating the overall computation across the processors is straightforward, if each processor is enforced to expand all paths to the same level of the search tree. However, this requires storing all the search nodes at a given level in core memory. Memory requirements depend on the width of the search tree. In contrast, with DFS only information about search nodes along the path from the current search node to the root of the search tree should be stored (Figure 2). Memory requirements depend on the height of the search tree. For problems, like MCE, there is a large degree of branching in the search tree (i.e. huge width), yet the height is bounded by the size of the maximum clique. Since input can be huge, fast and memory efficient parallel backtracking is needed.

An elegant data structure that represents a decomposable task is crucial in parallel backtracking. For MCE, such a structure corresponds to an

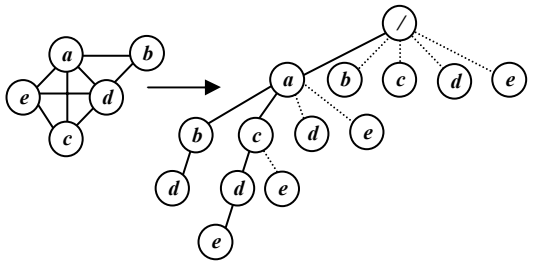


**Figure 2.** An input graph (Left) and corresponding search tree (Right) produced by BK-Base. Dotted lines in the search tree represent unexpanded paths due to the backtracking criterion of BK-Base.

expandable node in a search tree. Such a structure needs to be stored in memory for continuation of the search, and exchanged between processors for load balancing. Hence, to minimize memory and to ensure a self-guided node expansion when migrated to a different processor, a compact and self-subsistent representation of decomposition at any level of the search tree should be devised.

Based on these observations, this paper introduces a framework for parallelizing a backtracking strategy. It applies this strategy to the maximal clique enumeration (MCE) problem. The framework proposes the independent (from other search steps) and self-sufficient (for further search tree expansion) representation of search tree decomposition called *candidate path* to distribute the work of the backtracking search among the processors. It adopts Depth First Search (DFS) strategy to minimize memory requirements. It also utilizes the least stringent synchronization scheme; each processor continues to explore its regions in the search tree without intervention. For load balancing, the framework exploits *random work stealing* and *stack splitting*; when a processor becomes idle, it fetches an unexpanded (unexplored) node from a randomly selected processor. The parallelization framework is scalable for random and real-world biological graphs.

## 2. Background

The rest of the paper is largely focused on a representative combinatorial enumeration problem, called maximal clique enumeration (MCE) in graphs, which is quite ubiquitous in biological applications. *NP*-hard nature of MCE limits the applicability of existing MCE algorithms to large-scale biological problems. High-performance parallel MCE algorithms that scale to real biological problems are the focus of this study. While specific to MCE, the results of this paper can be adopted to other combinatorial enumeration problems on graphs.

A quick overview of the backtracking MCE by Bron and Kerbosh [2] (thus dubbed as *BK*) is presented here to highlight the properties that need to be incorporated into our parallelization framework. BK is the most efficient and widely accepted MCE algorithm among many others; it is used as the core in our parallelization framework. BK effectively avoids redundant branching by keeping an order among "eligible" vertices when expanding a path in a search tree.

A maximal clique in a graph can be found by visiting and marking a set of vertices that are all pair-wise connected. Since a vertex in a graph can be visited via different search paths, it can appear as different nodes in the search tree. A path is expanded to include a vertex that is connected to all vertices in the path (i.e., a common neighbor) to maintain the property of a clique. A path to a leaf node is potentially a maximal clique. It is important to identify leaf nodes as early as possible so that unnecessary expansion can be avoided. Backtracking expands a search tree (or branched) with some extra constraints (or bounded) in addition to the *common neighbor* criterion.

Let us assume that a search node has $k$ children representing the eligible (common neighbors) vertices $v_1$, $v_2$, $v_3$, …, $v_k$ from which to expand the path $P$ leading to the search node. Then $P$ can be expanded into $k$ paths, $P_1$,…, $P_k$ so that each path includes the corresponding $v_i$. BK prevents appearances of $v_i$ in any of $P_j$ ($i<j$) and its future expanded paths. BK backtracks after expanding $P_i$, if $v_i$ is a common neighbor to all subsequent $v_j$-s ($i<j$). In other words, expansions to the rest $P_i$-s are not necessary. Not only redundant branching is avoided, but also a sorted output is produced. Figure 2 pictorially illustrates this backtracking behavior of BK. An improved version of BK, called *BK-Improved*, improves the performance of the base BK (*BK-Base*) by dynamically identifying the most suitable ordering in expanding each path. It first expands the child that is connected to the largest number of other children and then expands each of the children that are not adjacent to this child. This insures that the condition to stop expanding children nodes is reached by expanding the fewest children possible.

## 3. Related Work

Our previously developed *pClique* [12], the first (and only so-far existing) parallel MCE algorithm, extends the algorithm of Kose *et al* [13] (dubbed as KOSE). In principle, KOSE is identical in spirit to BK-Base; it branches with the alphanumerically ordering. However, whereas BK is recursive with DFS branching, KOSE is serial BFS branching. This property allows cliques of size $k$ to be generated from cliques of size *k-1*, similar to an association rule mining algorithm *a priori* [14]. All maximal cliques are produced in non-decreasing order, an invaluable asset to certain applications. BFS branching inevitably makes KOSE memory-intensive. Although *pClique* improves KOSE by bit-vector manipulation of common neighbors, huge memory requirements remain. This limits its applicability to small size graphs (~5,000 vertices, <10,000 edges).

## 4. Method

Most backtracking enumeration algorithms are recursive, thus are not readily suitable for parallelization. Their parallelization requires converting a recursive step into a sequential version such that each recursion step becomes independent on previous steps. Such a "sequentialization" should result in memory manipulation that will substitute system stack operations. What is needed is a self-sufficient decomposition of the search tree that makes each expansion (or backtracking) of the search tree *independent* of information about previous searches. We first propose a sequentialization of recursive backtracking that leads to an independent decomposition of the backtracking search tree, and illustrate its BFS- and DFS-based implementations in the context of the MCE problem. We then devise a parallel framework based on this decomposition.

### 4.1. Independent Search Tree Decomposition

When recursive backtracking expands a search path at a certain node, all information needed for expansion (or backtracking) is stored in the system stack. It is stored and retrieved following the Last In First Out (LIFO) order. Since a sequential algorithm does not operate on such a system stack, all information should be supplied during the expansion regardless of the order of expansion steps. This necessitates a representation of such information for expansion.

For MCE, our devised representation embeds:

- The clique represented by the path from the root to the current node in the search tree.
- All eligible vertices for the path (i.e. common neighbors for all the nodes in the above clique).
- Vertices covered earlier in expanding the parent path (to avoid redundant coverage).

We dub a data structure that includes all these information as a *candidate path*. It is of order $k$ if the clique represented by the path is of size $k$.

***Sequential-Backtracking-Clique-Enumeration***

1. Insert all order one candidate paths in the queue.
2. While queue is not empty
3. Retrieve a candidate path $C_i$ from the queue.
4. Expand $C_j$ to paths $C_j$ -s at the next level.
5. For each $C_j$
6. Print out $C_j$ if it is maximal.
7. Otherwise, put $C_j$ at the tail of the queue.

**Figure 3.** A framework of sequential backtracking MCE. Breadth First Clique (BFC) and Depth First Clique (DFC) are differ in retrieving a candidate path from the queue (Step 3). BFC(DFC) retrieves a candidate path from the queue head(tail).

### 4.2. BFS- and DFS-based Sequentialization of Recursive Backtracking MCE

Since a candidate path is sufficient for further expanding the path, the sequential enumeration is not affected by the order these candidate paths are stored and retrieved. Breadth-first or depth-first based sequential versions can be implemented by employing different strategies in storing and retrieving candidate paths in a given memory queue – First In First Out (FIFO) for BFS and Last In First Out (LIFO) for DFS.

We call the BFS-based BK-Improved as Breadth First Clique (BFC) and the DFS-based BK-Improved as Depth First Clique (DFC). An algorithmic description is shown in Figure 3.

### 4.3. Parallelization of BFC and DFC with Dynamic Load Balancing

Parallelization of a backtracking MCE is based on the idea of "dynamic search tree decomposition." Each search tree region is assigned to a processor for further expansion. The region is the candidate list structure corresponding to the node at the root of the sub-tree being assigned. Various assignments are possible depending on the priority of parallelization schemes. If fully balanced loads amongst the processors are desired, a random allocation strategy where every newly expanded node is assigned to a randomly chosen processor could be explored. The communication cost for such a strategy may be overwhelming for large graphs, considering the possibly enormous number of candidate paths generated during enumeration.

On the other hand, if only an initial decomposition of the search tree is done, a processor is responsible for searching the entire search space of all the sub-trees it was initially assigned. Because the search tree for BK is highly irregular, this leads to an extremely unbalanced load among the processors. It is practically impossible to predict *a priori* the size of each sub-tree and the time required to expand it.

Based on these observations, we choose to adopt a dynamic load balancing scheme based on the *random stealing* [3, 4] with a simplified *stack splitting* [5]. Since the final size of a sub-tree is difficult to predict, an initial random assignment of sub-trees to the processors is deployed. Each processor then continues on independently until it finishes enumerating all maximal cliques derived from its assigned sub-trees by fully expanding its initially assigned candidate paths. When no more candidate paths exist in the processor's stack, a processor sends a request to a randomly chosen processor for a candidate path. If no candidate path is received after trying all the other processors, the processor halts. In spite of its simplicity, *random stealing* was shown to provide a more scalable solution to a dynamic load balancing when compared with other strategies across different hardware architectures [5].

Upon receiving a request, the processor sends a candidate path of the lowest order from the queue. Since a candidate path is not yet fully explored, one of lower order tends to spawn a larger number of higher order candidate paths. For BFC and DFC, this scheme is realized by exchanging a candidate path that is retrieved from the queue head. We implement a parallel framework for both BFC and DFC by adapting this simple *stack splitting* paradigm. Parallel BFC (pBFC) and DFC (pDFC) are illustrated in Figure 4.

***Parallel-Backtracking-Clique-Enumeration***

1. Each processor starts with a set of order one candidate paths that are stored in the queue.
2. Retrieve a candidate path $C_i$ from the queue.
3. If queue is empty
4. Receive a candidate path from an arbitrarily chosen processor and put it into the queue.
5. If no path is received after some trials
6. Go to Step 16
8. Go to Step 2.
10. Expand $C_i$ to generate candidate paths $C_j$-s at the next level.
11. For each $C_j$
12. Print out $C_j$ if it is maximal.
13. Otherwise, put $C_j$ at the tail of the queue.
15. Go to Step 2.
16. Halt.

**Figure 4.** Parallel branch and bound MCE. Parallel CBF (pCBF) and CDF (pCDF) are different in retrieving a candidate path from the queue (Step 2).

## 5. Results

This section reports on the scalability of pDFC over multiple processors using empirical results. Scalability in terms of both memory requirements and runtime are discussed. We compare memory requirements of BK-Improved on a small graph of ~1000 vertices implemented as a BFS (BFC) and as a DFS (DFC). We also look at memory distribution in multiple processors for parallel implementations of these algorithms (pBFC and pDFC, resp.). Runtime of pDFC is examined for medium sized dense and large sized sparse graphs. Runtime of pBFC cannot be measured due to large memory demands. Both pBFC and pDFC are shared memory, multithreaded. All measurements were on an SGI Altix 3700.

### 5.1. Memory Requirements

Memory requirements of both BFC and DFC depend on the size of candidate paths that should reside in memory for the search to continue on. For BFC, all candidate paths of order $k$ should be accessible to produce candidate paths of order $k+1$, and an order $k$ path can be safely removed once it is expanded. The BFC memory requirement is bounded by the largest candidate path set of order $k$ and $k+1$. For DFC, only candidate paths generated along the current path need to be stored in memory. The deepest path of the search tree bounds the DFC memory requirement.

To empirically verify the observation, we measure the memory usage of both BFC and DFC clique enumeration algorithms on a graph that has 858 vertices and 10,823 edges. The graph has a total of 12,631 cliques that contain at least 3 vertices. The largest clique size is 21. Figure 5 shows the memory usages of both algorithms that are measured after every 200 new cliques are found during their runs. As anticipated, the DFC memory usage is high in the beginning and decreases with some fluctuations as more cliques are found. In contrast, the BFC memory usage gradually increases and then decreases forming a bell shaped curve. In summary, the overall memory requirement of BFC is much larger than that of DFC.

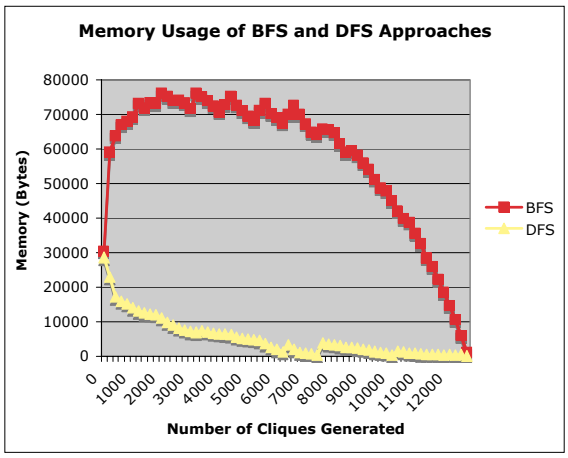


**Figure 5.** BFS and DFS MCE memory usage measured every time 200 new cliques are found.

A memory-efficient parallel algorithm should distribute memory requirement as evenly as possible across the participating processors. As shown in Figure 6, for both pBFC and pDFC, average memory requirement per processor is decreased as more processors are used. However, unlike in pDFC, variance in memory requirement per processor is very wide in pBFC. Typically memory needed to expand such a large order candidate path is substantially smaller than that to expand lower order ones. Since candidate paths of larger order are likely exchanged with pBFC, Figure 6.a-b empirically verifies this

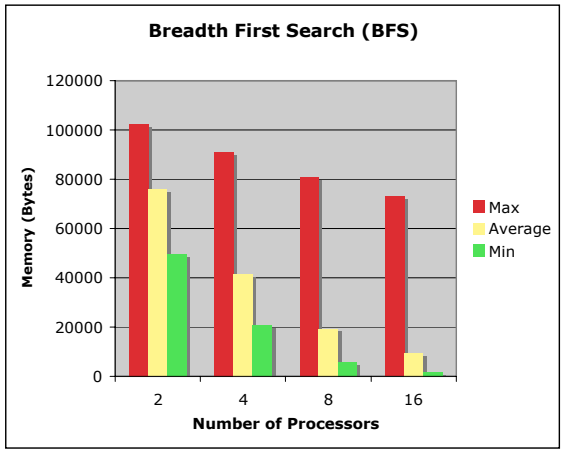


**Figure 6a.** Average, minimum, and maximum memory per processor to finish BFS clique enumeration (over 10 different runs).

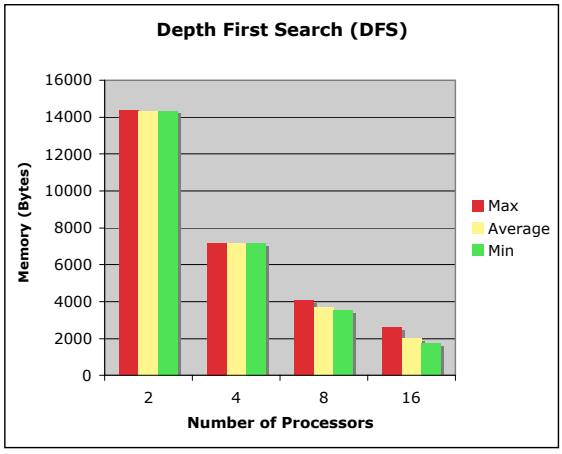


**Figure 6b.** Average, minimum, and maximum memory per processor to finish DFS clique enumeration (over 10 different runs).

observation. Thus memory requirement balancing will be better achieved with pDFC.

### 5.2. Scalability

Since pDFC is more memory efficient than pBFC, pDFC is a better candidate to produce an efficient scalable parallel algorithm. To test this we study the scaling of pDFC's runtime and memory requirements as the number of processors increases. The results shown in Figure 7.a-b are produced from pDFC runs on a dense graph with 5,000 vertices and 2,496,740 edges. The graph is randomly generated according to a given edge probability (to control graph density) and maximum clique size. The graph has 1,074,127,772 maximal cliques with the largest clique size of 70.

The runtimes of pDFC on the generated graph for up-to 128 processors are shown in Figure 7.a. The runtimes measured did not include the time spent to read in the input graph and to save the maximal cliques to secondary devices. As shown in the Figure 7.a, a near linear speedup is observed for up to 8 processors. Different initialization schemes and parallel I/O methods could be used to ensure that this linear scalability is maintained for large numbers of processors over the total time of the algorithm, but those methods are beyond the scope of this paper.

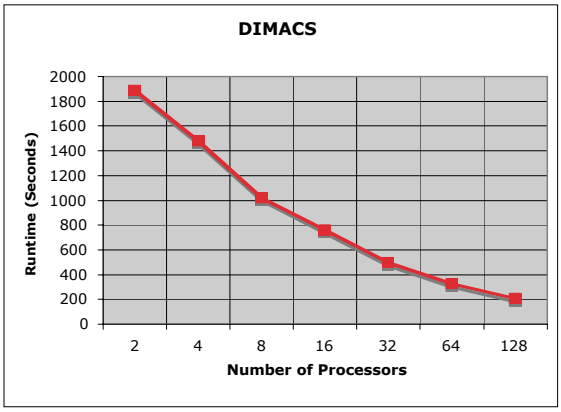


**Figure 7a.** pDFC runtimes for a DIMACS random graph of 5,000 nodes and 2,496,740 edges.

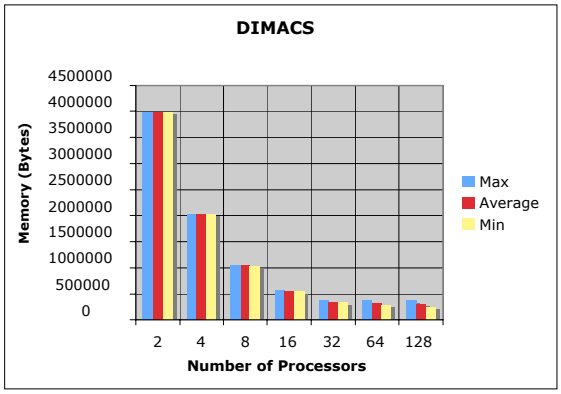


**Figure 7b.** pDFC average, maximum, and minimum memory requirements across processors.

Figure 7.b delineates pDFC's memory requirement spread over multiple processors. It shows the maximum, the average, and the minimum memory requirement of a processor in each case that involves 1, 2, 4, 8, 16, 32, 64, and 128 processors, respectively. The memory distribution remains nearly even as up-to 32 processors are used. The memory size for each processor decreases linearly for up-to 32 processors.

Since many of the graphs derived from real-world biological problems exhibit a scale-free nature, we adopted *GTgraph* (http://www-static.cc.gatech.edu/~kamesh/GTgraph/) to generate a scale free real-world type graph of 100,000 vertices, 2,000,000 edges, and 504,976 maximal cliques. The generation of the graph is based on *R-MAT* [6]. In addition, we considered a real-world phenotypic gene network derived for 80 genomes as described in Section 5.3. The edge degree distribution of vertices for this phenotypic graph exhibits a scale-free property (i.e., embeds a power law distribution). The graph had 193,568 vertices, 2,260,872 edges, and 395,306 maximal cliques.

As can be seen from Figure 9, the speedup for the pDFC algorithm when run on these scale-free graphs was nearly linear for 2, 4, and 8 processors. Also, the memory requirements per processor were found to be highly scalable and relatively balanced (Figure 8).

### 5.3 Application of pDFC to Bioethanol-Related Biological Problems

**5.3.1 Identifying Key Genes for Efficient Bioethanol Production.** Efficient production of ethanol from biomass using a bacterial consortium requires certain phenotypic traits be present in the microorganisms. The beneficial traits include the ability to metabolize different sugars found in biomass, resistance to ethanol in the environment, and facultative anaerobic respiration. It is thus important to link desirable microbial traits with specific genes that are likely to be important for these traits. To predict such bioethanol-related genes, we divide the organisms according to the presence or absence of a trait and to search for genes that are dominant on one side of the divide but not the other. The underlying intuition is that if a gene were critical to a trait, then it would be conserved by evolution. Hence, genes that are crucial to the trait will cluster, or form a clique, on one side of the divide. The problem can then be formulated as the one of finding maximal cliques of genes conserved throughout evolution among the organisms possessing a trait.

Due to a limited availability of sequenced genomes with characterized phenotypic traits, we initially

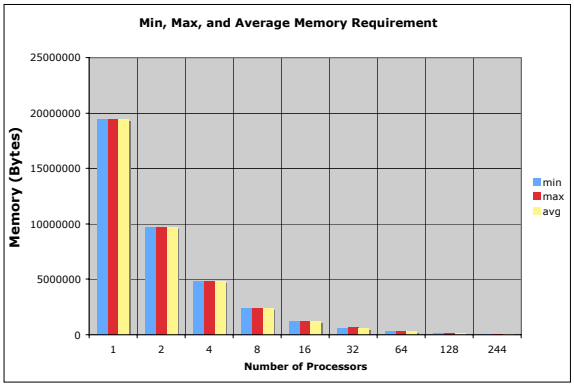


**Figure 8a.** Minimum, maximum, and average memory requirements per processor with respect to a total number processors involved. Run for phenotypic network of 80 genomes

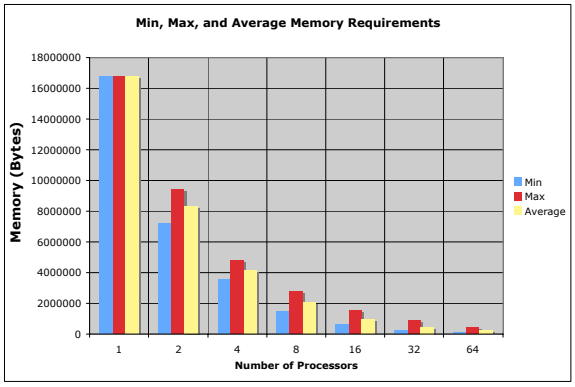


**Figure 8b.** Minimum, maximum, and average memory requirements per processor with respect to a total number processors involved. Run for synthetic graph by GTgraph.

focused on aerobic (growing with oxygen) versus anaerobic (growing without oxygen) comparison to reveal this phenotype specific orthologous genes. The details of this study and biological findings are presented elsewhere. Here, we used the phenotypic gene graphs derived from various genomes used in that study to understand computational resource requirements for enumeration of all maximal cliques in such graphs. The size of a graph will grow substantially as more genomes are added (each microbial genome has about 4,000 genes). To estimate the requirements to finish the job within the allowable time, we first selected 10 genomes from each phenotypic group (i.e. 20 in total) and constructed a graph. Subsequently, we repeatedly selected 10 additional genomes from each group, creating the graphs of 40, 60, and 80 genomes. From the four graphs thus created, we measured the growths of (1) the number of genes, (2) the number of edges, (3) the number of cliques, and (4) memory requirement. For these cases, the number of genes, the number of edges, and the memory requirement grow linearly, whereas the number of cliques grows quadratically with the number of genomes.

**5.3.2. Characterizing a Stress Related Gene Network of Ethanol Producing Yeast.** Ethanol is produced from the fermentation of sugar by yeast. In general, to enhance the productivity of bioethanol by yeast, thermochemical pretreatment of plant material is applied, which results in high concentration of toxic non-sugar constituents. Such changes in the hydrolysate make yeast exposed to a mixed and interrelated group of different stresses such as osmotic, oxidative, and thermic [7]. These stressful conditions, in combination, significantly impair the fermentative process and make yeast less tolerant to ethanol. Currently, factors that may enhance stress resistance of the yeast cells without affecting their growth are poorly understood [8].

Objectives of our study were (1) to identify cellular processes in yeast that are consistently co-regulated in response to different stresses and (2) to sift specific genes that might increase the tolerance level of yeast to ethanol under stressful conditions. For this, we first construct a gene network induced by the stresses and infer related cellular processes. More specifically, we use the gene expression profiles of the *S. cerecisae* in 173 conditions that represent response of the yeast cells to environmental changes (heat and osmotic stress, nutrient and carbon starvation, stationary state). For more details of the data, refer to [9]. The stress-induced gene network was built by selecting gene pairs with similar expression patterns across the conditions. Nodes in the network are genes and edges are drawn between genes with similar expression profiles. We then apply pDFC to find all maximal cliques in the network. To infer meaningful clusters of co-regulated genes, we applied a stringent post-processing step that iteratively merges highly overlapping cliques, and produces a reduced number of clusters.

We have analyzed the biological processes represented by the gene clusters using the KEGG pathway information on *S. cerecisae* and GO information downloaded from the Saccharomyces Genome Database (SGD) (http://www.yeastgenome .org). Here we report 7 confirmed clusters: chaperone related genes, ribosome and translation, L-asparaginase II, oxidative phosphorylation enzymes, retrotransposon TYA Gag and TYB Pol genes, stress-induced enzymes, TCA cycle enzymes and transporters. All other genes are up-regulated. The largest cluster (ribosome and translation), which is suppressed by all stresses, represents cellular processes of the ribosome biogenesis, tRNA processing and protein translation. The rest clusters represent specific cellular processes

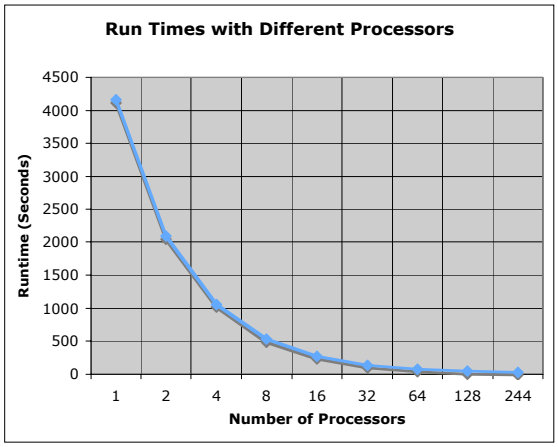


**Figure 9.** Run times with respect to a total number of processors.

generally activated by the studied stressful environments [10]. We then compared the set of ethanol-tolerance genes identified in a previous study [11], with genes in the clusters revealed by our study. We find that the first cluster referred to as "Ribosome and translation" includes two genes FEN1 (fatty acid elongase required for sphingolipid formation) and SUR4 (sterol isomerase, fatty acid elongase) that are essential for growth of *S. cerevisiae* under high ethanol concentration. Average down-regulation of these genes among 173 stressful conditions was 77% for SUR4 and 51% for FEN1. This decrease in the expression is the greatest if compared with 1 - 29% decrease among the rest down-regulated genes required for ethanol tolerance. This indicates that a shortage of these enzymes may have a crucial effect on sensitivity of yeast to ethanol. Clustering SUR4 and FEN1 with the ribosome biogenesis and translation related genes shows that down-regulation of both enzymes and the resulting ethanol sensitivity may be a part of the general stress response program in yeast. Both processes are co-regulating; therefore a decreased production of the enzymes, important for ethanol tolerance, may inevitably follow any stressful conditions in the yeast environment.

## 6. References


[1] G. Brassard and P. Brately, Fundamentals of Algorithmics, *Prentice Hall,* 1996.

[2] C. Bron and J. Kerbosch, Algorithm 457: finding all cliques of an undirected graph, *Commun. ACM,* vol. 16, pp. 575-577, 1973.

[3] A. A. Umut, E. B. Guy, and D. B. Robert, The data locality of work stealing, *The 12th annual ACM symposium on Parallel algorithms and architectures,* Bar Harbor, Maine, 2000.

[4] J. Joxan, E. S. Andrew, H. C. Y. Roland, and Q. Z. Kenny, Scalable distributed depth-first search with greedy work stealing, *The 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'04)*, 2004.

[5] K. Vipin, Y. G. Ananth, and V. Nageshwara Rao, Scalable load balancing techniques for parallel computers, *J. Parallel Distrib. Comput.,* vol. 22, pp. 60-79, 1994.

[6] D. Chakrabarti, Y. Zhan, and Christos Faloutsos, R-MAT: A recursive model for graph mining, *SIAM Intern'l Conference on Data Mining,* 2004.

[7] R. Pérez-Torrado, J. M. Bruno-Bárcena, and E. Matallana, Monitoring stress-related genes during the process of biomass propagation of *Saccharomyces cerevisiae* strains used for wine making, *Appl Environ Microbiol,* vol. 71, pp. 6831-6837, 2005.

[8] M. Versele, J. M. Thevelein, and P. V. Dijck, The high general stress resistance of the *Saccharomyces cerevisiae* fil1adenylate cyclase mutant (Cyr1 Lys1682) is only partially dependent on trehalose, Hsp104 and overexpression of Msn2/4-regulated genes, *Yeast,* vol. 21, pp. 75-86, 2004.

[9] A. P. Gasch, P. T. Spellman, C. M. Kao, O. Carmel-Harel, M. B. Eisen, G. Storz, D. Botstein, and P. O. Brown, Genomic expression programs in the response of yeast cells to environmental changes, *Mol. Biol. Cell,* vol. 11, pp. 4241-4257, 2000.

[10] P. G. Audrey and W.-W. Margaret, The genomics of yeast responses to environmental stress and starvation, *Functional & Integrative Genomics,* vol. 2, pp. 181-192, 2002.

[11] F. van Voorst F, J.L Houghton-Larsen, M.C. Kielland-Brandt, A. Brandt, Genome-wide identification of genes required for growth of *Saccharomyces cerevisiae* under ethanol stress, *Yeast,* vol. 23, pp. 351-359, 2006.

[12] Y. Zhang, F. N. Abu-Khzam, N. E. Baldwin, E. J. Chesler, M. A. Langston, and N. F. Samatova, Genome-scale computational approaches to memory-intensive applications in systems biology, *The 2005 ACM/IEEE conference on Supercomputing*: IEEE Computer Society, 2005.

[13] F. Kose, W. Weckwerth, T. Linke, and O. Fiehn, Visualizing plant metabolomic correlation networks using clique-metabolite matrices, *Bioinformatics,* vol. 17, pp. 1198-1208, 2001.

[14] R. Agrawal and R. Srikant, Fast algorithms for mining association rules, *The 20th Intern'l Conference on Very Large Data Bases (VLDB),* pp. 487--499, 1994.