Parallel Algorithms for Graph Cycle Extraction Using the Cyclical Conjunction Operator

G. Cerruela García, I. Luque Ruiz,* and M. A. Gómez-Nieto

Department of Computing and Numerical Analysis, University of Córdoba,
Campus Universitario de Rabanales, Edificio C2, Planta-3, E-14071 Córdoba, Spain

Received March 26, 2002

With a view to reducing the computational cost of extracting all the cycles from complex graphs, the authors have examined the viability here of parallel processing. Based on the cyclical conjunction operator, which uses an iterative process to extract every cycle from a graph, a study was performed of the factors intervening in the parallelization of this algorithm, namely the following: granularity of the parallel algorithm, requirements for synchronization points, and the spreading of the load across different processors. Tests were performed on two granularities and four different load distributions. Algorithm implementation is carried out using SGI MP and OpenMP libraries, and, in the light of the present findings, the authors propose a dynamically distributed fine-grain algorithm using that allows all the cycles in a complex graph to be found in an acceptable computational time.

1. INTRODUCTION

There are often real-world problems whose solution requires excessive time when using commonly available computational resources. In an effort to reduce calculation time, researchers are working in two often-coincidental directions: the design of new, more efficient algorithms, and the parallelization of sequential algorithms.

In computational chemistry, parallel techniques[1] have been widely applied to the solution of complex problems. Molecular simulations are very important tools for the calculation of macroscopic and miscroscopic properties of chemical and biochemical systems. The calculation of these properties is usually carried out by Monte Carlo (MC)[2] and Molecular Dynamics (MD)[3,4] methods, in which parallel algorithms are widely used because of the high number of linear equations to solve which can be computed independently. Although the algorithms are not equally efficient on any type of parallel computer.[5]

Computer-Assisted structure elucidation (CASE),[6] 3D structure prediction,[7] calculation of molecular electronic structure properties,[8] management of chemical structure database,[9,10] searching DNA sequences,[11] and others much, are some of the multiple applications of parallel computing on chemical sciences.[12]

The extraction of the present system of rings in a graph is another complex problem that requires a high computational effort. Information concerning the number, size, and composition of a compound's rings is widely used, alongside other topological descriptors, to tackle many issues in chemistry, such as the following:

•Deriving the relationship between structure and physicochemical properties of compounds (QSPR/QSAR applications).[13−15]

•Information clustering and screening criteria for searching huge databases of compounds.[16−19]

•The development of many other solutions for measuring similarity/diversity or molecular complexity, substructure searches in complex structures, etc., used both in the aforementioned applications and in the design and synthesis of new compounds.[20,21]

Many parallel algorithms have been developed for graph analysis, on account of the computational complexity involved therein. Of these, some aim to extract the Hamiltonian cycles from the graph.[22−25]

Working from the algorithm developed by Edmonds,[26] Nepomniaschaya[27] proposed a parallel algorithm based on a different parallel model, aimed at finding the critical cycles, while Boukerche[28] presented an algorithm to trace the local cycles and nodes in distributed parallel processes.

While a great number of algorithms have been proposed for the extraction of all the cycles from a graph representing a chemical compound, it has constantly been the case that excessive computational time is required for complex graphs,[29−31] suggesting a need for parallel techniques.

Parallel algorithms are closely linked to parallel computing architectures. A parallel computing system comprises a set of processors (CPUs) linked by some kind of communication network, giving rise to a wide variety of parallel systems ranging from a group of workstations connected through a local network to computers composed of hundreds of CPUs linked by high-speed networks. Clearly the speed of the CPUs and the communications network will play decisive roles in the efficiency of any parallel algorithm, yet from the programmer's perspective it is more important how the CPUs are controlled and how information is shared among them.

Figure 1 displays the different structures into which parallel computers may be classified. The *control model* shows how multiple instructions are executed simultaneously. Two modes—*SIMD* (Single Instruction, Multiple Data) and *MIMD* (Multiple Instruction, Multiple Data)—have been the distinguishing features of parallel computers since their concep-

* Corresponding author phone: +34 957 212082; fax: +34 957 218639; e-mail: ma1lurui@uco.es.

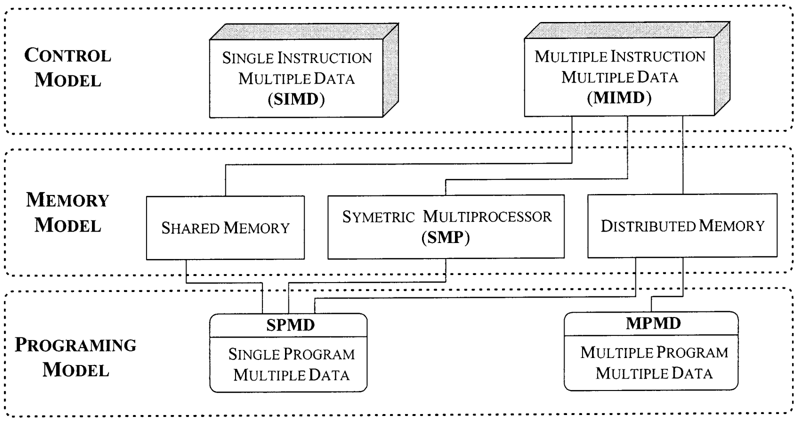
CONTROL MODEL
SINGLE INSTRUCTION MULTIPLE DATA (SIMD)
MULTIPLE INSTRUCTION MULTIPLE DATA (MIMD)
MEMORY MODEL
SHARED MEMORY
SYMETRIC MULTIPROCESSOR (SMP)
DISTRIBUTED MEMORY
PROGRAMING MODEL
SPMD
SINGLE PROGRAM MULTIPLE DATA
MPMD
MULTIPLE PROGRAM MULTIPLE DATA

Figure 1. Structure of the parallel systems.

tion;[32] this distinction remains true today, although it is not the only feature by which parallel architectures may be identified.

The *memory model* indicates how many CPUs can directly access a given memory area. In shared-memory machines, every CPU can access all memory areas, whereas in distributed-memory machines memory is local to each processor. In symmetrical multiprocessors, memory is common to a small group of processors, so in the context of system memory it remains local to these groups.

The *programming model* shows the restrictions in the number of executable items taking part in parallel execution. In the *MPMD* (Multiple Program, Multiple Data) model the programmer builds independent programs for each CPU, while with the *SPMD* (Single Program, Single Data) model the same program is run on every CPU.

Interaction of the control model with the memory model gives rise to four classes of parallel architectures: *SIMD*, *shared memory*, *distributed memory*, and *SMP* (see Figure 1):

•In computers using the *SIMD* model, such as the MAS PAR MP-2 or the Thinking Machine Connection Machine, the application (problem) must follow a totally synchronous parallel model, since this kind of architecture would make it tremendously difficult to do otherwise.

•In *distributed memory* multiprocessors such as the Cray T3D, IBM SP-2, Intel Paragon, and Meiko CS-2, each CPU has local memory with private access to its address space.

•In *shared memory* multiprocessors such as the Cray Y/MPs or the Fujitsu VPs, the processors access memory in a single set of shared addresses. Currently the trend is to use *distributed shared memory*, as seen in the SGI Origin 2000 supercomputer, the machine used in the development of the present study.

•Symmetrical multiprocessors or *SMP*s, such as the SGI Power Challenger and Sun Spaceserver, combine several (usually four or eight) CPUs with a common memory system. The term “symmetrical” arises from the way each CPU can access any memory address with no speed loss.

The present paper describes the implementation of parallel techniques in the extraction of all cycles from a graph, using the cyclical conjunction operator.[33] First, there is an overview of the cyclical conjunction operator, as detailed in a previous article,[33] describing how parallel algorithms may improve calculation speed for cycles in complex graphs. To this end, the application to be parallelized is analyzed and categorized by application group, and the theoretical expectation for the benefit resulting from the application of parallel algorithms is obtained. In section 3 there is a description of the various algorithms developed, taking into account their respective granularities, load distributions, and synchronization points. These algorithms are validated in section 4. Finally, results are analyzed using a test set of graphs of varying complexity.

2. PARALLELIZATION OF THE CYCLICAL CONJUNCTION ALGORITHM

The cyclical conjunction operator allows error-free extraction of all cycles from a graph, at acceptable computational cost. When graphs are complex, however, computing time rises to an unacceptable level for systems requiring a short response time.

The sequential cyclical conjunction algorithm[33] is an iterative algorithm which at each step extracts a set of cycles by performing cyclical conjunction between cycle pairs belonging to an initial cycles list (***ICL***) and the set of cycles obtained from the previous iteration of the algorithm (***PCL***).

Cyclical conjunction is implemented through an iterative process where three processing stages follow a preprocessing stage:[33]

Preprocessing: Beginning with the adjacency matrix designed to represent a graph, a set of initial cycles whose cardinality is equal to the cyclomatic complexity is obtained.[34] This cycle set is stored in a data structure called the *Initial Cycles List* (***ICL***).

Processing: Once the initial cycles have been obtained, we can calculate all cycles in the graph by applying cyclical conjunction to this set (***ICL***) and to the new sets of cycles generated by this function. This is performed by a three-stage iterative process and terminates when no new cycles are left to be discovered or when the maximum possible number of cycles has been extracted from the problem graph (2^{η}-1).

*Stage 1***:** Cycles comprising the ***ICL*** are compared with the cycle set obtained from the previous iteration of the algorithm (the first comparison will be made with the ***ICL*** itself). This cycle set is then stored in the *Previous Cycles List* (***PCL***). At this point the properties of the cyclical conjunction operator are applied and intersection is performed between pairs of cycles belonging to the ***ICL*** and ***PCL*** sets, respectively. The second stage will only come into play if: $R \cap T \equiv Z$, so that $R \in LCI$ and $T \in LCP$.

Stage 2: The sets *R* and *T* are sorted according to the order of *Z*. This process simply consists of updating the attributes of the elements of ***ICL*** and ***PCL*** where the cycles *R* and *T* are included.

Stage 3: Cyclical conjunction is performed for $R \oplus T$, thus obtaining a cycle *C*. A check is made to determine whether this cycle has been previously generated in the *New Cycles List* (***NCL***)—the cycle set generated by each iteration of the algorithm—and in the *Total Cycles List* (***TCL***), which contains all the graph cycles hereto detected. Whenever a new cycle is detected, it is inserted into the ***NCL***. On completing the iteration of the ***NCL*** cycles, these are placed into the ***TCL***, and the ***NCL*** is used as the ***PCL*** for the next iteration.

The computational cost of this algorithm is mainly due to the following:[33]

1. Preprocessing cost: used to obtain the initial cycle set (***ICL***).[34] The cost of this algorithm is comparable to that of others listed in the literature and at worst will correspond to *N* (the number of nodes in the graph).

2. Cost of stages 1 and 2 of the algorithm: here, the structures representing the cycles are prepared for the cyclical conjunction operator to check for any possible new cycles.

3. Cost of stage 3 of the algorithm: where cyclical conjunction is applied to pairs of cycles and a check is made to determine whether any new cycle produced already exists.

The data structures designed[33] to represent the total cycles of a graph and the cycles detected at each iteration of the algorithm considerably reduce the cost of processing each stage of the algorithm (stages 1 to 3), due to the efficient application of the properties of the cyclical conjunction operator, hence the number of invalid cyclical conjunction operations (those which do not give rise to a new cycle) is substantially reduced.

Notwithstanding, computational cost will rise with graph complexity—and the ensuing increase in the number of cycles therein—since increased complexity will mean a higher number of both cyclical conjunction operations and iterations required to extract all the cycles.

The need becomes apparent, then, for the design and development of parallel algorithms to take advantage of the multiprocessing capacities of many modern computer systems, with a view to cutting down the time involved in calculating the cycles in complex graphs.

The first step toward the design of a parallel algorithm consists of analyzing the problem and how the data access model and the computational power required might influence the desired parallel algorithm.

Fox[35] was the first to study this aspect of the parallelization process, setting out three categories or architectures within which the problem could be framed, associating each category with different types of parallel computers, and in a more recent study, Pancake[36] added a fourth category:

1. Perfect or job-level parallelism: when the program can be executed on independent machines running individual copies of the application and working with completely independent data.

2. Pipeline parallelism: when the application (the problem or algorithm to be parallelized, which in the present study is the algorithm for the extraction of all cycles from a graph, using the cyclical conjunction operator) may be divided into a series of independent stages, in which the simulation of the next step depends only on the results of the previous stage.

3. Fully synchronous parallelism: in many applications results cannot be restricted to one single direction for the calculation of each phase, rather all operations must be performed on all data in order to obtain the intermediate solutions required for moving on to the next iteration.

4. Loosely synchronous parallelism: where in applications such as those described above, all operations must be performed on all data to obtain intermediate solutions, but with the added problem of having an uneven and unpredictable data set, one which varies from iteration to iteration.

The proposed sequential algorithm[33] for the calculation of all cycles in a graph clearly falls into the fourth category (*loosely synchronous parallelism*), since cyclical conjunction is performed between a known set of data (the initial cycles) and a data set of uneven and unknown size that is calculated at a prior stage. This kind of loosely synchronous application is the most difficult to deal with when designing a parallel algorithm, and good results can only be expected when reducing the number of interaction and synchronization points between the processing units running the application.

It is clearly necessary to carry out a theoretical analysis of the expected best results achievable using parallel implementation, while also determining the optimal resources (number of processors) required. For this application is difficult to carry out a theoretical analysis of the system as function of the graph characteristics, the computational complexity is also dependent on factors such as total number of cycles and graph topology. Amdahl[37] produced a law which sought to determine the extent of improvement that might be attained through the use of parallelism.

Estimates are based on the time of the basic sequential program and determine which parts (*P*) are potentially suitable for parallelization without taking input/output operations into account. This value may be calculated using the following expression:

$$\boldsymbol{P} = \frac{\text{time for the potentially parallel part of the algorithm}}{\text{total algorithm time}} \tag{1}$$

Amdahl's Law allows us to calculate the speedup that may be achieved by the potentially parallelizable part of the application with respect to the number of processing units employed (*n*). Theoretical Amdahl speedup (S_{Ta}) may be calculated using the following expression:

$$S_{Ta} = \frac{1}{(1 - P) + \dfrac{P}{n}} \tag{2}$$

Amdahl's Law assumes a fixed workload or fixed problem size program. Otherwise, *Gustafson's Law*[38] assumes that parallel work scales with the problem size while the serial works remains constant. Theoretical Gustafson speedup may be calculate using the following equation

$$S_{Tg} = n + (\alpha \times (1 - n)) \tag{3}$$

where α is the fraction of "nonparallelizable" code, $\alpha = 1 - P$.

Table 1. Theoretical Speedups for Parallelization of the Application

CPUs number	1	2	3	4	5	6	7	8	9	10	...	∞
theoretical speedup (Amdahl's law)	1.00	1.86	2.60	3.26	3.84	4.27	4.82	5.24	5.62	5.97	...	13.00
theoretical speedup (Gustafson's law)	1.00	1.92	2.85	3.77	4.70	5.62	6.55	7.47	8.40	9.32	...	∞

In the present application, P has a value of 0.925 (for a fixed medium problem), meaning that 92.5% is potentially parallelizable, whereas 7.5% has to be sequential and would not be worth the effort of parallelizing. What is unparallelizable is the preprocessing algorithm, which deals with gathering the initial cycle set, while the potentially parallelizable part corresponds to the processing stage, where iterative cyclical conjunction operations are carried out on cycles recognized in the graph.[15]

Table 1 shows the theoretical speedup values for the application using both lows. It may be seen that even when an unlimited number of processors is used, speedup never rises above 13 according to Amdahl's Law. However according to Gustafson, the speedup should have an ideal behavior when unlimited processors are used.

Considering the penalizations (not taking into account in the theoretical analysis) of the parallel algorithm, it is especially due to synchronizm steps and the irregular characteristic of the problem. In the present study, a criterion of five processors was established as the maximum computational resource to be used to achieve an acceptable speedup over the sequential algorithm.

3. DEVELOPMENT OF THE PARALLEL ALGORITHMS

As described above, our application may be classed as loosely synchronous and two degrees of parallelism may be established:

•A thicker-grain parallelism where for each iteration the cyclical conjunction operations are distributed across the initial cycles list (***ICL***) and all the new sets of detected cycles.

•A finer-grain parallelism in which for each iteration and in each of the processing units cyclical conjunction of the entire initial cycles list is performed on a subset of the new list of cycles (***PCL***).

For each degree of parallelism, different parallelization options are presented for the sequential algorithm, depending on how the computational load is distributed across the processors and the point at which synchronization of the latter is carried out. Based uniquely on situations where parallelization would theoretically produce good results, two parallel algorithms for shared-memory parallel architectures were developed. The first of these, the *IDL* (*Initial Distributed List*), implements thicker-grain parallelism, while the second, the *CSDL* (*Current Synchronized Distributed List*), implements fine-grain parallelism.

In both proposed parallel algorithms, the initial transformations (preprocessing) are carried out sequentially, hence we shall focus on describing the parallel algorithms for performing the processing phase of the algorithm for calculating all the cycles in a graph.

3.1. The IDL Algorithm. As may be seen in the pseudocode shown in Chart 1, this algorithm comprises the following stages:

Chart 1. The *IDL* Parallel Algorithm

```
IDL Algorithm.
    /* Prepare the list of previous cycles for the first iteration */
assignlist(ICL, PCL);
    /* Initialize the list of new cycles */
NCL = NULL;
    /* Add the first saga to the TCL */
insertlist(ICL, TCL);
    /* Initialize terminator */
end = TRUE;
    /* Begin iteration loop */
While (end)
    /* Begin parallelization */
  PRAGMA #OMP PARALLEL /* For OpenMP */
    /* PRAGMA PARALLEL for SGI MP */
  PRAGMA OMP FOR Schedule (STATIC, DYNAMIC, GUIDED)
    /* PRAGMA PFOR Schedtype (SIMPLE, INTERLEAVE, DYNANMIC, GSS)
    for SGI MP */
    /* Walk all cycles of the ICL */
  For i = 0 to ICL.total_cycles
    /* Walk the cycles from the last saga obtained (PCL) */
    For j = 0 to PCL.total_cycles
    /* Check if operation will be effective */
      If property(ICL.cycle[i], PCL.cycle[j]) = FALSE
    /* Perform cyclical conjunction */
        cycle = cyclicalconjunction (ICL.cycle[i], PCL.cycle[j]);
    /* Check cycle and its presence in the NCL and TCL */
        If is_cycle(cycle)
          If search(NCL, cycle) = FALSE
    /* If it is a new cycle, insert in new list (NCL) */
            If search(TCL, cycle) = FALSE
              insert(NCL, cycle);
            EndIf
          EndIf
        EndIf
      EndIf
    EndFor
  EndFor
  END PARALLEL FOR
    /* End parallelization */
  END PRAGMA PARALLEL
    /* Duplicates cycles are deleted */
    deleteduplicates(NCL);
    /* On ending iteration, insert NCL into TCL, then initialize
    variables for next iteration.
    If no new cycles are created or end of total cycles is reached,
    terminate the process */
  If (NCL != NULL)
    insertlist(NCL, TCL);
    assignlist(NCL, PCL);
    NCL = null;
  EndIf
  Else
    end = FALSE;
  EndElse
EndWhile.
End IDL Algorithm.
```

1. Each processor walks a subset of initial cycles (***ICL***) using the parallel loop represented by the variable i. The cardinality of this subset is determined by the number of processors and the data distribution employed (four data distributions were evaluated for the proposed algorithms; these are described later in this section).

2. The cyclical conjunction operator is applied to all the cycles extracted from the previous step (***PCL***).

3. The steps employed to carry out conjunction are similar to those of the sequential algorithm[33] but with the following differences:

(a) A local list is created, where each processor stores the cycles it detects. Checking to eliminate repeated cycles is performed by each processor, working exclusively with its own local list.

(b) Once the cyclical conjunction operations have been performed between all the initial cycles (***ICL***) and all those in the current list (***PCL***), and prior to deeming the new cycles as valid, any repeated cycles are erased, taking every cycle detected by each processor into account.

3.2. The CSDL Algorithm. The *IDL* algorithm ensures data coherence by checking, on completion of each iteration, whether any detected cycles are repeated, but without setting synchronism points that would penalize parallel execution. Nevertheless, carrying out such checks may lead to the generation of repeated cycles that will in turn involve additional cyclical conjunction operations. It would thus seem convenient to set a synchronism point to ensure data coherence before each iteration is completed, even though

Chart 2. The *CSDL* Parallel Algorithm

```
CSDL Algorithm.
assignlist(ICL, PCL);
NCL = NULL;
insertlist(ICL, TCL);
end = TRUE;
While (end)
  For i = 0 to ICL.total_cycles
    /* Begin parallelization */
    PRAGMA OMP PARALLEL /* For OpenMP */
    /* PRAGMA PARALLEL for SGI MP */
    PRAGMA OMP FOR Schedule (STATIC, DYNAMIC, GUIDED)
    /* PRAGMA PFOR Schedtype (SIMPLE, INTERLEAVE, DYNANMIC, GSS)
    for SGI MP */
    For j = 0 to PCL.total_cycles
      If property(ICL.cycle[i], PCL.cycle[j]) = FALSE
        cycle = cyclicalconjunction (ICL.cycle[i], PCL.cycle[j]);
        If is_cycle(cycle)
          If search(NCL, cycle) = FALSE
            If search(TCL, cycle) = FALSE
              insert(NCL, cycle);
            EndIf
          EndIf
        EndIf
      EndIf
    EndFor
    PRAGMA OMP BARRIER /* For OpenMP */
    /* PRAGMA SYNCRONICE for SGI MP */
      deleteduplicates(NCL);
    END OMP BARRIER /* For OpenMP */
    /* END PRAGMA SYNCRONICE for SGI MP */
    END PARALLEL FOR
    /* End parallelization */
    END PRAGMA PARALLEL
  EndFor
  If (NCL != NULL)
    insertlist(NCL, TCL);
    assignlist(NCL, PCL);
    NCL = null;
  EndIf
  Else
    end = FALSE;
  EndElse
EndWhile.
End CSDL Algorithm.
```

this would slow the parallel algorithm. To this end, the *CSDL* parallel algorithm was designed, as shown in Chart 2.

The first two stages of this algorithm are similar to those of the *IDL* algorithm. The third stage involves inserting a synchronizm phase into each iteration, to remove repeated cycles. This phase occurs on processing the cyclical conjunction of a cycle from the initial cycles list (***ICL***) with the subset of cycles in the current list (***PCL***) assigned to each processor according to the distribution employed.

The data structures have been designed to allow that each processor (or thread), depending of the process identifier (pid), carries out the ***NCL*** modifications in different memory areas, allowing this way the parallel processing in this stage. Processor synchronization preserves data coherence while also decreasing the number of repeated cycles passed on to the next iteration, though the parallel algorithm will clearly be penalized.

3.3. Data Position, Coherence, and Load Balancing. Given that the present application is loosely synchronous, a shared-memory parallel architecture would seem most appropriate, and algorithms were designed accordingly.

As memory is shared by all processors, data position is assured by the machine itself. Extreme caution must be exercised with respect to data coherence, however, since one processor might want to read data while another was modifying it, and the value of that data would thus be invalid.

Coherence is, for the most part, guaranteed by the memory-sharing architecture of the machines. This approach involves wait states however, meaning that on many occasions the parallel and sequential algorithms will exhibit similar behavior. In the present study, data coherence played a major role in the design of the algorithm, particularly when repeated cycles are excluded during cyclical conjunction.

The need to synchronize and update data in certain parts of the algorithm calls for careful load-balancing at runtime. Accordingly, four different schedule techniques for load distribution were evaluated:

Simple distribution, where iterations of the parallel loop on the data set (list of cycles) are evenly divided into a number of chunks equal to the number of processors, and each chunk is assigned to its own processor. This distribution is only available in SGI MP library.

Interleaved or Static distribution, in which iterations are divided into chunks. The chunks are statically assigned to processors in the team in a round-robin fashion in the order of the processor or thread number. This distribution is available in SGI MP and OpenMP libraries.

Dynamic distribution, which splits up the iterations set in the same way as interleaved distribution, but assigns chunks according to the demands of each processor (available in SGI MP and OpenMP libraries).

Guided Self-Scheduling (*GSS*), similar to the dynamic technique, but where the size of the chunks is not constant; at first large chunks are distributed, followed by smaller ones (available in SGI MP and OpenMP libraries).

4. VALIDATING THE PARALLEL ALGORITHMS

The efficiency of the proposed parallel algorithms was tested using the Silicon Origin 2000 shared memory multiprocessor equipped with eight processors. The algorithms were programmed in the C language, using the MP[39,40] and OpenMP[41,42] library set for parallel programming.

Validation tests were performed on a set of graphs of varying complexity as shown in Table 2. Validation of the parallel code was done by comparing the results for the time

Table 2. Experimental Results for the Sequential Algorithm for the Test Set of Graphs[a]

	characteristics			results						
graphs	nodes	edges	η	ICN	TCN	O_T	O_P	time (s)	R	E
G-08x28	8	28	21	21	8018	168147	20370	29.68	88	0.39
G-09x30	9	30	22	22	9975	219197	26566	58.74	88	0.37
G-60x76	60	76	17	17	24635	418642	87771	303.28	79	0.28
G-09x33	9	33	25	25	27588	689375	68974	421.02	90	0.40
G-13x33	13	33	21	21	33843	710472	75359	631.16	89	0.45
G-09x36	9	36	28	28	62814	1758386	164136	3064.81	91	0.38
G-10x38	10	38	29	29	76514	2218471	191527	5201.72	91	0.40
G-10x40	10	40	31	31	145014	4494938	375459	17927.67	91	0.39
G-14 39	14	39	26	26	165471	4301895	387423	24341.08	91	0.43
G-12x40	12	40	29	29	194614	5643371	409190	28558.63	93	0.48
G-14x40	14	40	27	27	217338	5867748	467265	41646.35	92	0.47
G-12x41	12	41	30	30	262184	7865055	575390	62303.45	93	0.46

[a] η: cyclomatic complexity, *ICN*: initial cycles number, *TCN*: total cycles number, O_T: theoretical operations, O_P: real operations, *R*: reduction factor, *E*: effectiveness.

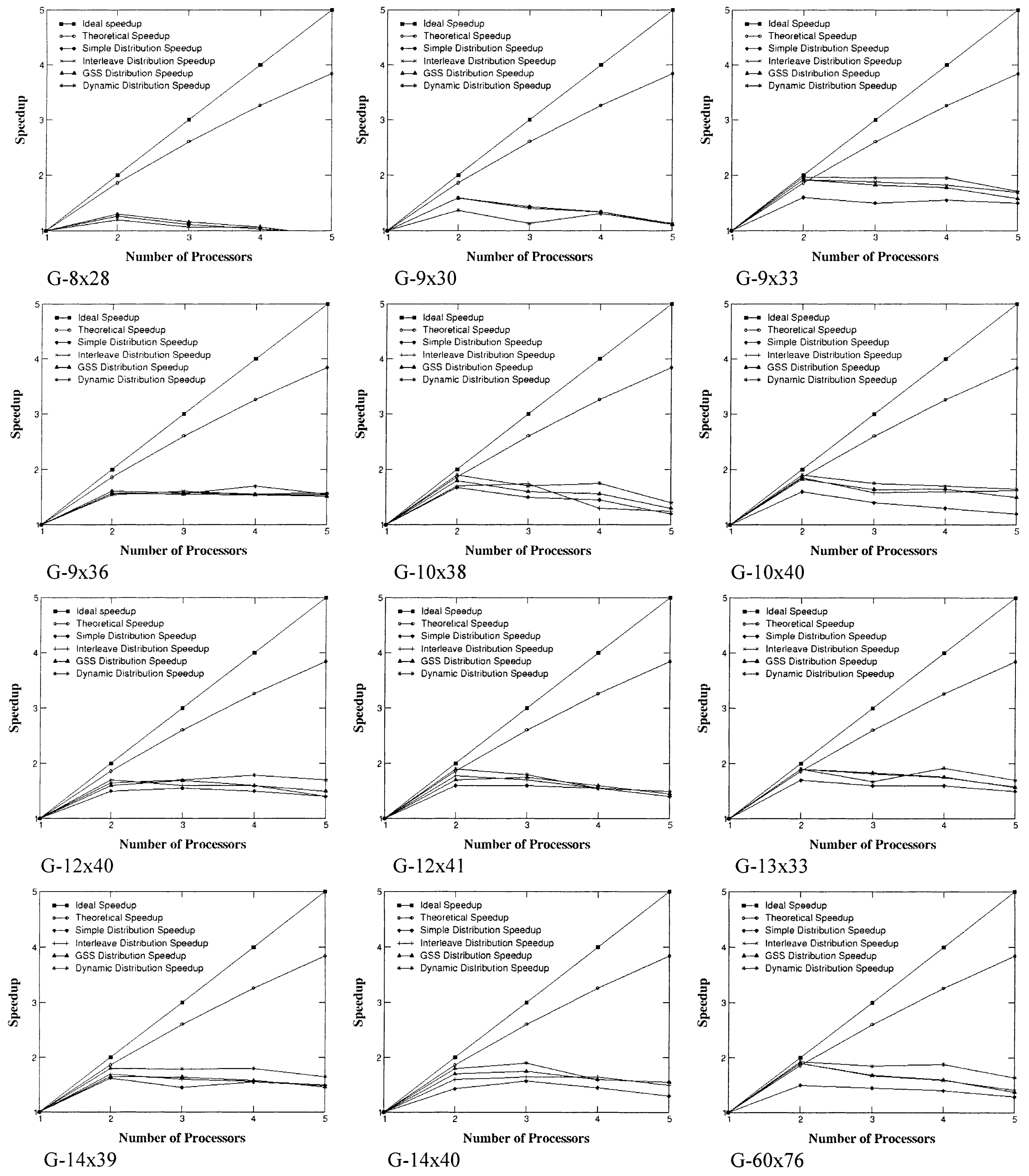

Figure 2. Behavior of the *IDL* algorithm for different load distributions.

taken to run the two proposed algorithms for the four data distributions employed, using from one to five processors. Figures 2 and 3 show results for the *IDL* and *CSDL* parallel algorithms, respectively.

Table 3 shows the experimental results for both algorithms compared with those obtained using the sequential algorithm. The following may be seen:

•As it was verified for the sequential algorithm,[33] tests showed that calculation time for the extraction of graph cycles is dependent on factors such as total number of cycles and graph topology and complexity, in addition to the nature of the initial cycles detected at the preprocessing phase—factors that are either unknown or excessively costly to determine a priori. This effect can be observed in the graphs G-10x38 and G-10x40, which having a number of nodes, edges, and cyclomatic complexity very close contain a very different total number of cycles which gives rise to a great difference in the time spend by the algorithm.

•In every instance, the *IDL* parallel algorithm only improved performance when two processors were used

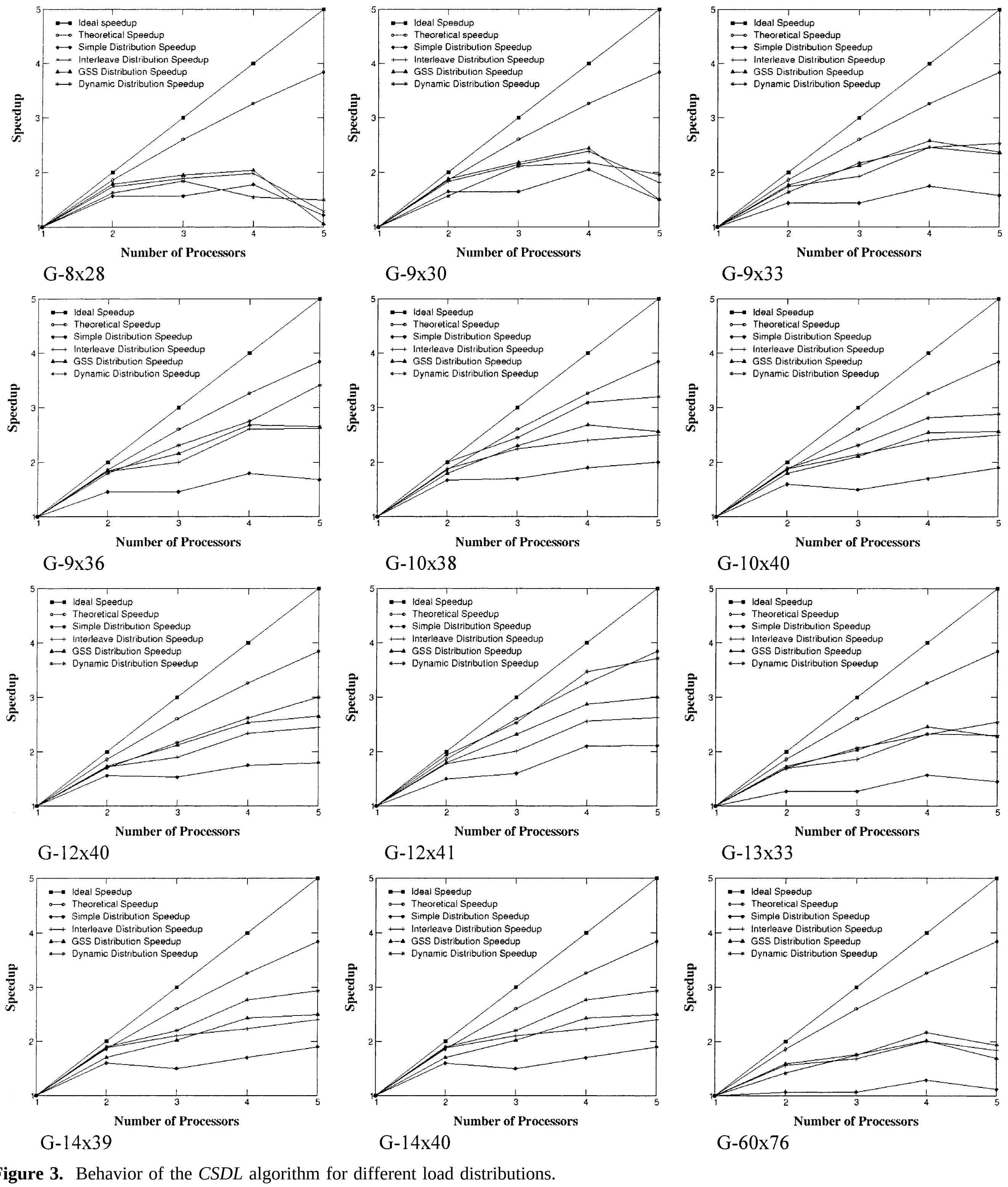

Figure 3. Behavior of the *CSDL* algorithm for different load distributions.

(Figure 2). On increasing the number of processors, this algorithm achieved, at best, the same results as the sequential algorithm, irrespective of load distribution.

•No improvement in results was observed on altering data distribution for the *IDL* algorithm. In this study it was seen that at this level of thicker-grain parallelism the load distribution mechanisms (*interleave, dynamic, GSS*) used to compensate for computational imbalance did not work satisfactorily.

•On employing the *CSDL* algorithm significant acceleration was recorded (Figure 3). Differences observed relative to the ideal and theoretical expectations are due to the overhead produced by the synchronization barrier at the end of the parallel loop and penalizations of the mechanisms to guarantee the load balance.

•For the *CSDL* algorithm it was observed that as computational load (graph complexity) rose, so too did the acceleration provided by the parallel algorithm (Figure 3). Again, this finding shows good scalability of the algorithm as computational load increases and improved use of cached memory by the parallel algorithm.

Table 3. Variation in Speedup According to Number of Processors for the *CSDL* Algorithm with Dynamic Distribution Using SGI and OpenMP Libraries

		time (s) and speedup for SGI MP					time (s) and speedup for OpenMP				
graphs	sequential	1	2	3	4	5	1	2	3	4	5
G-08x28	29.68	39.4	19.53	16.85	19.15	19.79	39.31	17.95	15.87	18.88	19.67
		0.75	**1.52**	**1.76**	**1.55**	**1.50**	**0.68**	**1.63**	**1.85**	**1.55**	**1.48**
G-09x30	58.74	58.88	39.25	27.64	27.01	30.99	58.98	37.45	27.93	27.03	30.09
		0.99	**1.50**	**2.13**	**2.18**	**1.90**	**0.99**	**1.55**	**2.11**	**2.18**	**1.96**
G-60x76	303.28	303.2	216.57	170.34	132.98	155.48	308.31	215.76	176.08	141.49	158.29
		1	**1.40**	**1.78**	**2.28**	**1.95**	**0.98**	**1.43**	**1.75**	**2.18**	**1.95**
G-09x33	421.02	432.56	262.15	201.19	180.23	172.33	435.14	264.06	200.39	176.37	171.61
		0.97	**1.65**	**2.15**	**2.40**	**2.51**	**0.97**	**1.65**	**2.17**	**2.47**	**2.54**
G-13x33	631.16	633.28	376.95	291.83	276.54	252.3	635.49	372.83	306.30	273.59	249.76
		0.99	**1.68**	**2.17**	**2.29**	**2.51**	**0.99**	**1.70**	**2.07**	**2.32**	**2.54**
G-09x36	3064.81	3167.25	1666.97	1347.76	1139.29	915.39	3145.35	1748.26	1321.84	1147.64	920.37
		0.97	**1.90**	**2.35**	**2.78**	**3.46**	**0.97**	**1.80**	**2.38**	**2.74**	**3.42**
G-10x38	5201.72	5305.42	2422.56	2192.32	1684.26	1632.43	5307.88	2527.83	2163.56	1714.35	1657.03
		0.98	**2.18**	**2.42**	**3.15**	**3.25**	**0.98**	**2.10**	**2.45**	**3.10**	**3.20**
G-10x40	17927.67	18093.13	9522.7	7832.52	6579.30	6196.27	18293.54	9639.18	7902.65	6506.79	6337.35
		0.99	**1.90**	**2.31**	**2.75**	**2.92**	**0.98**	**1.90**	**2.31**	**2.81**	**2.89**
G-14x39	24341.08	24596	12945.26	11180.2	8944.3	8337.62	24586.95	12940.44	11175.45	8856.15	8378.97
		0.99	**1.90**	**2.20**	**2.75**	**2.95**	**0.99**	**1.90**	**2.20**	**2.78**	**2.93**
G-12x40	28558.63	28847.08	16026.15	13232.6	10968.47	9583.74	28847.11	16792.62	13256.56	10986.35	9596.03
		0.99	**1.80**	**2.18**	**2.63**	**3.01**	**0.99**	**1.72**	**2.18**	**2.63**	**3.01**
G-14x40	41646.35	41698.54	20849.27	18288.83	13807.46	13030.79	41688.04	20844.10	18125.22	13896.61	12824.08
		0.99	**2.00**	**2.28**	**3.02**	**3.20**	**0.99**	**2.00**	**2.30**	**3.00**	**3.25**
G-12x41	62303.45	62331.14	32805.86	24636.81	17808.89	17069.44	62332.93	32118.01	24582.57	17958.40	16772.88
		1.00	**1.90**	**2.53**	**3.50**	**3.65**	**1.00**	**1.94**	**2.54**	**3.47**	**3.72**

•While in theory there are no significant differences between *dynamic* and *GSS* distributions, the nondeterministic behavior of the iteration set to be distributed, together with the extra burden of *GSS* distribution, may mean that *dynamic* distribution produces much better results, as indeed is the case in the present study. Thus, given the uneven nature of the application and viewing the experimental results, the appropriate distribution to employ will be *dynamic*, since despite the performance hit involved in its use, *dynamic* distribution achieved the most impressive acceleration.

DISCUSSION

On determining that the best results were obtained with the *CSDL* algorithm and *dynamic* distribution, this algorithm was applied to the test set of complex graphs, and results were compared to those obtained when applying the sequential algorithm to the same graphs.

Table 3 shows the experimental results using MP and OpenMP libraries, while Figure 4 shows the variation in speedup according to the number of processors.

•For one processor it may be seen that the ratio of the times taken by the two algorithms is close to unity as graph complexity rises. Only for graphs of little complexity (G-08x28 or less) does it fall below unity, since here the increased complexity of the parallel algorithms is simply not warranted.

•Speedup does not alter linearly with the number of processors, which would have been ideal behavior. This deviation is clear in low-complexity graphs, showing that a large number of processors would be a waste of computational resources in this instance. Reasons for this behavior are manifold: the nature of the graph, the number of nodes and edges, and the number and nature of the cycles present. In other words, the lack of graph complexity might mean that it is not worth dedicating extensive computer resources for their solution. Some of these factors also affect the sequential cyclical conjunction algorithm and thus its parallel implementation.

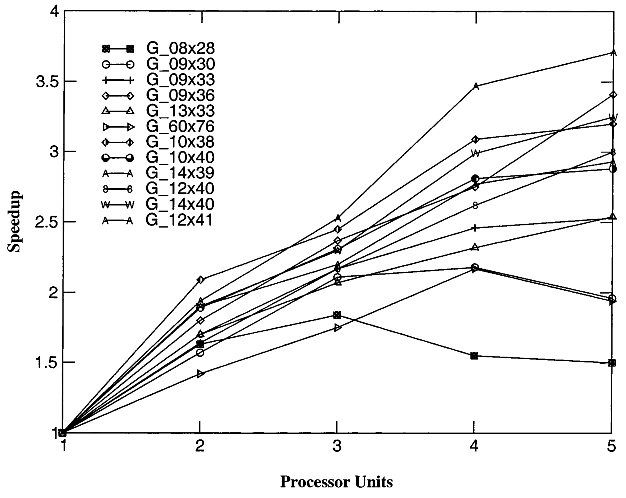

Figure 4. Variation in speedup according to the number of processors with the *CSDL* algorithm and *dynamic* distribution.

•In every instance the *CSDL* algorithm considerably reduced processing time with respect to the sequential algorithm, although the following observations are worthy of note:

▪For simple graphs the time taken by the sequential algorithm is so low that there is no need for parallelization.

▪For graphs of medium complexity, the *CSDL* algorithm processing times make it suitable for an interactive process, whereas the sequential algorithm would be too slow.

▪For highly complex graphs, although computing time is orders of magnitude faster than for the sequential algorithm, it is still perhaps impracticably high for use in interactive systems.

•They are not significant time differences using MP or OpenMP libaries. However, we suggest the utilization of

OpenMP, given that this standard for shared memory parallel algorithms implementation is available on many other platforms.

In conclusion, experimental results show both algorithms to be suitable for the extraction of cycles from a graph. When graphs are simple (up to thousands of cycles), the sequential method is acceptably efficient, but as graph complexity rises, the proposed *dynamic-CSDL* algorithm becomes necessary.

REFERENCES AND NOTES

(1) *Parallel Computing in Computational Chemistry*; Mattson, T. G., Ed.; ACS Symposium Series 592; American Chemical Society: Washington, DC, 1995.

(2) Palace Carvalho, A.; Gomes, J. A. N. F.; Cordeiro, N. D. S. Parallel implementation of a Monte Carlo Molecular Simulation Program. *J. Chem. Inf. Comput. Sci*. **2000**, *40*(3), 588−592.

(3) Straatsma, T. P.; Philippopoulos, M.; McCammon, J. A. NWChem: Exploiting parallelism in molecular simulations. *Comput. Physics Commun.* **2000**, *128*, 377−385.

(4) Roccatano, D.; Bizzarri, R.; Chilemi, G.; Sanna, N.; Di Nola, A. Development of a parallel molecular dynamics code on SIMD computers: an algorithm for the use of the pair list criterium. *J. Comput. Chem.* **1998**, *19*, 685−691.

(5) Trobec, R.; Merzel, F.; Janezic, D. The complexity of parallel symplectic molecular dynamics algorithms. *J. Chem. Inf. Comput. Sci.* **1997**, *37*(6), 1055−1062.

(6) Steinbeck, C. SENECA: A Platform-Independent, Distributed, and Parallel System for Computer-Assisted Structure Elucidation in Organic Chemistry. *J. Chem. Inf. Comput. Sci.* **2000**, *41*(6), 1500−1507.

(7) del Carpio, C. A. A Parallel genetic algorithm for polypeptide three-dimensional srtructure prediction. A transputer implementation. *J. Chem. Inf. Comput. Sci.* **1996**, *36*(2), 258−269.

(8) Fruchtl, H. A.; Kendall, R. A.; Harrison, R. J.; Dyall, K. G. An implementation of RI-SCF on parallel computers. *Intl. J. Quantum Chem.* **1997**, *64*(1), 63−69.

(9) Rasmussen, E. M.; Downs, G. M.; Willet, P. Automatic classification of chemical structure databases using a high parallel array processor. *J. Computat. Chem.* **1988**, *9*, 378−386.

(10) Gillet, V. J.; Welford, S. N.; Lynch, M. F.; Willet, P.; Barnard, J. M.; Downs, G. M. Computer storage and retrieval of generic chemical structures in patents. VII. Parallel simulation of a relaxation algorihm fro chemical substructure search. *J. Chem. Inf. Comput. Sci*. **1986**, *26*, 118−126.

(11) Mauri, G.; Pavesi, G. A parallel algorithm fro pattern discovery in biological sequences. *Future Generation Computer Systems* **2002**, *18*, 849−854.

(12) Lee, J.; Pillardy, J. et al. Efficient parallel algorithms in global optimization of potential energy functions for peptides, proteins and crystals. *Comput. Physics Commun.* **2000**, *128*, 399−411.

(13) MACCS Keys. MDL Information Systems, Inc.: 14600 Catalina Street, San Leandro CA94577.

(14) Rücker, G.; Rücker, C. Topogical Indices, Boiling Points and Cycloalkanes. *J. Chem. Inf. Comput. Sci.* **1999**, *39*, 788−802.

(15) Ravi, M.; Hopfinger, A. J.; Hormann, R. E.; Dinan, L. 4D-QSAR Analysis of a set of ecdysteroids and a comparison to CoMFA modeling. *J. Chem. Inf. Comput. Sci.* **2001**, *41*(6), 1587−1604.

(16) McGregor, M. J.; Pallai, P. V. Clustering of large databases of compounds: using MDL keys as structural descriptors. *J. Chem. Inf. Comput. Sci*. **1997**, *37*, 443−448.

(17) Xue, L.; Bajorath, J. Accurate partitioning of compounds belonging to diverse activity classes. *J. Chem. Inf. Comput. Sci*. **2002**, *42*(3), 757−764.

(18) Lipkus, A. H. Mining a Large database fro Peptidomimetic Ring Structures Using Topological Index. *J. Chem. Inf. Comput. Sci.* **1999**, *39*(3), 582−586.

(19) Lipkus, A. H. Exploring chemical rings in a simple topological-descriptor space. *J. Chem. Inf. Comput. Sci*. **2001**, *41*(2), 430−438.

(20) Flower, D. R. On the properties of bit string-based measures of chemical similarity. *J. Chem. Inf. Comput. Sci*. **1998**, *38*, 379−386.

(21) Godden, J.; Xae, L.; Bajorath, J. Combinatorial preferences affect molecular similarity/diversity calculations using binary fingerprints and Tanimoto coefficients. *J. Chem. Inf. Comput. Sci*. **2000**, *40*, 163−166.

(22) Bollobás, B.; Fenner, T. I.; Frieze, A. M. An Algorithm for Finding Hamilton Paths and Cycles in Random Graphs. *Combinatoria* **1987**, *7*(4), 327−341.

(23) Frieze, A. M. Parallel Algorithms for Finding Hamilton Cycles in Random Graphs. *Inform. Process. Lett.* **1987**, *25*, 111−117.

(24) MacKenzie, P. D.; Stout, Q. F. Optimal Parallel Construction of Hamiltonian Cycles and Spanning Trees in Random Graphs. *ACM-SPAA Velen Germany* **1993**, 224−229.

(25) Thomason, A. A Simple Linear Expected Time Algorithm for the Hamilton Cycle Problem. *Discrete Math.* **1989**, *75*, 373−379.

(26) Edmonds, J. Optimum Branching. *J. Res. Nat. Bur. Stand.* **1967**, *71*, 233−240.

(27) Nepomniaschaya, A. S. An Associative Parallel Algorithm for Finding a Critical Cycle in Directed Graphs. *7th International Conference on Parallel and Distributed Systems (ICPADS'00)*, 2000.

(28) Boukerche, A.; Tropper, C. A Distributed Graph Algorithm for the Detection of Local Cycles and Knots. *IEEE Trans. Parallel Distributed Systems* **1998**, *9*(8), 748−757.

(29) Steinbeck, C. SENECA: A Platform-Independent, Distributed, and Parallel System for Computer-Assisted Structure Elucidation in Organic Chemistry. *J. Chem. Inf. Comput. Sci.*, Published on Web, 2000.

(30) Chua, L. O.; Chen, L. K. On Optimally Sparce Cycle and Coboundary Basis for a Linear Graph. *IEEE Trans. Circuit Theory* **1973**, *20*, 495−503.

(31) Downs, G. M.; Gillet, V. J.; Holliday, J. D.; Lynch, M. F. Theoretical Aspects of Ring Perception and Development of the Extended Set of Smallest Rings Concept. *American Chemical Society* **1988**, *29*(3), 187−206.

(32) Knuth, D. E. The Art of *Computer Programing*, *1*; Addinson-Wesley: 1968; pp 363−370.

(33) Cerruela García, G.; Luque Ruiz, I.; Gómez-Nieto, M. A. Cyclical Conjunction: an Efficient Operator for the Extraction of Cycles from a Graph. *J. Chem. Inf. Comput. Sci.* **2002**, *42*, 1415−1424.

(34) Cerruela García, G.; Luque Ruiz, I.; Gómez-Nieto, M. A. Un algoritmo en zigzag para la obtención de un conjunto de ciclos característicos de un grafo; Internal Report UCO-ISCBD-JCVG93-00; University of Córdoba: 2000.

(35) Fox, G. C.; et. al. *Solving Problems on Comcurrent Proccesors*; Prentice-Hall: Englewood Cliffs, NJ, 1988.

(36) Pancake, C. M. Is Parallelism for You. *IEEE Comput. Sci. Eng.* **1996**, 18−37.

(37) Amdahl, G. Validity of the Single-Processor Approach to Achiving Large-Scale Computing Capabilities. *Proc. AFIPS Conf*. **1967**.

(38) Gustafson, J. L. *Reevaluating Amdahl's Law. Comm. ACM* **1988**, *31*, 5.

(39) Silicon Graphics. *The Iris Power C. User's Guide* (007-0702-030), 1999.

(40) Silicon Graphics. *MIPSpro Power C Manual Kit* (M4-PWRC-7.1), 1999.

(41) *OpenMP C and C++ Aplication Program Interface*. Copyright of OpenMP Architecture Review Board, March 2002.

(42) Chandra, R.; Menon, R.; Dagum, L.; Kohr, D.; Maydan, D.; McDonald, J. *Parallel Programming in OpenMP*; Morgan Kaufmann: 2000.

CI020012Z