Rapid Subgraph Search Using Parallelism

W. TODD WIPKE* and DAVID ROGERS

Department of Chemistry, University of California, Santa Cruz, California 95064

Received May 8, 1984

A new parallel processing algorithm is reported for subgraph matching. Parallelism is achieved for the first time within the process of node-by-node matching of two individual graphs. A SIMULA program is described for simulating this parallel subgraph search algorithm. Simulation results from a series of chemical substructure search problems show an average utilization of 84% on a 25-processor machine and up to a 24-fold speed enhancement over a single processor. Potential applications include starting material selections for synthesis as well as general substructure search problems.

INTRODUCTION

Subgraph matching is an important method used in many different computer applications in organic chemistry, including the recognition of functional groups,[1] synthesis planning,[2] metabolism prediction,[3] constraint testing in structure generation,[4,5] selection of starting materials for synthesis,[6] and structure-oriented retrieval exemplified by COUSIN,[7] CAS ONLINE,[8] and MACCS.[9,10] The fundamental problem is, given a *query substructure* (QS) and a *candidate superstructure* (CS), determine if there exists a mapping of the atoms (nodes) of the substructure onto the candidate superstructure such that connected atom pairs in the query substructure are also connected in the candidate superstructure and that atom types and bond types also correspond. Extensions of this problem include searching over a library of candidate superstructures and allowing ambiguity in the query substructure (e.g., generic queries).

Although substructure searching is a nonnumerical problem, it is computationally demanding because it ultimately involves establishing an atom-by-atom correspondence between the QS and the CS, and this problem is a member of the class of NP-complete problems.[11,12] In a worst case for N atoms in the QS and M atoms in the CS ($M > N$), one may have to consider $N!/(M-N)!$ mappings for each CS. Some methods exist for reducing the computational effort of substructure searching as summarized later in this paper. Our objective in this work was not to reduce computational effort but to reduce the time required to perform substructure matching by applying parallel processing to this problem.

Parallel processing is widely used in many areas of numerical chemistry. Array processors that optimize floating-point matrix computations are common attachments to minicomputers today. Compiliers now exist that can capitalize on parallelism in the arithmetic. These techniques are applied in a wide range of numerical chemical computations, including chemical kinetics, molecular mechanics, and ab initio quantum calculations.[13] However, these tools have not been useful in nonnumerical chemical computations where the combinatorial explosion of simple nonnumerical operations is the real difficulty. Data structure and procedural complexities of many symbolic computations further complicate efforts to achieve parallelism. Subgraph matching is an example of a nonnumerical problem that has not been influenced greatly by the advent of parallel machinery.

SUBSTRUCTURE SEARCH

We shall use the term *substructure search* to mean the process of finding all structures in a database for which a given query structure is a subgraph. Most current substructure search systems utilize the following three-step process.

(1) A key mask search is performed to eliminate consideration of any structure in the database that cannot have the desired relationship to the query. Keys are features in the query structure that must also be in the candidate structure; all structures passing the key search must then be considered individually in steps 2 and 3.

(2) A partitioning procedure is used on an individual candidate to eliminate potential correspondences between candidate atoms and subgraph atoms.[14] For example, a carbon atom could not correspond to a nitrogen atom in the subgraph, and similarly, a primary atom could not correspond to a secondary atom in the subgraph.

(3) A permutation procedure then tests the remaining possible correspondences. A backtracking algorithm is used in this case; upon reaching a dead-end, the search returns to its state at the last choice point, i.e. the point where more than one assignment of an atom was possible.

Parallelism is used by the CAS ONLINE search system. The large data file (over 6 million compounds) was divided into 10 sections. Each section is searched serially with standard methods by its own PDP-11.[15,16] This is parallelism at the data base level. As the file grows, more section:PDP-11 units can be added to maintain relatively constant search times.

Ullman[17] proposed a parallel asynchronous implementation of a partitioning procedure, though the hardware was never constructed, nor simulated. This was the first suggested application of parallelism to the subgraph isomorphism problem at the algorithm level.

Surprisingly, parallelism has not been previously applied to the third step, the permutation process, the most computationally demanding step. It is this permutation process that is NP complete. Computation time in this step rises rapidly as the size of the graphs being matched increases, and for certain queries (generic or very common fragment), screening efficiency of steps one and two is low, leaving a large number of structures for the time-consuming permutation process. Through our earlier work with SST (starting material selection strategies), we saw potential for applying parallelism to the permutation step.[6]

HARDWARE REQUIREMENTS

It will be useful before describing our algorithm to first describe the preferred hardware configuration. The basic hardware requirement is that each processor node must have considerable logical capabilities. Array processors, such as the commercially available Floating Point Systems, involve a number of special-purpose processors working in parallel, but the task assigned to each is usually a simple arithmetic operation such as addition or multiplication, and all of the processors are expected to run simultaneously, so that the results of all operations can be reported at once. The limited operations available for the task, along with the synchronicity requirement, made this architecture ill suited for our purpose.

An architecture that allows more complex operation at each processor is the Advanced Flexible Processor, which contains

0095-2338/84/1624-0255$01.50/0

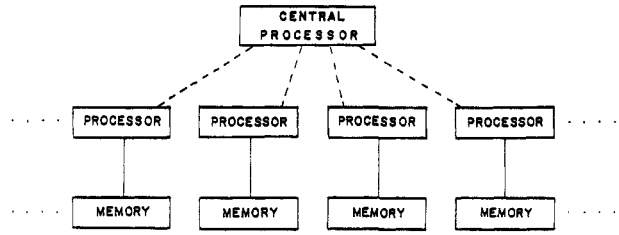

Figure 1. Proposed architecture for a parallel processing machine.

up to 16 processors in a bidirectional ring network.[18] This system offers very fast processing capability, but the ring architecture forces processors to devote some of their computational time to passing messages. We wished to avoid the costs of this communication scheme and the problems with synchronizing the processors.

We prefer the "star" configuration shown in Figure 1. The central processor has a communication line to each lower processor, but the lower processors have no direct communication lines between them. The lower processors are general-purpose asynchronous processors, each with a small local memory as a private "working space", which avoids problems inherent in shared memory. The central processor is different only in that it is the master that communicates with the system user and it may have more memory.

The star configuration is most useful when the number of processors is relatively small and the communication needs are also small. Compact communication packets minimize the storage and channel band width required at the central processor. Our algorithm requires no synchronization of the processors during the problem-solving computation. A configuration like this has been built for running SIMULA in parallel[19] and for simulating the CHiP computer.[20]

DESCRIPTION OF THE SUBSTRUCTURE SEARCH ALGORITHM

We approached the problem by eliminating backtracking in the permutation (atom-by-atom matching) process so the NP-complete[11] problem becomes order[12] $O(N)$, where N is the number of atoms in the substructure. This is easily done; the first atom of the substructure can be matched against any atom in the superstructure, so a task is created for each possibility and the problem has been divided into M subproblems, where M is the number of atoms in the superstructure. For each of these tasks that have one atom mapped, subtasks can be generated to map a second atom onto the superstructure, further subdividing the problem. If a task cannot map its assigned atom, it dies. A task successfully mapping the last node announces the completed mapping. This algorithm finds *all possible* substructure matches and *never backtracks*.

Unfortunately, the algorithm is now NP complete with respect to tasks! But the tasks are ideal for parallel computation. Each task can be completed independently of any other task, so we can freely allocate tasks among individual processors without any need for synchronization or communication between the tasks. We have not reduced the complexity of the algorithm, but we have established a basis for parallelism within the node-by-node matching process, and this will allow reducing elapsed execution time.

To study the likely performance, we have simulated the parallel machine by using typical chemical problems. A high-level description of the algorithm as we implemented it is shown in Figure 2. This algorithm is strictly the permutation algorithm that would be applied to the candidates passing the key mask and partition screening. Further, our goal was a simple description to highlight the emphasis on parallelism. There are many modifications that would improve the efficiency of the algorithm, but we ask the reader to focus

```
PARALLEL PROCEDURE $TryChoice (Node, Mapping, ChoiceSet);

{ This procedure takes a NODE in the subgraph to attempt to map along
  with a current MAPPING and set of possible assignments CHOICESET and
  generates all possible mappings for that node onto the supergraph.}

EXTERNAL INTEGER SubGrSize, SuperGrSize;
EXTERNAL SET ARRAY SubAdjacentSet[SubGrSize];
EXTERNAL SET ARRAY SupAdjacentSet[SuperGrSize];

INTEGER ARRAY Mapping[SubGraphSize]; SET ChoiceSet;
SET Choices, SupGrSet, SubGrSet, NewChoiceSet;
INTEGER Node, SupGrNode, SubGrNode, NewNode, NewChoice;
BOOLEAN Fail;

Choices := ChoiceSet;
WHILE NOT EMPTY Choices DO BEGIN
    NewChoice := GET ITEM FROM Choices;
    REMOVE NewChoice FROM Choices;
{ We have a possible assignment. Make sure that an edge to any other
  assigned node exists in the supergraph as well. }

    SubGrSet := SubAdjacentSet[Node];
    SupGrSet := SupAdjacentSet[NewChoice];
    Fail := FALSE;

    WHILE NOT EMPTY SubGrSet AND NOT Fail DO BEGIN
        SubGrNode := GET ITEM FROM SubGrSet;
        REMOVE SubGrNode FROM SubGrSet;

        SupGrNode := Mapping[SubGrNode];
        If SupGrNode > 0 AND SupGrNode NOT MEMBER OF SupGrSet
        THEN Fail := TRUE;
    END;
{ If mapping was successful, then generate a new task to extend
  mapping. If the mapping is finished, output the results.}

    IF NOT Fail THEN BEGIN
        Mapping[Node] := NewChoice;
        IF Node = SubGrSize THEN $Output(Node, Mapping)
        ELSE BEGIN
            NewNode := Node + 1;
            NewChoiceSet := ChoiceSet;
            REMOVE NewChoice FROM NewChoiceSet;
            $TryChoice(NewNode, Mapping, NewChoiceSet);
        END;
    END;
END;
$Halt;
```

Figure 2. Algorithm for substructure search.

on the parallelism, which is new. This algorithm finds *all possible matches* of the substructure on the superstructure. If one only needs to know *if* a match exists, then the central processor could, on receipt of a successful mapping, stop distributing tasks, announce success, and signal all processors to stop. We now turn to details of the algorithm, which we will call PPA for Parallel Permutation Algorithm (Figure 2).

The PPA program code is resident in each processor node. Information about the substructure and superstructure (the number of nodes and adjacency) is passed once to each local processor before the first execution of the code. Variables to be stored in local memory in this manner are referred to as EXTERNAL in the source code. The advantages of preloading this information is that the node does not have to wait for information and communication is minimized.

A processor running this code receives a *task packet* containing the number of the node to assign, the current assignment mapping, and the set of possible assignments for that node. To start the process, a task is created: Node = 1; Mapping = ⟨0, ..., 0⟩; ChoiceSet = {all node numbers in superstructure}. This task is sent to an idle processor. The processor then runs until it executes the $HALT instruction, at which point it signals the controlling processor that it is ready for another task packet.

During execution, only two other situations require messages to be sent to the controlling processor. The first is the output of a successful completed assignment ($OUTPUT). The second is the generation of a new task packet, which is sent to the controlling processor for later distribution ($TRYCHOICE). The central processor collects completed assignments and distributes tasks to keep processors busy. Tasks created when all processors are busy are queued in the central processor.

The basic test for each possible node assignment involves making sure that the connectivity of the supergraph is preserved by the assignment. This is done by checking that two

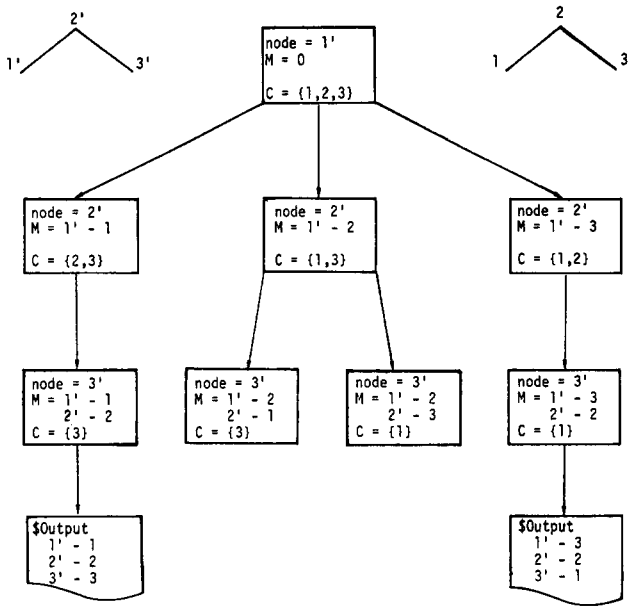

Figure 3. Hierarchy of tasks created by the parallel permutation algorithm in matching substructure propane against superstructure propane.

adjacent nodes in the subgraph are assigned onto two adjacent nodes in the supergraph. If two nodes are connected in the subgraph, the corresponding nodes must also be connected in the supergraph. Failure of this test for a potential assignment does not result in backtracking but rather simply generates no new tasks for that assignment; consequently, that assignment dies without producing any output. The middle tasks in Figure 3 illustrate such nonproductive assignments.

PPA (parallel permutation algorithm) applied to match propane (substructure, primed numbering) against propane (superstructure) generates the task hierarchy shown in Figure 3. Each rectangle represents a task generated; inside are the parameters to $TRYCHOICE: Node, Mapping (M), and ChoiceSet (C). Two tasks generate $OUTPUT, corresponding to the two possible valid mappings for this problem.

DESIGN OF THE SIMULATION

The algorithm was simulated on a DEC-2060 system using the SIMULA simulation language.[21] To test the performance of the algorithm in chemical examples, 25 molecules containing from 6 to 31 non-hydrogen atoms were selected from the Aldrich Chemical catalog and were preprocessed into graphs by removing all atom and bond types. Hydrogen atoms were ignored. Twenty-five pairs of graphs were randomly selected from this set. In each pair, the graph with the fewer nodes was considered the potential subgraph, and the other was considered the potential supergraph.

Restrictions on Subgraph Numberings. The numberings allowed on a graph were restricted to allow the permutation to function efficiently: **Rule 1.** A graph with N nodes had to use the numbers 1, 2, ..., N for the nodes. This assures that all node numbers from 1 to N are assigned to some node. **Rule 2.** The numbering must be *contiguous*; i.e., all collections of nodes 1 to K, with K less than or equal to N, must be connected. Such numberings exist for all graphs[22] and can be rapidly generated by growing a spanning tree (e.g., the SEMA numbering[23]). This assures that, as we assign subgraph nodes sequentially starting at node 1, all the nodes will be simply connected, allowing earlier discovery of the lack of proper adjacency relationships.

The algorithm attempts to assign nodes sequentially starting at node one. Without the contiguity condition, perverse numberings are possible, which cause rapid generation of tasks

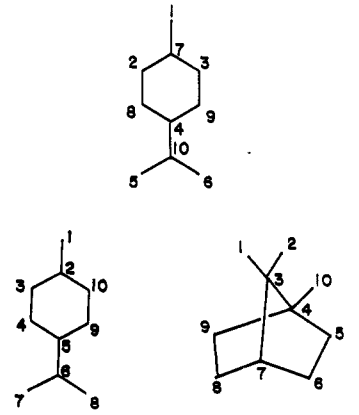

Figure 4. Perverse numbering for query graph.

Table I. Instruction Types Counted in Simulation

instruction	description
Memory Read	read a work from memory
Memory Write	write a work to memory
Memory Test	test register word against memory word
Set Read	fetch a set from memory
Set Write	store a set in memory
Set Test	test a set against another set
Addition	add a value to another value

at the start-up of the algorithm. Figure 4 shows such a numbering. Since none of the first six nodes are contiguous, an attempted mapping against a supergraph of N atoms would result in $N!/(N-6)!$ tasks being generated before any elimination occurs. For example, if the supergraph contained 12 atoms, 665280 tasks would be generated before elimination began.

Timing the Search Process. As a task was executing, counters kept track of the number of each instruction type that was used. Of the seven different instruction types involved (see Table I), all were assigned a cost of one unit except for the three set operations, which were assigned a cost of two units. The "time" was calculated by summing the number of instructions of each type times the cost of that instruction type. Communication costs at task start-up, task completion, and task generation were ignored, except for the cost of initial transfer of the task information package into the memory. This model is valid as long as the central processor is not a bottleneck for task distribution.

There are two limits to the performance of this algorithm with respect to the number of processors. If we have only one processor, then each task packet must wait for the same processor, and we have a model of the sequential algorithm. Thus, too few processors leads to *queuing* of task packets, with all processors receiving a new task packet whenever halted. This is good from the standpoint of utilization of processing power, but it also implies that the system could benefit by the addition of more processors.

At the other extreme, we have many more processors than tasks generated. In this case, a task packet can be sent to a free processor at the moment it is generated, and we have rapid execution of the algorithm. However, it is inefficient with respect to processor time, for most processors will not have a task at any one time and must sit idle.

RESULTS

The interactive simulation program called MOLSIM allows one to run a variety of simulations with minimal effort. A trial run of MOLSIM is shown in Figure 5 for the subgraph and supergraph molecules in Figure 6. User input is underlined.

Processor Utilization. To study the behavior of the algorithm, we ran the same 25 pairs of graphs with increasing

```
@run MOLSIM

(The MOLSIM program is run on a Tops-20 operating system.)

Command: new

 This is molecule number  1

Command: new

 This is molecule number  2

(We use the NEW command to create new graphs.)

Command: makebonds

MOL number: 1

Input atom pairs. Give atom 0 to end.

1 2  2 3  3 4  4 5  5 6  6 7  6 8  5 9  9 10  10 2  0

(We input the query subgraph as a list of edges.)

Command: makebonds

MOL number: 2

Input atom pairs. Give atom 0 to end.

1 3  2 3  3 4  3 7  4 10  4 5  5 6  6 7  7 8  8 9  9 4  0

(The potential supergraph is similarly input as a list of edges.)

Command: setup

 Number of CPU's to setup: 10

 SUBGRAPH mol number: 1
 Subgraph size is 10

 SUPERGRAPH mol number: 2

(The SETUP command sets up the number of processors and the graphs.)

Command: run

 Found a successful mapping.        Found a successful mapping.
```

Figure 5. Trial run of the MOLSIM program.

```
 1--->10                            1--->10
 2---> 4                            2---> 4
 3---> 5                            3---> 5
 4---> 6                            4---> 6
 5---> 7                            5---> 7
 6---> 3                            6---> 3
 7---> 1                            7---> 2
 8---> 2                            8---> 1
 9---> 8                            9---> 8
10---> 9                           10---> 9

 Found a successful mapping.        Found a successful mapping.

 1--->10                            1--->10
 2---> 4                            2---> 4
 3---> 9                            3---> 9
 4---> 8                            4---> 8
 5---> 7                            5---> 7
 6---> 3                            6---> 3
 7---> 1                            7---> 2
 8---> 2                            8---> 1
 9---> 6                            9---> 6
10---> 5                           10---> 5

(The successful assignments of the subgraph onto the supergraph are
now printed out.)

Timing statistics for 10 CPU's:

 I found 4 successful mappings.
 Mapping halted at time 10677.
 Total CPU time used is 103105.
 Amount of time idle is 3%
 Total number of tasks =321
 Average TASK timing is 322
 Shortest TASK time was 76
 Longest  TASK time was 688

(Statistics about the trial run are now presented to the user.
The user can now prepare another simulation, or exit the program.)

Command: quit

8 garbage collection(s) in 572 ms

End of SIMULA program execution.
CPU time: 14.45 Elapsed time: 2:51.75
@
```

Figure 6. Query subgraph and query supergraph used in trial run of MOLSIM.

numbers of processors. *Processor utilization* is defined as the sum of the time used for each task by each processor divided by the total time elapsed from start until the final processor halts. This utilization is the effective speed enhancement gained by the addition of more processors relative to the sequential (one-processor) algorithm.

Table II. Processor Utilization for Simulated PPA Algorithm

no. of CPU[a]	worst util[b]	av util[c]	best util[d]	SD[e]
2	1.988	1.995	2.000	0.003
5	4.713	4.936	4.996	0.058
10	8.755	9.638	9.970	0.260
25	12.341	21.010	24.719	2.720
50	12.341	29.583	48.540	9.693
100	12.341	31.576	92.651	21.984
500	12.341	31.833	149.408	31.349

[a] The number of processors used in the simulation. [b] The worst case utilization (total processor time divided by real time). [c] The average utilization. [d] The best case utilization. [e] Standard deviation from the average of the data points.

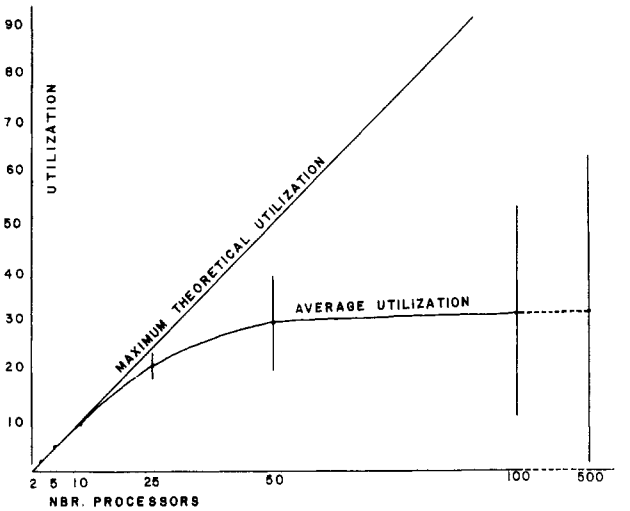

Figure 7. Graph of number of processors vs. average utilization. Vertical bars show standard deviation; the data are also presented in Table II.

The total amount of processor time used by all processors is a constant for a given pair of graphs; what is important is the effect of multiple processors on the halting time. For example, if we divide the task as efficiently as possible between 10 processors, the halting time should be one-tenth of the total processor time needed for the algorithm. In this case, the processor utilization would be 10. In real cases, the utilization will always be less than the theoretical maximum.

Table II shows the result of the simulation for the 25 test pairs of graphs. Figure 7 shows the average utilizatin plotted against the number of processors; the sharp leveling of the curve shows that there seems to be a maximum speedup of the algorithm for the test cases at about 31. Trends in Table II and Figure 7 are discussed below.

For small numbers of processors (2–10), the best and worst utilizations (columns 2 and 4 in Table II) are very close, and the average utilization (column 3) is almost equal to the best utilization possible (column 1). Thus, the algorithm is ideally suited for small-scale parallelism, allowing efficient and almost total use of all processing resources.

For moderate numbers of processors (25–100), the average utilization begins to degrade; for instance, a 100-processor system has the utilization efficiency (on average) of only a 31.5 theoretical processor system. A 400% increase in processors from 25 to 100 results in only a 50% gain in performance. In the worst case, no benefits at all are derived from the additional processing capability.

For large numbers of processors, the average case shows no real increase in performance. The average utilizations for a 100-processor system and a 500-processor system are nearly

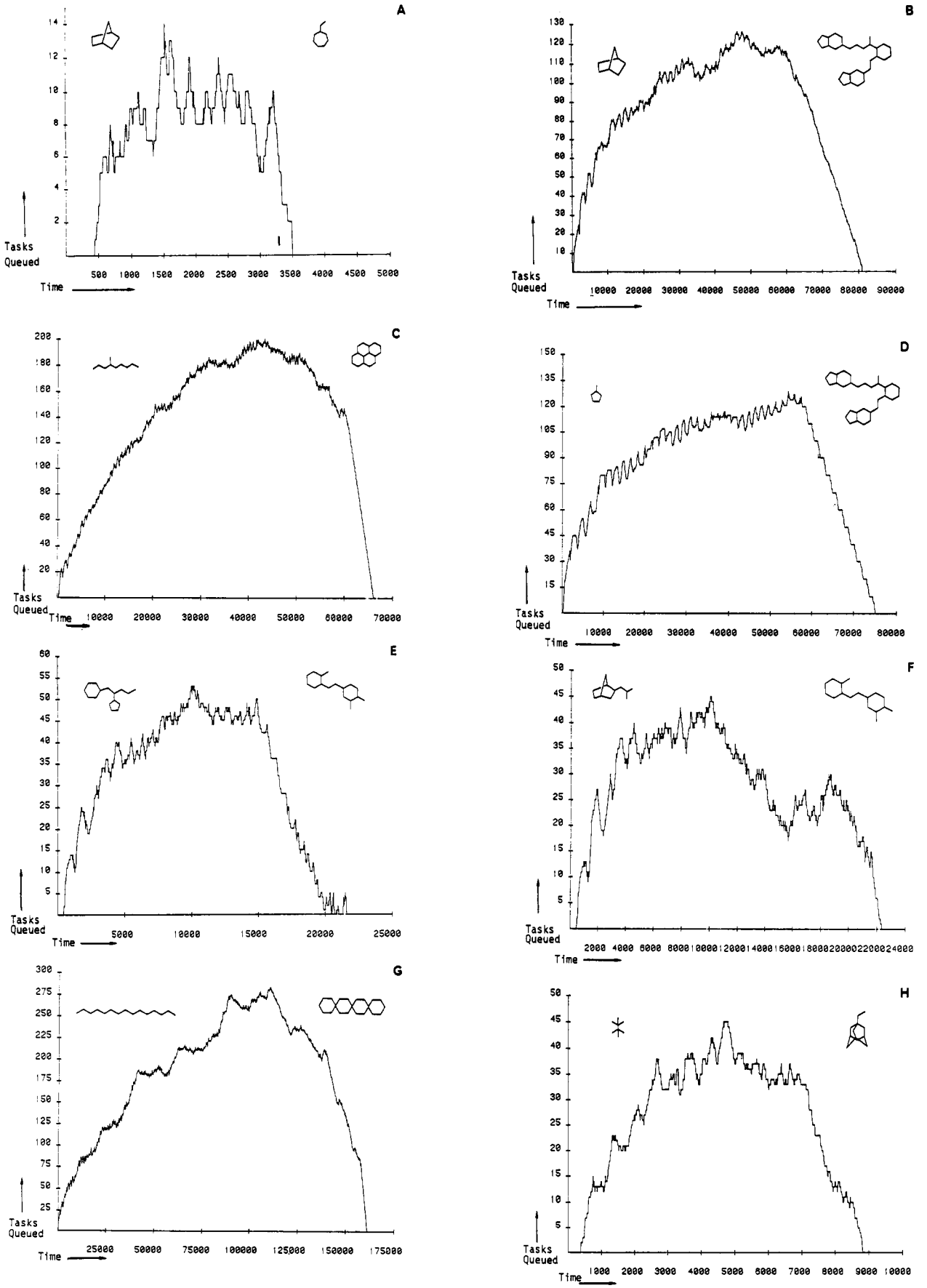

Figure 8. Number of tasks queued vs. time for a 10-processor system for different pairs of graphs. In the plot for each pair, the subgraph is shown in the upper left and the supergraph in the upper right.

identical (31.6 vs. 31.8, respectively). For chemical substructure search queries up to 31 non-hydrogen atoms, this demonstrates a practical maximum of around 50–100 processors; this also illustrates that a speedup of around 30 is likely to be the best attainable with this particular implementation of the PPA. The leveling off of average processor utilization

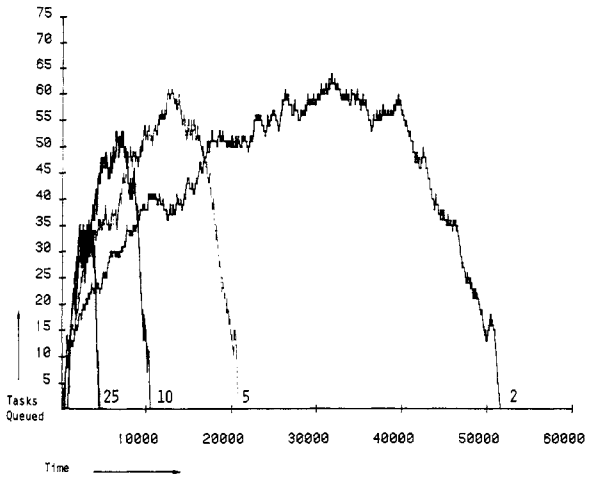

Figure 9. Number of tasks queued vs. absolute time for 2, 5, 10, and 25 processors.

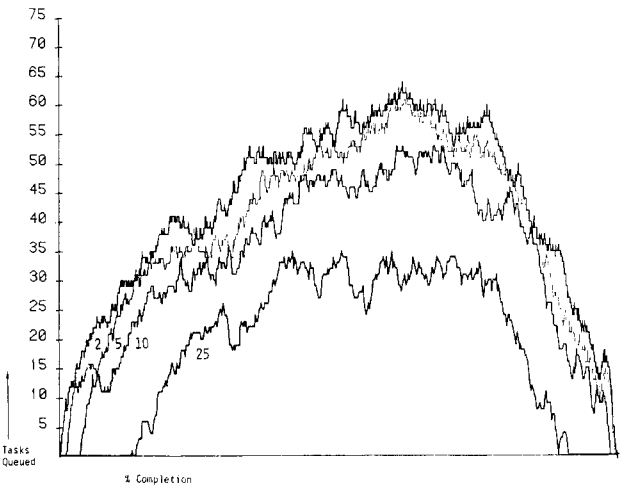

Figure 10. Number of tasks queued vs. *percent completion* for 2, 5, 10, and 25 processors.

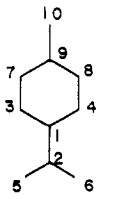

Figure 11. Another contiguous numbering for query graph.

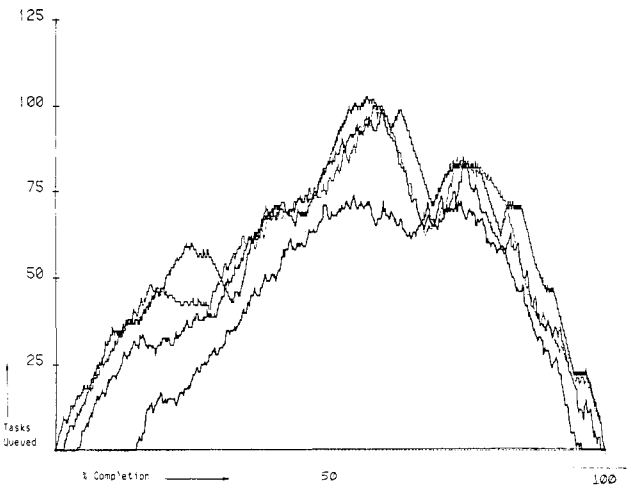

Figure 12. Number of tasks queued vs. percent completion for the graph pair from Figure 6 but with the subgraph numbered as shown in Figure 11. Curves are for 2, 5, 10, and 25 processors.

at around 30 is not related to the fact that the maximum query size was 31 atoms.

Storage Requirements for Central Processor. Table III shows the task generation information for the 25 trials. The number of tasks generated depends only on the nature of the subgraph–supergraph pair and is independent of the number of processors used; the trials confirmed this fact.

Figure 8 shows representative trial runs for the graph pairs with 10 processors. For each run, the number of tasks queued at the central processor is plotted vs. time. From this data we derive the *storage requirements* of the central processor and the *distribution of tasks* during the matching process.

First, the queue memory requirements depend on the number of tasks and their size. A task packet consists of the three $TRYCHOICE arguments (see Figure 2). Of these, the current mapping assignments are by far the largest, needing 1 byte of storage for each node in the subgraph. The worst case for a 30-node subgraph generating a number of tasks near the maximum from Table II would require around 30 × 2270 or 68K bytes of storage; this is a very reasonable amount of storage to require of the central processor.

From our results, the total amount of storage demanded is much less than this theoretical maximum. In the worst case mentioned above (Figure 8G), the maximum number of task packets queued at any one time is not 2270 but is less than 300. This suggests that task storage at the central processor will not be a problem in a physical implementation of this algorithm.

Distribution of Tasks. The shape of the curves for tasks queued vs. time (Figure 8) gives a picture of how the workload develops over time. The queue builds as soon as all processors are occupied, then rises to a maximum at between one-third to two-thirds of the total elasped time, and then declines as tasks terminate without generating new ones. The fact that there are no extreme spikes or valleys means the central processor has a rather even load and can keep the peripheral processors constantly busy.

The height of the curves in Figure 8 measures the complexity of the matching problem. From Figure 8C we observe the result of a large molecule having many possible matches. Similarly, the match in Figure 8G is complex because many assignments match nearly until the end when they are ruled out.

Does the shape of the task distribution curve change as the number of processors is increased? Figure 9 shows the number of tasks queued vs. absolute time for the trial graphs from Figure 6 run with 2, 5, 10, and 25 processors. The absolute completion time decreases as does the maximum number of tasks queued as the number of processors increases, but the shape of the curve remains the same.

This is more clearly illustrated if we plot the number of tasks queued vs. *relative completion* (Figure 10). The similarity between the shapes of the graphs is striking. We conclude the distribution of tasks over relative time is not significantly affected by the number of processors.

Numbering of Query Graph. The algorithm depends on the input numbering of the query subgraph to determine the order in which the atoms will be assigned. Rule 2 (see Design Of The Simulation) described a contiguous numbering condition that must be applied previous to the search. However, many contiguous numberings are possible for a given query graph. As might be expected, different numberings affect the search in significant ways. The same graph given in Figure 6, when renumbered as shown in Figure 11, gives the task distribution curves shown in Figure 12. While the general shape remains, the new curves show more local structure and higher maxima

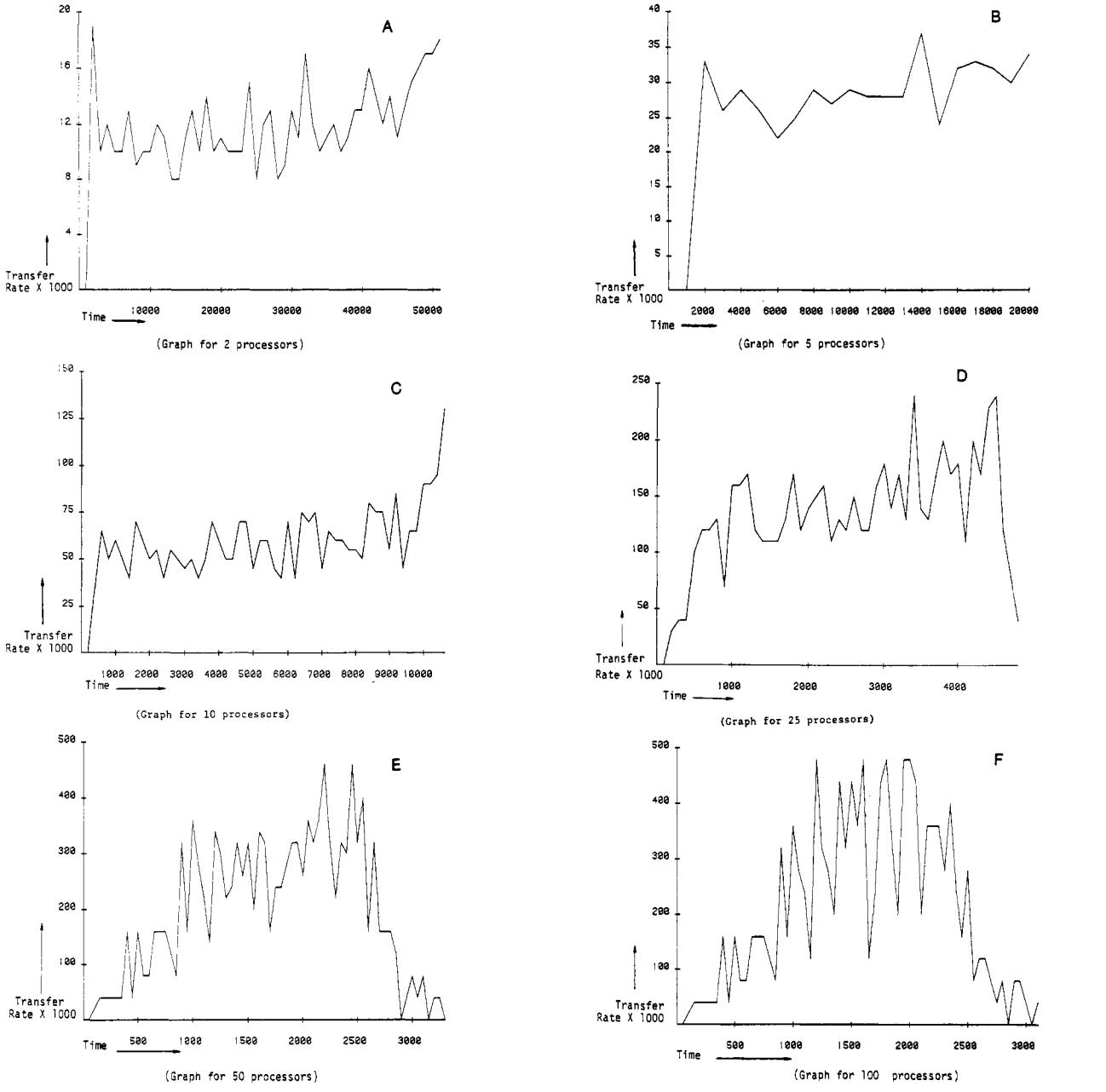

Figure 13. Task transfer rate (×1000) vs. time: (A) graph for 2 processors; (B) graph for 5 processors; (C) graph for 10 processors; (D) graph for 25 processors; (E) graph for 50 processors; (F) graph for 100 processors.

Table III. Task Generation for 25 Sample Searches

av no. of tasks[a]	min no. of tasks[b]	max no. of tasks[c]
383	92	2270

[a] The average number of tasks generated during a search. [b] Of all 25 searches, this is the minimum number of tasks generated by a search. [c] Of all 25 searches, this the maximum number of tasks generated by a search.

in the number of tasks queued.

Data-Transfer Requirements. The number of tasks queued is the excess of tasks created over tasks completed, thus not a good reflection of data-transfer rate. To directly calculate the data-transfer rate, we counted the number of tasks sent and received by the central processor during a specified period of time. The graph pair from Figure 6 was taken through a series of runs with 2, 5, 10, 25, 50, and 100 processors. Figure 13 shows the dynamic rate of task transfer (×1000) vs. time for each member of the series. Recall that changing the number of processors does not change the number of task packets generated; thus, changes in data-transfer rate are due to compaction of the time scale and changes in local distribution of the task generation times.

Table IV. Communication Rates Needed for Transferring Tasks between Central Processor and Terminal Processors

no. of processors	max transfer rate (Mbaud)	no. of processors	max transfer rate (Mbaud)
2	2.4	25	30.0
5	4.2	50	54.0
10	14.4	100	60.0

In this example, the task packet has a maximum of around 15 bytes (120 bits). Assuming that on our time scale an instruction executes at 10^{-6} s, Table IV shows the approximate maximum data-transfer rates experienced. (Each unit of the rate scale in Figure 13 translates into a data-transfer requirement of approximately 120K baud.) Data-transfer rates varied between 2.4 and 60 Mbaud, within the range of the data-transfer rate of a high-speed bus. Since most of the graphs are relatively smooth, with the maximum data-transfer rate only a few times the average transfer rate, the bus should be evenly loaded for most of the program run.

Modifications could be made to the parallel permutation algorithm in a number of directions. First, processor nodes could retain some tasks, in effect increasing the amount of work a processor can perform before needing to communiate with the central processor. Second, partitioning techniques[14] could be utilized to reduce needless task generation, e.g., avoiding the assignment of degree-two substructure nodes to degree-one superstructure nodes (2′–1 or 2′–3 in Figure 3). Time savings from such improvements may on "underloaded" parallel systems result in lower processor utilization but will still reduce the elapsed computation time.

CONCLUSIONS

This simulation demonstrates that a moderate number of general-purpose processors in a simple "star" architecture can be used efficiently to speedup the atom-by-atom mapping stage of substructure search. The algorithm decomposes the problem into a reasonable number of independent tasks, which can be solved in parallel without backtracking. Storage and communication costs were shown to be of reasonable magnitude on the basis of today's technology. The results suggest that an implementation using 5–25 processors should utilize the local processors in an efficient manner for chemical structures of up to 32 non-hydrogen atoms, should not overload the central processor, and should give speed enhancements of up to 24 times the speed of a single processor (for the 25-processor implementation). The PPA can, in principle, handle any-sized molecule and is not limited by the 32 non-hydrogen atom limitation we imposed for the simulation of the PPA. The simulation was simply done with molecules smaller than 32 atoms. One could combine PPA with data base parallelism by dividing the file among n such parallel systems to achieve a $24n$ speed enhancement (for the 25-processor implementation).

ACKNOWLEDGMENT

This work was supported in part by the Biotechnology Resources Program, Division of Research Resources, National Institutes of Health, through Grant RR01059 and an allocation of the SUMEX Resource (RR00785), through ES02845, and through grants from the Stauffer Chemical Co.

REFERENCES AND NOTES

(1) Wipke, W. T. "Computer-Assisted Three-Dimensional Synthetic Analysis". In "Computer Representation and Manipulation of Chemical Information"; Wipke, W. T.; Heller, S. R.; Feldmann, R. J.; Hyde, E., Eds.; Wiley: New York, 1974; pp 147–174.
(2) Wipke, W. T.; Braun, H.; Smith, G.; Choplin, F.; Sieber, W. "Computer-Assisted Organic Synthesis". *ACS Symp. Ser.* **1977**, *No. 61*.
(3) Wipke, W. T.; Ouchi, G. I.; Chou, J. T. "Computer-Assisted Prediction of Metabolism". In "Structure Activity Correlation as a Predictive Tool in Toxicology: Fundamentals, Methods, and Applications"; Goldberg, L., Ed.; Hemisphere: New York, 1983; pp 151–169.
(4) Smith, D. H.; Masinter, L. M.; Sridharan, N. S. "Heuristic DENDRAL: Analysis of Molecular Structure". In "Computer Representation and Manipulation of Chemical Information"; Wipke, W. T.; Heller, S. R.; Feldmann, R. J.; Hyde, E., Eds.; Wiley: New York, 1974; p 287.
(5) Masinter, L. M.; Sridharan, N. S.; Lederberg, J.; Smith, D. H. "Applications of Artificial Intelligence for Chemical Inference. XII. Exhaustive Generation of Cyclic and Acyclic Isomers". *J. Am. Chem. Soc.* **1974**, *96*, 7702.
(6) Wipke, W. T.; Rogers, D. "Artificial Intelligence in Organic Synthesis. SST: Starting Material Selection Strategies. An Application of Superstructure Search". *J. Chem. Inf. Comput. Sci.* **1984**, *24*, 71–81.
(7) Howe, W. J.; Hagadone, T. J. "Molecular Structure Searching: Computer Graphics and Query Entry Methodology". *J. Chem. Inf. Comput. Sci.* **1982**, *22*, 8–15.
(8) Dittmar, P. G.; Stobaugh, R. E.; Watson, C. E. "The Chemical Abstracts Service Chemical Registry System". *J. Chem. Inf. Comput. Sci.* **1976**, *16*, 111–121.
(9) Wipke, W. T.; Dill, J. D.; Peacock, S.; Hounshell, D. "Search and Retrieval Using an Automated Molecular Access System". Presented at the 182nd National Meeting of the American Chemical Society, New York, Aug 1981.
(10) Wipke, W. T. "Three Dimensional Substructure Search". Presented at the 186th National Meeting of the American Chemical Society, Washington, DC, Aug 1983.
(11) Tarjan, R. E. "Graph Algorithms in Chemical Computation". In "Algorithms for Chemical Computation"; Christofferson, R. E., Ed.; American Chemical Society: Washington, DC, 1977; Vol. 46, pp 1–20.
(12) The *order* of an algorithm is a measure of how costly an algorithm becomes as we increase the size of the problem. If this difficulty can be approximated by a polynomial, the algorithm is said to be of *polynomial time*. NP-complete algorithms are especially difficult and cannot be represented in polynomial time.
(13) Lykos, P.; Shavitt, I. "Supercomputers in Chemistry". *ACS Symp. Ser.* **1981**, *No. 173*.
(14) Sussenguth, E. H. "A Graph-Theoretic Algorithm for Matching Chemical Structures". *J. Chem. Doc.* **1965**, *5* (1), 36–43.
(15) Farmer, N. A.; O'Hara, M. P. "CAS-ONLINE: A New Source of Substance Information from Chemical Abstracts Service". *Database* **1980**, *3* (4), 10–25.
(16) "CAS Offers New Online Service". *Chem. Eng. News* **1980**, *58* (40), 34–35.
(17) Ullman, J. R. "An Algorithm for Subgraph Isomorphism". *J. Assoc. Comput. Mach.* **1976**, *23* (1), 31–42.
(18) Colton, Bruce "The Advanced Flexible Processor, Array Architecture". In "Supercomputers in Chemistry"; Lykos, P.; Shavitt, I., Eds.; American Chemical Society: Washington, DC, 1981; Vol. 173, Chapter 15.
(19) Papazoglou, M. P.; Georgiadis, P. I.; Martsas, D. G. "Designing a Parallel Simula Machine". *Comput. Design* **1983**, *Oct*, 125–132.
(20) Berman, F.; Snyder, L., "On Mapping Parallel Algorithms into Parallel Architectures". Submitted for publication in *Parallel Process. Conf.*
(21) Arnborg, S.; Bjorner, O.; Enderin, L.; Engstrom, E.; Karlsson, R.; Ohlin, M.; Palme, J.; Wennerstrom, I.; Wihlborg, L. "DECsystem-10 SIMULA Language Handbook". Sept 1975, Technical Report, Swedish National Defense Research Institute.
(22) Even, S. "Graph Algorithms"; Computer Science Press: Potomac, MD, 1979.
(23) Wipke, W. T.; Dyott, T. M. "Stereochemically Unique Naming Algorithm". *J. Am. Chem. Soc.* **1974**, *96*, 4834.