generously supported by the Bundesministerium für Forschung und Technologie (BMFT). The authors and the Beilstein Institute gratefully acknowledge this support. We would also like to thank Dr. G. Penzlin of the Beilstein Nomenclature Department for his expert advice.

REFERENCES AND NOTES

(1) Part 1: Wisniewski, J. L. *J. Chem. Inf. Comput. Sci.* **1990**, *30*, 324-332.
(2) International Union of Pure and Applied Chemistry, Organic Chemistry Division, Commission on Nomenclature of Organic Chemistry. *Nomenclature of Organic Chemistry, Sections A, B, C, D, E, F, and H*, 1979 ed.; Rigaudy, J., Klesney, S. P., Eds.; Pergamon Press: Oxford, England, 1979; 559 pp.
(3) Cooke-Fox, D. I.; Kirby, G. H.; Rayner, J. D. Computer Translation of IUPAC Systematic Organic Chemical Nomenclature. *J. Chem. Inf. Comput. Sci.* **1989**, *29*, 101 and references therein.
(4) Ref 2, p 6.
(5) Ref 2, p 7.
(6) Ref 2, p 9.
(7) Ref 2, p 80.
(8) Weissbach, O. Private communication.
(9) Ref 2, p 13.
(10) Ref 2, p 14.
(11) Ref 2, p 17.
(12) Ref 2, p 18.
(13) Ref 2, p 19.
(14) Ref 2, p 20.
(15) Ref 2, p 21.
(16) Ref 2, p 108.
(17) Ref 2, p 20.
(18) Ref 2, pp 22-27.
(19) Ref 2, p 28.
(20) Ref 2, p 30.
(21) Rücker, G.; Rücker, C. Nomenclature of Organic Polycycles out of the Computer—How to Escape the Jungle of the Secondary Bridges. *Chimia* **1990**, *44*, 116.
(22) Ref 2, p 31.
(23) Ref 2, p 35.
(24) Ref 2, p 38.
(25) Ref 2, p 42.
(26) Ref 2, p 106.
(27) Ref 2, p 46.
(28) Haberfield, P. *Chem. Eng. News* **1990**, *68* (34), 4.
(29) Ref 2, p 47.
(30) Ref 2, p 53.
(31) International Union of Pure and Applied Chemistry, Organic Chemistry Division, Commission on Nomenclature of Organic Chemistry, Revision of the Extended Hantzsch-Widman System of Nomenclature for Heteromonocycles (Recommendations 1982). *Pure Appl. Chem.* **1983**, *55*, 409-416 and references therein.
(32) Ref 2, p 56.
(33) Ref 2, p 62.
(34) Ref 2, p 64.
(35) Ref 2, p 101.
(36) Ref 2, p 64.

Atom-by-Atom Searching Using Massive Parallelism. Implementation of the Ullmann Subgraph Isomorphism Algorithm on the Distributed Array Processor†

PETER WILLETT* and TERENCE WILSON

Department of Information Studies, University of Sheffield, Western Bank, Sheffield S10 2TN, U.K.

STEWART F. REDDAWAY

Active Memory Technology, 65 Suttons Park Avenue, Reading RG6 1AZ, U.K.

Received January 30, 1991

The AMT Distributed Array Processor (DAP) is a massively parallel SIMD processor array that contains thousands of processing elements. This paper describes the implementation of atom-by-atom searching on the DAP using Ullmann's subgraph isomorphism algorithm. Two alternative algorithms are discussed. The first of these allows rapid processing of a single structure, the adjacency matrix of which is distributed across the array of processing elements. The second is much slower in execution for a single molecule but allows very large numbers of structures to be searched in parallel. With current codes, the first algorithm is faster and out-performs a large mainframe; however, developments of the second algorithm are expected to make this the faster. Combined algorithms are also described that utilize both approaches.

INTRODUCTION

Parallel processing involves the use of some, or many, processors operating in parallel to reduce the elapsed time that is required to carry out some computational task.[1-3] Parallel processing techniques have been used for scientific and engineering applications for many years; more recently, there has been increasing interest in the use of such techniques for applications that involve extensive nonnumeric or symbolic computation. One such application is that of database searching, which involves the matching of a query record against a large database of analogous records to identify those that satisfy some retrieval criterion. While each individual matching operation may be quite rapid in execution, searching can be highly demanding of computational resources when a large database containing many records needs to be processed: this suggests the use of a *data parallel*[4] system architecture that allows the query record to be compared with as many database records as possible at the same time. The last few years have seen many reports of the use of parallel computer hardware for a range of chemical and biochemical database applications. Examples of this include studies of searching protein sequence databases,[5,6] of clustering chemical structure databases,[7,8] and of both atom-by-atom[9-11] and maximal common substructure searching.[12]

In this paper, we report the initial results of an investigation of the implementation of atom-by-atom searching using a massively parallel computer, the Active Memory Technology (AMT) Distributed Array Processor (DAP). The second section describes the particular atom-by-atom searching procedure that we have investigated, this being based on Ullmann's subgraph isomorphism algorithm.[13] The third section then describes the DAP. The fourth section presents two, very different, ways of implementing the Ullmann algorithm on the

* To whom all correspondence should be addressed.
† Dedicated to Michael Lynch to celebrate his 25 years at the University of Sheffield.

0095-2338/91/1631-0225$02.50/0

DAP, and this is followed, in the fifth section, by the results of atom-by-atom searches that use these two implementations. The paper concludes with a summary of our main findings and suggestions for future work.

ULLMANN'S SUBGRAPH ISOMORPHISM ALGORITHM

Chemical substructure searching involves the identification of all of those compounds in a structure database that contain some specified query substructure, irrespective of the environment in which the query substructure occurs. Substructure searching is the chemical equivalent of the graph-theoretic problem of subgraph isomorphism, which involves determining whether a query graph is contained within another graph and which is known to belong to the class of NP-complete computational problems;[14,15] in the chemical context, subgraph isomorphism is usually referred to as atom-by-atom searching. The NP-complete nature of atom-by-atom searching means that a chemical substructure search normally commences with a screening search, which rapidly eliminates that great percentage of the database that cannot possibly contain the query substructure.[16,17]

There is an obvious brute-force algorithm for subgraph isomorphism, given the adjacency matrices **A** and **B** representing a query pattern and a database structure containing P_α and P_β atoms, respectively. The algorithm involves generating all $P_\beta!/[P_\alpha!(P_\beta - P_\alpha)!]$ combinations of P_α atoms from the structure and then determining whether any of these combinations is an exact match for the query pattern, this match involving an exhaustive check of the arrangement of all of the atoms and bonds. This is prohibitively expensive for all but the smallest queries and structures, and backtracking and pruning procedures are normally used to order and to control the search. Such procedures are widely used for improving the efficiency of graph matching operations (and of other types of combinatorial problems that can be expressed as a depth-first tree search).[18]

Ullmann noted that drastic increases in the efficiency of depth-first backtracking search can be obtained by the use of a *refinement procedure*.[13] This is a heuristic that is called at each node of the search tree in order to eliminate branches leading to mismatches and, thus, to minimize the amount of searching that is required. Ullmann advocated the use of the refinement technique in the context of general graphs. Brint and Willett demonstrated that it appeared to be very well suited to the processing of 3-D chemical graphs,[19] and the algorithm has been used in 3-D substructure searching systems for both small molecules[20] and macromolecules.[21] Downs et al.[11] subsequently showed that the refinement procedure was analogous to the use of relaxation techniques in substructure searching[22] and that it appeared to be more efficient for 2-D substructure searching than the conventional set reduction algorithm.

Using the notation given previously, the refinement procedure involves a *match matrix*, **M**, that contains all of the possible equivalences between atoms from **A** and atoms from **B**. The elements of this matrix, m_{ij} $(1 \le i \le P_\alpha; 1 \le j \le P_\beta)$ are such that

$$m_{ij} = \begin{cases} 1 & \text{if the } i\text{th atom of A can be mapped to the } j\text{th atom of B} \\ 0 & \text{otherwise} \end{cases}$$

M is initialized by setting to one all elements that correspond to pairs of atoms $\{a_i,b_j\}$ that are of the same atomic type and that have sets of connected atoms and bonds such that the set surrounding a_i is identical with or a subset of that surrounding b_j.

The Ullmann heuristic may be stated as follows: given a query atom a_i that has a neighbor x, and a database atom b_j that can be mapped to a_i, then there must exist a neighbor of b_j, y, that can be mapped to x. This can be stated more formally as follows: if a_i corresponds to b_j, i.e., if $m_{ij} = 1$, then

$$\forall\, x(1 \ldots P_\alpha)[(a_{ix} = 1) \Rightarrow \exists\, y(1 \ldots P_\beta)(m_{xy}b_{jy} = 1))$$

(i.e., for all x, if x is a neighbor of a_i, this implies that there must exist an atom y in **B** such that x and y correspond, and y is a neighbor of b_j). If this condition is *not* true, then m_{ij} is set to 0.

The heuristic above is tested for every $m_{ij} = 1$, with $i = 1 \ldots P_\alpha$ and with $j = 1 \ldots P_\beta$. Generally some of the m_{ij} elements will be negated, thus reducing the number of possible mappings between query and database atoms. If during this process *any* element of **M** is negated, the refinement is repeated since this may cause further changes to be made. If at any stage during the refinement process the following condition holds

$$\exists\, i(1 \ldots P_\alpha)[m_{ij} = 0\ \forall\, j(1 \ldots P_\beta)]$$

(i.e., there exists an atom i in **A** such that $m_{ij} = 0$ for all of the atoms, j, of **B**) then a mismatch is identified, since there exists a query atom that cannot be mapped to any of the database atoms. This condition is tested after every iteration of the refinement procedure so that mismatches are identified as rapidly as possible.

The refinement procedure can thus be expressed in pseudo-code as follows:

```
mismatch:=FALSE
REPEAT
  change := FALSE
  FOR i := 1 TO Pα DO
    FOR j := 1 TO Pβ DO
      IF mij THEN
        BEGIN
          mij := ∀ x (1...Pα) ((aix = 1) ⇒ ∃y (1...Pβ) (mxy · byj = 1))
          change:=change OR (NOT(mij))
        END
  mismatch:=∃ i (1...Pα)(mij = 0 ∀ j (1...Pβ)
UNTIL (NOT(change) OR mismatch)
```

The description so far has been simplified in that we have considered only binary adjacency matrices, i.e., matrices that note whether two atoms are connected, without consideration of the type of bond that links them together. Such information is readily incorporated by assuming that the matrices **A** and **B** contain integer bond types rather than Boolean connectivities. In this case, the Ullmann heuristic may be restated as follows: Given a query atom a_i that has a neighbor x connected to it by a bond of type t, and a database atom b_j that can ba mapped to a_i, then there must exist a neighbor of b_j, y, that is connected to it by a bond of type t and that can be mapped to x. Note that the symmetric nature of the adjacency matrices means that $b_{yj} = b_{jy}$ and thus the mathematical formulation of the heuristic is

$$\forall\, x(1 \ldots P_\alpha)[(a_{ix} = t) \Rightarrow \exists\, y(1 \ldots P_\beta)(m_{xy} \wedge (b_{jy} = t))]$$

A pseudo-code version of the complete Ullmann algorithm is given in the Appendix to this paper, while a worked example of its use is presented by Mitchell et al.[21]

DISTRIBUTED ARRAY PROCESSOR

The DAP is an example of a *processor array*.[23-27] The earliest designs for processor arrays, by Unger[28] and by Slotnick et al.,[29] defined the basic machine architecture that consists of a 2-D synchronous array of simple processing elements, or PEs, under the supervision of a single master control unit, or MCU, which broadcasts instructions for execution in the processor array. A schematic diagram of the DAP is shown in Figure 1, from which it will be seen that a processor array is an example of a single instruction stream, multiple

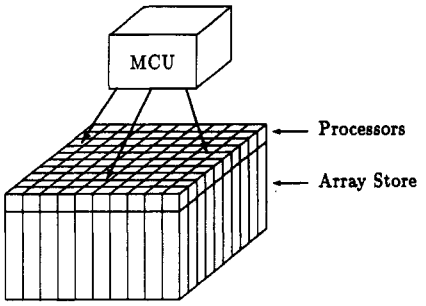

Figure 1. Schematic diagram of the DAP, illustrating the way in which instructions are broadcast from the MCU to the processor array.

data stream (or SIMD) computer[1] with all of the PEs executing the same sequence of instructions in parallel: a facility known as *masking* can be used to disable PEs temporarily as a result of previously executed instructions. Individual types of processor array are characterized by the number and complexity of the PEs, the size of the local memory that is associated with each of the PEs, and the type of interconnection network that is used to link the PEs together.[1] The first processor array actually constructed was the ILLIAC IV, which was a word-slice computer with one word per PE processed at a time. More recent processor arrays, such as the DAP, the Massively Parallel Processor,[2,30] and the Connection Machine,[31,32] have all been based on arrays of bit-organized PEs. The bit-level organization results in arithmetic processes being carried out in software, so that operation times are dependent on their bit complexity and precision. Thus, an operation on a 16-bit integer is faster than that on a 32-bit integer, and integer operations are much faster than real operations. This characteristic of the DAP also ensures that logical operations are executed very efficiently indeed and that fast, high-level functions can be built up in software to operate in parallel on the array of PEs.

The DAP was the first array processor to be marketed commercially, having been originally developed by International Computers Limited using a large mainframe processor as the host; the DAP is now manufactured by AMT, who produce versions hosted by VAX or SUN workstations. Current models consist of 1024 or 4096 PEs in a 32 × 32 or 64 × 64 2-D array (the DAP 510 and DAP 610, respectively) and with memory sizes of up to 512 Mbytes. An individual PE is connected to its N, E, S, and W neighbors, so as to give a 2-D array, and the PEs in each row and column are connected by data highways, which allow data to be broadcast (or received) from the MCU (or all of the PEs) simultaneously. These linkages are all bit-serial in nature and are illustrated in Figure 2. PEs at the edges of the array are connected to those on the opposite boundary in a wrap-around fashion. There is a body of powerful software, the Parallel Data Transforms (PDTs) referred to under Algorithm II, for achieving a wide range of types of data rearrangement. A fast input/output channel allows the transfer of data in or out of DAP memory at a rate of some 50 Mbytes/s, this characteristic making the DAP well suited to real-time applications such as signal processing and graphics. The DAP is usually programmed in either a high-level Fortran extension, Fortran Plus, which was used for all of the work reported here, or a low-level assembler, APAL (although other language extensions, such as C, Prolog and Ada, are under development or have been studied). Fortran Plus is essentially an extension of Fortran 77 that allows the full exploitation of the parallel capabilities of the DAP and that encompasses many of the features that are expected to become available in Fortran 90, e.g., vector and matrix handling functions.

A national DAP service is provided in the U.K. by the Centre for Parallel Computing at Queen Mary and Westfield College, University of London. The experiments here used a DAP 610 at the Centre. This machine contains a 64 × 64 array of PEs, each of which has 8 Kbytes of local storage, giving 32 Mbytes overall.

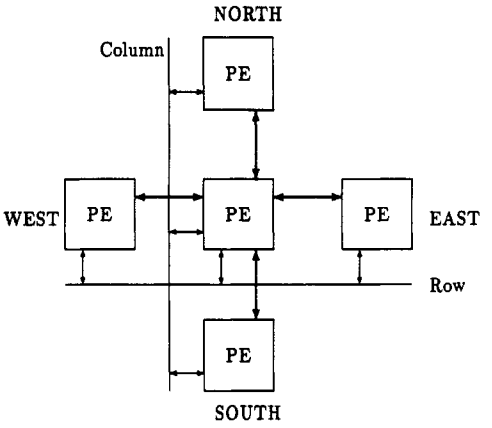

Figure 2. Interconnection of the PEs comprising the DAP's processor array.

PARALLEL IMPLEMENTATIONS OF THE ULLMANN ALGORITHM

Introduction. There are two main types of data parallelism that can be used to implement a sequential algorithm on a processor array:[33]

Inner loop parallelism involves the identification of any inherent parallelism in the algorithm that is to be implemented, so that different parts of it can be executed concurrently. On an SIMD machine, this involves applying the same operations to (subsets of) the data structures involved in the algorithm.

Outer loop parallelism involves the same, undivided algorithm being applied to different data items at the same time.[4] This type of parallelism is particularly well suited to the architecture of processor arrays since the records comprising the database that is to be searched can be distributed across the available PEs. A search is then effected by loading the query record into the array processor's master control unit and broadcasting the attributes that have been specified in the query for comparison with the sets of attributes characterizing the records stored in each of the PEs. The distribution of the database may be done statically, by assigning each PE one record (or group of records), with all PEs sharing the same code; a review of the use of processor arrays for database searching using this approach is provided by Willett and Rasmussen.[27] Alternatively, the distribution of the database can be carried out dynamically by means of load balancing procedures, which ensure that records are assigned to PEs during program execution as the latter become available. This model of computation is often used with parallel machines that make use of the *processor farm* model of computation, where computational work is allocated to processors as they become free for further processing; the use of processor farms for database searching is discussed by Cringean et al.[35] and by Downs et al.[11]

These two, rather different approaches to parallel searching are exemplified by the two DAP implementations of the Ullmann algorithm that we have studied as described below.[36] Algorithm I uses inner loop parallelism while Algorithm II uses outer loop parallelism.

Algorithm I. The Ullmann algorithm consists of two main parts, these being the depth-first backtracking tree search and the refinement procedure that is called at each node of this search tree. Much of the first stage is inherently sequential

in character and thus not obviously suited to an SIMD implementation. However, the great bulk of the processing takes place during the refinement procedure, and this is eminently well suited to the DAP, since refinement involves a large number of logical operations that can be mapped onto the DAP with great efficiency. Specifically, one bit in each of the 64 × 64 array of PEs can be used to represent one element in each of the logical matrices, **A**, **B**, and **M**, that lie at the heart of the Ullmann algorithm: The DAP allows the simultaneous processing of an entire matrix, whereas a conventional processor requires that each matrix element, m_{ij}, must be processed in sequence.

We have noted that the Ullmann heuristic is

$$\forall\ x(1 \ldots P_\alpha)[(a_{ix} = 1) \Rightarrow \exists\ y(1 \ldots P_\beta)(m_{xy}b_{jy} = 1)] \quad (1)$$

Let us define a vector **R** the elements of which, r_i, are given by

$$r_i = \exists\ y(1 \ldots P_\beta)(m_{xy}b_{yj} = 1) \quad (2)$$

then the Ullman condition may be rewritten as

$$m_{ij} = m_{ij} \wedge \overline{\bar{r}_i \wedge a_{ix}}\ \forall\ i, j \text{ such that } m_{ij} = 1 \quad (3)$$

This is the reverse of eq 1 and is inefficient for sequential implementation: it is, however, useful when considering a parallel implementation of the algorithm.

If **A**, **B**, and **M** are DAP-sized matrices, with each element corresponding to a single PE, r_i ($i = 1 \ldots P_\alpha$) may be evaluated in parallel and the results placed in a vector **R**:

$$r_i = \exists\ y(1 \ldots P_\beta)(m_{xy}b_{yj} = 1)\ \forall\ x = j \wedge x = 1 \ldots P_\alpha \quad (4)$$

The matrices **M** and **B** must be aligned to allow this operation to be executed. This is done by working with $\mathbf{M}^T$, instead of **M**, to allow $m_{xy}b_{yj}$ to be executed as a single logical AND of two matrices. The vector **R** may be determined by using the DAP function ORROWS, which evaluates the OR of all of the elements in a matrix column for all of the rows in the matrix: this operation is also performed efficiently on the DAP.

To complete a single relaxation eq 3 must be completed for $j = 1 \ldots P_\beta$ given a fixed x. For the DAP implementation the reverse case is adopted: when executing eq 3, x varies over $j = 1 \ldots P_\alpha$, since we deal with the matrix **A** in parallel and j is fixed by $x = j$. In order to vary j in the domain $j = 1 \ldots P_\beta$ for a fixed x, $\mathbf{M}^T$ is shifted after each execution of eq 3, P_β times in all. It follows that the matrix **A** must also be shifted similarly, which corresponds to checking the heuristic for a single neighbor x and updating **M** accordingly. To execute the heuristic for the other neighbors, the adjacency matrices must be aligned correctly by shifting. The actual order in which neighbors are processed depends upon the position in the array: however, the heuristic is complete for every neighbor after P_β shifts. Equations 3 and 4 may be written in Fortran Plus as

```
R=MATR(ORROWS(M.AND.B))
M=M.AND..NOT.((.NOT.R).AND.A)
```

where MATR is a DAP operator that replicates a vector *ES* times to fill a matrix rowwise (*ES* is the edge size of the DAP, which was 64 for the DAP 610 that we have used). The second line of code can be simplified by using De Morgan's Theorem to give

```
M=M.AND.(R.OR..NOT.A)
```

The relaxation algorithm may thus be implemented as follows, where ALL, LEQ, SHEC, and TRAN are Fortran Plus operators for checking that some condition holds in all of the PEs, for logical equivalence, for shifting East in a cyclic manner, and for transposing a matrix, respectively, and where ORCOLS is analogous to ORROWS:

```
REPEAT
  MTRAN=TRAN(M)
  MCOPY=M
  FOR J:=1 TO Pβ DO
    BEGIN
      R=MATR(ORROWS(MTRAN.AND.B))
      M=M.AND.(R.OR..NOT.A)
      MTRAN=SHEC(MTRAN)
      A=SHEC(A)
    END
  EXIT=.NOT.(ALL(ORCOLS(M)))
  NOCHANGE=ALL(MCOPY.LEQ.M)
UNTIL (EXIT.OR.NOCHANGE)
```

This code requires the use of a DAP with an edge size, *ES*, such that $ES = P_\beta$, i.e., a DAP that contains $P_\beta \times P_\beta$ PEs, if the cyclic shifts are to route the data in **A** and MTRAN correctly. This is unlikely to be the case in practice, and one of the following strategies will need to be adopted:[36]

Perform the loop *ES* times: this is optimally efficient when $P_\beta = ES$ but is much less efficient when $P_\beta \ll ES$

Introduce additional shifts that ensure the correct routing of the data: this is more efficient than the first when $P_\beta \ll ES$ but less so when $P_\beta \approx ES$

The former strategy was adopted for all of our experiments.

It is of interest to compare the complexity of this parallel version of the refinement procedure with the sequential version described previously. The latter has a time complexity of order $O(P_\alpha^2 P_\beta)$ since the basic condition must be checked for each of the $P_\alpha \times P_\beta$ elements of **M**, each such check requiring consideration of all of the $P_\alpha - 1$ neighbors, a_{ix}, of each pattern atom a_i. In practice, however, only those elements of **M**, m_{ij}, are checked for which $m_{ij} = 1$; since **M** is generally sparse, the sequential algorithm is much more efficient in execution than the complexity would suggest. The parallel version has a time complexity of order $O(P_\beta)$ on an $O(ES^2)$-PE DAP, since P_β array operations are performed. This version considers every element of **M**, irrespective of whether m_{ij} is one or zero, and thus the efficiency of the parallel version should increase, relative to the sequential version, as **M** becomes densely populated. However, the availability of both atom and bond types means that the **M** matrices used in chemical applications tend to be fairly sparsely populated, which allows efficient sequential processing but which cannot be fully utilized in parallel processing. We note in passing that other, nonchemical applications of graph matching involve graphs that would yield much denser **M** matrices than those considered here: dense matrices are processed very slowly on a sequential processor, and it might thus be of interest to consider the use of Algorithm I on such data.

The parallel implementation described above assumes that we are dealing with binary adjacency matrices, where the elements are set to TRUE if the corresponding atoms are bonded together but without specifying the type of linkage (single, double, etc.). The use of byte or integer matrices makes no difference on a conventional sequential machine, where the bits comprising a single word are processed in parallel. However, the DAP PEs are bit-organized processors, and thus the inclusion of bond-type information necessitates processing additional to that required for binary adjacency matrices. Specifically, if b bond types are used to characterize the molecules in a dataset, $\log(b)$ operations must be carried out for each single operation in the binary case (since each check for the equivalence of a bond in a database structure and in the query substructure will require a comparison of the binary representations of the bond orders). Thus, if a total of eight different bond types is used, as was the case in the experiments reported below, the refinement is expected to take as much as three times as long as would be the case if this

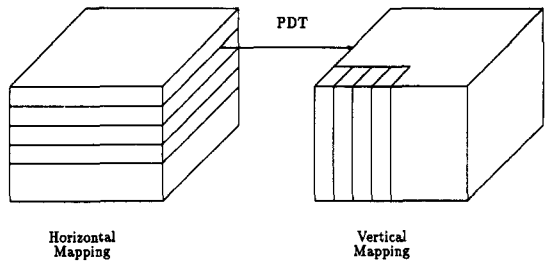

Figure 3. Use of a PDT to convert from a horizontal to a vertical data mapping.

information could be neglected. The time penalty can be reduced by carrying out the refinement procedure using just the connectivity information, but then checking any resulting subgraph isomorphisms for bond equivalence at the end of the search.

The main inefficiencies of Algorithm I are

Many of the PEs are wasted since the processing involves data structures of size $P_\beta \times P_\beta$ or less, rather than of size 64 × 64 (or of size 32 × 32 on a DAP 510).

The processing makes extensive use of control, shifting, and matrix operations, which are slower on a DAP than the basic Boolean operations that underlie Ullmann's original algorithm.

Algorithm II. The first approach utilizes inner loop parallelism to compare a query with a single database structure, requiring a detailed analysis of the basic matching algorithm to identify how it can be distributed across the array of PEs. The second approach utilizes outer loop parallelism, in which we seek to compare a single query with many database structures simultaneously. Thus, inner loop parallelism involves distributing the basic matching algorithm across the available processors, while outer loop parallelism involves distributing the database of different molecules across the available processors. The second approach is much simpler in concept, since, in essence, it involves using just the basic sequential algorithm, but with it being executed on different database records simultaneously with a different database structure being allocated to each of the PEs. This has the great advantage that up to 4096 database structures can be processed in parallel; however, the SIMD nature of the DAP means that rapid searching will be achieved only if it is possible to prevent most of the PEs being idle for most of the time.

Algorithm I involves a *horizontal mapping* of the data structures, in that the matrices **A**, **B**, and **M** are mapped onto a horizontal plane of the DAP's array of PEs, with one element of a matrix in each PE. This mapping allows all of the elements that comprise a matrix to be processed in parallel in a single step. Algorithm II involves a *vertical mapping*, in which all of the data associated with a particular structure must be stored in the local memory of a particular PE. The difference between these two types of mapping is illustrated in Figure 3. The mapping of arrays onto the memory of the DAP can be described by *mapping vectors* which associate array indexes with physical addresses within the DAP array store. The parallel data transform (PDT) library is a set of procedures, accessible within Fortran Plus, which allows data to be moved between certain regular mappings. Such PDTs provide an extremely powerful and efficient method of moving data between PEs. In the work reported here, a PDT was used to remap the **B** and **M** matrices for 4096 structures from the horizontal mode in which they had been read into the DAP into vertical mode for processing by Algorithm II.

Algorithm II involves executing the basic Ullmann algorithm, as described under Ullmann's Subgraph Isomorphism Algorithm, in each PE at the same time, subject to the status of masks that determine whether or not processing is to be carried out in a particular PE at a particular time. The flow

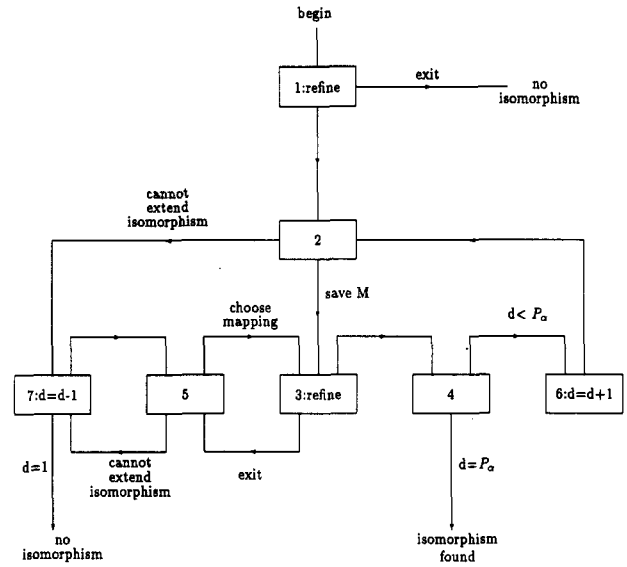

Figure 4. Flow of control in the Ullmann algorithm. The numbers in each box correspond to the main steps of the algorithm as detailed in Ullmann's original paper; d is the current level in the search tree.[13]

of control in the Ullmann algorithm is shown in Figure 4. When the Ullmann algorithm is implemented on a sequential machine, there are very many paths that can be followed, depending on the characteristics of the particular query substructure and database structures that are being processed. For example, with one database structure, a complete mismatch may be identified after the very first call of the refinement procedure; another database structure, conversely, may not identify an incorrect assignment of a query atom to a database atom until all but one of the query atoms have been mapped, so that backtracking will be required to consider alternative mappings for the P_α-th query atom. This is not a problem when only a single structure is being processed but is an important issue when many structures are being processed in parallel, since each structure may require a different path to be followed in the search tree.

In an SIMD machine, there is, necessarily, only a single control unit and thus a *superset code* is needed. This deals with all of the possible cases: a good superset code will be one that achieves high parallelism. A logical mask is used for each step of the chosen path to record which PEs wish to execute that step, and this is used to preserve data within PEs not wishing to participate. These masks are updated as the algorithm proceeds, thus guaranteeing that each PE, and its corresponding database structure, chooses the correct route. An efficient implementation will be achieved if these local paths are as similar as possible to the single global path that is executed by the control unit. A high degree of similarity will result in a high degree of processor usage: this is particularly important for the expensive relaxation stage, which accounts for about 95% of the total computational effort. The global path that is followed may contain global tests, and branches when a particular process, e.g., backtracking to find a new trial assignment, is no longer required. Several methods were tested for determining the precise order in which the execution of the algorithm should take place: the most effective approach involved the use of *majority voting*.[36] Here, at some point during the processing of a query, that next step is chosen for execution that is required by the largest number of PEs at that time. The identification of this most popular step is effected efficiently by taking the logical sum of the masks that denote which step should be executed next in each PE.

The time complexity of Algorithm II is dependent, inter alia, on the sizes of the structures that are being processed. Thus,

Table I. Characteristics of the Five Small Datasets from the Medicinal Chemistry Literature Used in the Initial Experiments

dataset	P_α	P_β mean	P_β SD	P_β maximum	comments
A	10	11.9	1.2	16	28 benzohydroxamic acids[37]
B	9	25.3	8.6	36	105 benzamidines[38]
C	13	17.2	1.9	20	24 hydrazides[39]
D	12	26.4	2.9	37	79 penicillins[40]
E	20	29.6	4.9	37	46 quinazolines[41]

Table II. Execution Times (in CPU s) on an IBM 3083 [$T(S)$] and on a DAP 610 [$T(P)$], and Performance Ratios (R) for the Use of Algorithm I on Five Small Datasets from the Medicinal Chemistry Literature[a]

dataset	$T(S)$	$T(P)$		R	
A	0.56	0.10	0.21	5.37	2.67
B	7.93	0.90	1.60	8.85	4.96
C	1.87	0.25	0.45	7.47	4.16
D	2.28	0.49	1.19	4.69	1.92
E	3.56	0.44	0.92	8.15	3.87

[a] The first of the two figures for $T(P)$ and R correspond to those runs of the refinement procedure that used just the connectivity information during the refinement, with the check on bond types being executed once all of the query atoms had been mapped, and the second of the two figures to those runs that used the actual bond types during the refinement. All results are to three significant figures.

the execution time of a single step for all 4096 molecules will be determined by the time requirement for the largest structure, suggesting that the molecules being processed should all be of about the same size if search efficiency is to be maximized. This behavior is observed in the experimental results discussed under the next section. There are two further characteristics of Algorithm II that are potential sources of inefficiency:

It will make best use of the DAP's architecture if most of the PEs are utilized. This requires that at least 4096 structures (for a DAP 610, or 1024 structures for a DAP 510) have passed the initial screening search and need to undergo the atom-by-atom search.

It requires much more local memory in each PE than does Algorithm I since the algorithm has a storage complexity of order $O(P_\alpha^2 P_\beta)$; with the current codes and with the machine used here, it was not possible to search for query substructures that contained more than 16 atoms.

EXPERIMENTAL DETAILS AND RESULTS

Initial Experiments with Algorithm I. Our initial experiments used Algorithm I to carry out searches for the parent substructure on five small sets of analogues from the medicinal chemistry literature as detailed in Table I.

Two sets of runs were carried out. In the first set, the bond-type information was included in the refinement procedure; in the second set, this information was checked in potential matching structures only after all of the query atoms had been mapped. The results of these runs are listed in Table II. The main measure used is the performance ratio, R. This is defined to be $T(S)/T(P)$, where $T(S)$ is the time required for running the best available sequential algorithm on a sequential processor, and $T(P)$ is the time required for running the parallel algorithm on the parallel processor.[42] The sequential processor used in our work was the IBM 3083 BX mainframe of the University of Sheffield.

An inspection of Table II shows that checking the bond-type information after the refinement procedure has terminated is typically about twice as fast on the DAP as including this information during the refinement. All of the subsequent experiments with Algorithm I thus used the former procedure. In the case of Algorithm II, conversely, the processing of different molecules is being completed at different times, and it was simplest to check bond types during each call of the refinement procedure.

Table III. Characteristics of the Seven Sets of 4096 Structures Used in the Main Experiments

dataset	P_β mean	P_β SD	P_β maximum	comments
F	25.9	6.6	64	random selection
G	16.2	7.3	64	random selection
H	22.0	1.7	25	center of the distribution
I	16.5	0.6	17	P_β = 16 or 17
J	21.5	0.6	22	P_β = 21 or 22
K	37.5	7.7	64	$P_\beta > 30$
L	16.0	6.9	61	random selection

Table IV. Execution Times (in CPU s) and Performance Ratios (R) for the Serial Implementation of Ullmann's Algorithm and the Three Parallel Implementations, Algorithms I, II, and III[a]

dataset	$T(S)$	$T(P)$ I	$T(P)$ II	$T(P)$ III	R I	R II	R III
F	305	186	573	139	1.64	0.53	2.19
G	137	139	452	123	0.99	0.30	1.12
H	218	158	255	46.4	1.38	0.85	4.64
I	160	174	121	30.8	0.92	1.32	5.20
J	207	179	180	43.7	1.15	1.15	4.73
K	478	173	823	150	2.77	0.58	3.19
L	122	137	355	110	0.98	0.34	1.10

[a] The searches are of the seven 4096-member datasets with the set of 57 queries with $P_\alpha \leq 16$. All results are to three significant figures.

Main Experiments. The main experiments used seven sets of 4096 structures—datasets F–L—from the *ChemQuest* database, together with 57 substructure queries taken from the journal literature on substructure searching systems and substructure searching algorithms. The mean number of atoms per query was 10.2 (with a standard deviation of 3.5). Datasets F, G, and L were selected at random from the database subject to a limit on P_β of 64 atoms. The remaining datasets were selected so that all of the structures that were to be searched contained comparable numbers of atoms: this was effected by sorting molecules into order of decreasing size and then selecting sets of 4096 structures. The characteristics of these datasets are detailed in Table III.

The datasets were searched using Algorithms I and II and using the sequential implementation on the IBM 3083: the resulting run times and R values are listed in Table IV (this table also contains results for a further implementation, Algorithm III, which is discussed next). It will be seen that our current version of Algorithm I is noticeably faster than the current version of Algorithm II.

It must be emphasized that the figures in Table IV are accumulated over all 57 searches of the 4096 structures, a total of 233 472 matches for each dataset, and there are some searches where Algorithm II is very efficient in operation. The problem with this code is that the overall run time is determined by the time for the slowest match: thus, even if a large fraction of the structures has been completed, a small number of long-running matches, or even a single match, can result in this algorithm giving an extended run time for a particular query. Thus, for one of the queries, the first seven calls of the refinement procedure resulted in the completion of the processing for 0, 0, 208, 3178, 578, 100, and 9 molecules; calls 8–41 produced no further completions; and then calls 42–44 resulted in the completion of 6, 2, and the final 5 molecules. Again, for another query, calls 1–4 of the refinement procedure resulted in the completion of 0, 0, 4040, and 35 molecules; calls 24, 25, and 30 produced 12, 1, and 7 completions, respectively;

Table V. Percentage Utilization of the PEs[a]

dataset	PE utilization mean	SD	minimum	maximum	refinement invocations
F	9.7	40.9	1.73	94.5	2706
G	10.6	33.1	3.37	93.2	2297
H	11.0	33.5	1.70	95.3	2376
I	11.5	34.3	4.39	94.8	2185
J	10.8	34.5	1.73	95.1	2409
K	10.7	39.2	2.20	96.0	2520
L	12.6	34.4	4.18	94.5	1941

[a] With mean, standard deviation, minimum, and maximum values and the total numbers of invocations of the refinement procedure for each of the seven 4096-member datasets used in the mean experiments.

with the single remaining molecule completing only after call 55. A better example of the utility of Algorithm II is provided by a query where all of the molecules were processed by a total of just seven calls to the refinement procedure, these resulting in the successive completion of 0, 0, 3411, 683, 0, 1, and 1 molecules.

A measure of the overall efficiency of Algorithm II is given by the percentage utilization of the PEs, which may be ascertained by noting how many PEs are active during each call to the refinement procedure: these figures are detailed in Table V. The reader will note the great differences between the minimum and the maximum values (and thus the very large standard deviations): these serve to emphasize the differences that have been exemplified in the previous paragraph. Since the mean efficiencies are comparable, the differences in the performance ratio that are observed in Table IV must be accredited to other factors. We believe that the main factor is the inefficiency caused by a disparate distribution of structure sizes within a file. In datasets H, I, and J, the range of structure sizes is low and the best performance is achieved; a wide range of structure sizes is found in the other four datasets, and the performance is correspondingly worse.

The low processor utilization in Algorithm II suggests the need for a mechanism to decrease the number of inactive PEs: this point is addressed in the two algorithms that are discussed in the remainder of this section.

Algorithms III and IV. The deleterious effect of long-running molecules on Algorithm II suggests two further algorithms that could be adopted. The first of these combined algorithms, referred to subsequently as Algorithm III, involves two stages as follows:

1. Continue the execution of Algorithm II until some user-defined number, n, of the matches have been completed.
2. The remaining 4096 − n structures (or 1024 − n structures for a DAP 510) are then processed in sequence using Algorithm I.

For large files of structures, these two steps would be repeated for each set of 4096 structures. Tests with this integrated strategy suggest that its overall efficiency is dependent on the value of n, rising steadily as n was increased from zero with increments of 50, and then leveling off. The optimal values of n that were obtained for the seven datasets were all in the range 3300–3900 (thus emphasizing the relatively small number of structures that adversely affect the overall performance of Algorithm II). The results of using Algorithm III are included in Table IV, where it will be seen that this combined strategy always gives a better level (and sometimes a substantially better level) of performance than either of the two individual algorithms.

There is an alternative, but rather more complex, way of improving the performance of Algorithm II that uses a processor farm technique:

1. Continue the execution of Algorithm II until n of the matches have been completed.
2. Suspend processing and load n new structures into those PEs where processing has been completed.
3. Repeat steps 1 and 2 until all of the structures that are to be searched have been completed.

The complexity here lies in the need to refill the PEs rapidly. The parallelism is maximized with a small value of n, but this is at the cost of the frequent loading of new structures; with large n, the efficiency of the algorithm will approach that of Algorithm II. This algorithm, Algorithm, IV, is expected to give the highest level of performance of the four individual algorithms when large files need to be processed but has not been implemented to date.

DISCUSSION

In this paper, we have discussed the use of a massively parallel processor array, the DAP, for the implementation of an atom-by-atom searching procedure based on Ullmann's subgraph isomorphism algorithm. The current inner loop parallel implementations of this algorithm are faster than the current outer loop parallelism implementation, though superior results are obtained in some cases from a combined algorithm that uses the inner loop implementation for those structures that are difficult to process by using the outer loop implementation.

Despite the apparent superiority of Algorithm I over Algorithm II, the latter has considerably more scope for improvement, in that our initial implementation of the algorithm used a large number of looping and control functions to organize the Boolean array operations of Ullmann's procedure (the Boolean operations themselves are so fast that these overheads tend to dominate the overall time requirement). Preliminary experiments show that the overheads can be reduced by making use of some of the new features in the most recent release of the Fortran Plus compiler that enable language operations to be performed on large arrays. The compiler code that is produced to do the looping is much more efficient than user-written Fortran Plus code: performance improvements of 5–10 times have been observed when this compiler is used for other applications requiring a large amount of such "oversize" Boolean work. There are also other parts of the present code that would benefit from this compiler. In addition, it would be possible to rewrite some of the critical code sections, such as the expansion of **R** and its ANDing with **A** in the APAL assembler. It is expected that the full implementation of these techniques will increase the performance of Algorithm II by about an order of magnitude. The first technique is not applicable to Algorithm I, and the second applies less (an in different ways). Thus, the use of these techniques should result in Algorithm II being considerably faster, and comparable comments would apply to Algorithms III and IV. We thus believe that the best results would be obtained by using Algorithm IV until there were no structures remaining to be loaded into the processor; then Algorithm III would be applied to those structures that had not been completed.

It is important to note that the precise performance levels that have been obtained with the sets of *ChemQuest* structures may be significantly lower than those that would be obtained in an operational substructure searching system, where an initial screening search would be used to eliminate many of the database structures from the time-consuming atom-by-atom search. The sets of 4096 molecules used here contained many structures that had little or no resemblance to the query substructures that were searched. These structures were thus eliminated at a much earlier stage of the processing than would be the case with conventional screened output. In this respect, the performance ratios obtained with the small datasets may give a better indication of the efficiency of our parallel al-

Table VI. Execution Times (in CPU s) and Performance Ratios (R) for the Serial Implementation of Ullmann's Algorithm and for Algorithm I[a]

dataset	$T(S)$	$T(P)$	R
F	233	57.7	4.04
G	42.8	12.1	3.55
H	89.3	30.4	2.94
I	2.98	2.69	1.11
J	75.5	24.5	3.09
K	302	54.8	5.52
L	18.7	6.99	2.68

[a] The searches are of the seven 4096-member datasets with the set of seven queries with $P_\alpha > 16$. All results are to three significant figures.

gorithms, since most of these structures were analogues having a large degree of commonality with each other and with the query substructure (as would normally be the case with the output from the screening stage of a substructure search). It is thus likely that the efficiency of the parallel processor, relative to the sequential processor, increases with the amount of computational work that is required. Evidence to support this suggestion comes from searches of the datasets that used seven further queries for which $P_\alpha > 16$ and which could thus not be searched by using Algorithm II (because of the memory problem mentioned under the fourth section). The results for these searches are detailed in Table VI, where it will be seen that the performance ratios are very much greater than those detailed for Algorithm I in Table IV. It will thus be clear that the relative performances of the various algorithms is crucially dependent on the particular data that is to be processed.

To date, we have considered only the atom-by-atom component of substructure searching; however, the DAP can also be used for the initial, screening stage where the search engine must compare a query bit string with the bit strings representing each of the molecules in a database to identify that small number of molecules that should undergo the second-stage search. Bit strings are increasingly being used in text retrieval systems, where the strings are referred to as *text signatures*, and it has been shown that the bit-level organization of processor arrays provides an extremely efficient mechanism for scanning large files of text signatures.[27] The screens for a small in-house structure database could readily be fitted into the processor array's memory; in addition, both the DAP and the Connection Machine have high-speed disk units, operating at 16–40 Mbytes/s, that would allow the screens for even very large structure databases to be scanned in a very few seconds. The use of the DAP for both the initial screening and the detailed atom-by-atom stages should provide a highly effective substructure searching engine.

ACKNOWLEDGMENT

We thank the staff of the Centre for Parallel Computing, Queen Mary and Westfield College, for the provision of computing facilities; Active Memory Technology Ltd. and the Science and Engineering Research Council for funding; and Fraser Williams (Scientific Systems) for provision of the *ChemQuest* database. We thank Dr. G. M. Downs for assistance with the preparation of the datasets and Dr. J. Steel for assistance with the use of the DAP service.

APPENDIX

There follows a Pascal-like pseudo-code description of the Ullmann algorithm. The procedure 'refine' is that described under the second section, while the values of P_α and P_β are user-defined constants.

```
TYPE
  maptype=ARRAY[1...Pα,1...Pβ] OF BOOLEAN
VAR
  M: maptype
  A: ARRAY[1...Pα,1...Pα] OF INTEGER
  B: ARRAY[1...Pβ,1...Pβ] OF INTEGER
  isomorphism: BOOLEAN

PROCEDURE ULLMANN(D: 1...Pα, M: maptype)
VAR
  M1: maptype
  mismatch: BOOLEAN
BEGIN
  M1:=M
  REPEAT
    choose new unique mapping for query atom D
    update M accordingly
    refine(M,mismatch)
    IF NOT(mismatch)
      IF D=Pα THEN
        BEGIN
          isomorphism:=TRUE
          output M as an isomorphism
        END
      ELSE
        ULLMANN(D+1,M)
    ELSE
      M:=M1
  UNTIL all unique mappings for query atom D have been tried
END

BEGIN
  isomorphism:=FALSE
  ULLMANN(1,M)
END
```

REFERENCES AND NOTES

(1) Hockney, R. W.; Jesshope, C. R. *Parallel Computers 2. Architecture, Programming and Algorithms*; Adam Hilger: Bristol, 1988.
(2) Hwang, K.; Briggs, F. A. *Computer Architecture and Parallel Processing*; McGraw-Hill: New York, 1984.
(3) Quinn, M. J. *Designing Efficient Algorithms for Parallel Computers*; McGraw-Hill: New York, 1987.
(4) Hillis, W. D.; Steele, G. L. Data Parallel Algorithms. *Commun. ACM* **1986**, *29*, 1170–1183.
(5) Collins, J. F.; Coulson, A. F. W. Applications of Parallel Processing Algorithms for DNA Sequence Analysis. *Nucleic Acids Res.* **1984**, *12*, 181–192.
(6) Collins, J. F.; Reddaway, S. F. High Efficiency Sequence Database Searching: Use of the Distributed Array Processor. In *Computers and DNA: SFI Studies in the Sciences of Complexity*; Bell, G., Marr, T., Eds.; Addison Wesley: New York, 1990.
(7) Rasmussen, E. M.; Downs, G. M.; Willett, P. Automatic Classification of Chemical Structure Databases Using a Highly Parallel Array Processor. *J. Comput. Chem.* **1988**, *9*, 378–386.
(8) Rasmussen, E. M.; Willett, P.; Wilson, T.; Manson, G. A.; Wilson, G. A. Searching of Chemical Structure Databases Using Parallel Computer Hardware. *Anal. Chim. Acta* **1990**, *235*, 77–86.
(9) Wipke, W. T.; Rogers, D. Rapid Subgraph Search Using Parallelism. *J. Chem. Inf. Comput. Sci.* **1984**, *24*, 255–262.
(10) Jochum, P.; Worbs, T. A Multiprocessor Architecture for Substructure Search. In *Chemical Structures. The International Language of Chemistry*; Warr, W. E., Ed.; Springer Verlag: Berlin, 1988.
(11) Downs, G. M.; Lynch, M. F.; Manson, G. A.; Willett, P.; Wilson, G. A. Transputer Implementations of Chemical Substructure Searching Algorithms. *Tetrahedron Comput. Methodol.* **1988**, *1*, 207–217.
(12) Brint, A. T.; Willett, P. Identification of 3-D maximal common substructures using transputer networks. *J. Mol. Graphics* **1987**, *5*, 200–207.
(13) Ullmann, J. R. An Algorithm for Subgraph Isomorphism. *J. ACM* **1976**, *16*, 31–42.
(14) Read, R. C.; Corneil, D. G. The Graph Isomorphism Disease. *J. Graph Theory* **1977**, *1*, 339–363.
(15) Tarjan, R. E. Graph Algorithms for Chemical Computation. *ACS Symp. Ser.* **1977**, *No. 46*, 1–19.
(16) Ash, J. E.; Chubb, P. A.; Ward, S. E.; Welford, S. M.; Willett, P. *Communication, Storage and Retrieval of Chemical Information*; Ellis Horwood: Chicester, 1985.
(17) Lipscombe, K. J.; Lynch, M. F.; Willett, P. Chemical Structure Processing. *Annu. Rev. Inf. Sci. Technol.* **1989**, *24*, 189–238.
(18) McGregor, J. J. Backtrack Search Algorithms and the Maximal Common Subgraph Problem. *Software—Pract. Exp.* **1982**, *12*, 23–34.
(19) Brint, A. T.; Willett, P. Pharmacophoric Pattern Matching in Files of 3-D Chemical Structures: Comparison of Geometric Searching Algorithms. *J. Mol. Graphics* **1987**, *5*, 49–56.

(20) Sheridan, R. P.; Nilakantan, R.; Rusinko, A.; Bauman, N.; Haraki, K. S.; Venkataraghavan, R. 3DSEARCH: a System for Three-Dimensional Substructure Searching. *J. Chem. Inf. Comput. Sci.* **1989**, *29*, 255-260.
(21) Mitchell, E. M.; Artymiuk, P. J.; Rice, D. W.; Willett, P. Use of Techniques Derived from Graph Theory to Compare Secondary Structure Motifs in Proteins. *J. Mol. Biol.* **1990**, *212*, 151-166.
(22) von Scholley, A. A Relaxation Algorithm for Generic Chemical Structure Screening. *J. Chem. Inf. Comput. Sci.* **1985**, *25*, 235-241.
(23) Flanders, P. M.; Hunt, D. J.; Reddaway, S. F.; Parkinson, D. Efficient High Speed Computing with the Distributed Array Processor. In *High Speed Computers and Algorithm Organization*; Kuck, D. J., Lawrie, D. H., Sameh, A., Eds.; Academic Press: New York, 1977.
(24) Hunt, D. J.; Reddaway, S. F. Distributed Processing Power in Memory. In *The Fifth Generation Computer Project*; Pergamon Infotech: London, 1983.
(25) Parkinson, D. The Distributed Array Processor (DAP). *Comput. Phys. Commun.* **1983**, *28*, 325-336.
(26) Parkinson, D.; Litt, J., Eds. *Massively Parallel Computing with the DAP*; Pitman: London, 1990.
(27) Willett, P.; Rasmussen, E. M. *Parallel Database Processing. Text Retrieval and Cluster Analysis Using the Distributed Array Processor*; Pitman: London, 1990.
(28) Unger, S. H. A Computer Oriented towards Spatial Problems. *Proc. Inst. Radio Eng. (USA)* **1958**, *46*, 1744-1750.
(29) Slotnick, D. L.; Borck, W. C.; McReynolds, R. C. The SOLOMON computer. *AFIPS Conf. Proc.* **1962**, *22*, 97-107.
(30) Potter, J. L., Ed. *The Massively Parallel Processor*; MIT Press: Cambridge, MA, 1985.
(31) Hillis, W. D. The Connection Machine. *Sci. Am.* **1987**, *256* (6), 86-93.
(32) Tucker, L. W.; Robertson, G. G. Architecture and Applications of the Connection Machine. *Computer* **1988**, *21* (8), 26-38.
(33) Reddaway, S. F. Achieving high performance applications on the DAP. In Jesshope, C. R. *CONPAR 88*; Jesshope, C. R., Reinartz, K. D., Eds.; Cambridge University Press: Cambridge, 1989.
(34) Moitra, A.; Iyengar, S. S. Parallel Algorithms for some Computational Problems. *Adv. Comput.* **1987**, *26*, 93-153.
(35) Cringean, J. K.; Manson, G. A.; Willett, P.; Wilson, G. A. Efficiency of Text Scanning in Bibliographic Databases Using Microprocessor-Based, Multiprocessor Networks. *J. Inf. Sci.* **1988**, *14*, 335-345.
(36) Wilson, T. Ph.D. Thesis; University of Sheffield (in preparation).
(37) van't Riet, B.; Kier, L. B.; Elford, H. L. Structure Activity Relationships of Benzohydroxamic Acid Inhibitors of Ribonucleotide Reductase. *J. Pharm. Sci.* **1980**, *69*, 856-857.
(38) Hansch, C.; Yoshimoto, M. Structure-Activity Relationships in Immunochemistry. 2. Inhibition of Complement by Benzamidines. *J. Med. Chem.* **1974**, *17*, 1160-1167.
(39) Richard, A. J.; Kier, L. B. Structure-Activity Analysis of Hydrazide Monoamine Oxidase Inhibitors Using Molecular Connectivity. *J. Pharm. Sci.* **1980**, *69*, 124-126.
(40) Adamson, G. W.; Bawden, D. A Method of Structure-Activity Correlation Using Wiswesser Line Notation. *J. Chem. Inf. Comput. Sci.* **1975**, *15*, 215-220.
(41) Chen, B. K.; Horvath, C.; Bertino, J. R. Multivariate Analysis and Quantitative Structure-Activity Relationships. Inhibition of Dihydrofolate Reductase and Thymidylate Synthetase by Quinazolines. *J. Med. Chem.* **1979**, *22*, 483-491.
(42) Parkinson, D.; Liddell, H. M. The Measurement of Performance on a Highly Parallel System. *IEEE Trans. Comput.* **1983**, *C32*, 32-37.

Computer Storage and Retrieval of Generic Chemical Structures in Patents. 11. Theoretical Aspects of the Use of Structure Languages in a Retrieval System

WINFRIED DETHLEFSEN
BASF, Ludwigshafen/Rhein, Germany

MICHAEL F. LYNCH,* VALERIE J. GILLET, GEOFFREY M. DOWNS, and JOHN D. HOLLIDAY
Department of Information Studies, University of Sheffield, Sheffield S10 2TN, England

JOHN M. BARNARD
Barnard Chemical Information Ltd., 46 Uppergate Road, Stannington, Sheffield S6 6BX, England

Received November 16, 1990

A rational basis for discussion of issues relating to the storage and retrieval of generic chemical structures is developed in this paper and those which follow. It rests on well-known logical and linguistic foundations, and seeks to establish a consistent conceptual framework for considering generic structures as they occur in patents and as represented for storage and retrieval in information systems. The syntax, semantics, and pragmatics of chemical structure languages, in general, are described, together with the meaning-relations between the notation, the intension, and the extension of a structural expression. Development of this basis provides a framework for considering issues of the representation of generic structures in formally defined languages, such as GENSAL, together with the process of translation from chemists' language into GENSAL, the surface language, and of further translation into other internal representations, including the ECTR (Extended Connection Table Representation), and ring, fragment, and reduced graph screens for processing and searching. The question of the definiteness of structure representations and its consequences for searching are discussed, together with formal properties of structural expressions in the GENSAL system, and the applicability of a variety of algorithms. Finally, the relations between query and file structure languages are described.

1. INTRODUCTION

The representation of generic structures for retrieval poses complex questions, many of which have already been the subject of publications from Sheffield, e.g., by Downs et al.[1] and preceding articles in this series, and elsewhere, in particular by Shenton et al.[2,3] and by Fisanick.[4,5]

The importance of the area for chemical industry is reflected in the introduction of the Markush DARC system in 1989 by Derwent Publications Ltd., with Questel S.A. and INPI (the French Patent Office), and of the MARPAT system in 1990 by Chemical Abstracts Service. International Documentation in Chemistry GmbH (IDC) is also active in innovation, creating a database of generic structures in GENSAL in order to generate GREMAS fragment codes automatically.[6]

GENSAL, first described by Barnard et al.,[7] is an artificial and formally defined language, the purpose of which is to record chemical structure information in patents in such a way that the resultant expressions are amenable to algorithmic manipulation in order to support a variety of types of structure-based searches. Its use involves translations from the language used by chemists, patent agents, and lawyers in patent documents into the formally defined language. GEN-

0095-2338/91/1631-0233$02.50/0