ADAPTATION OF A LARGE-SCALE COMPUTATIONAL CHEMISTRY PROGRAM TO THE INTEL iPSC CONCURRENT COMPUTER

Allan Roger Larrabee
B.S., Bucknell University, 1957
PhD., Massachusetts Institute of Technology, 1962

A thesis submitted to the faculty
of the Oregon Graduate Center
in partial fulfillment of the
requirements for the degree
Master of Science
in
Computer Science & Engineering

August, 1986

The thesis "Adaptation of a Large-Scale Computational Chemistry Program to the Intel iPSC Concurrent Computer" by Allan R. Larrabee has been examined and approved by the following Examination Committee:

Robert Babb II
Associate Professor
Thesis Research Advisor

Richard Kieburtz
Professor

Dan Hammerstrom
Associate Professor

Frank Hauser
Professor

ACKNOWLEDGEMENTS

Dr. Robert Babb provided the opportunity and he is a shining example of the aging breed of talented "Fortran Fossils". His encouragement to work on a problem that is personally exciting motivated this work. Anywhere from a few to hundreds of suggestions and encouragement were provided by the following...

Dr. Frank Momany -- Polygen Corporation, Waltham, MA.

Dr. Rebecca Jones -- Department of Chemistry, UCSD, CA.

Tony Anderson -- Intel Scientific Computers, Beaverton, OR.

Sandy Arnold -- Warn Industries, Milwaukie, OR.

I am indebted to the Intel Scientific Computers corporation for financial support toward the end of this work.

TABLE OF CONTENTS

List of Figures v

Abstract vi

1. Introduction 1

2. Rational and Early Studies 5

3. Development of the Program 18

4. Results 33

5. Performance Measurements 37

6. Conclusions 44

References 47

Appendix A: ECEPP86 User's Guide 49

Appendix B: Prettyprint 57

Appendix C: A Peptide Primer for Non-chemists 59

Biographical Note 63

LIST OF FIGURES

1. Transmission Rate Between Nearest Neighbors Versus Message Length 6
2. Transmission Rate (Two Connections) Versus Message Length 7
3. Time for Message Sending (Blocked) 8
4. A Non-optimized Program Calculating Pi 9
5. An Optimized Program Calculating Pi 11
6. Data Flow View of ECEPP86 Loaded on the iPSC 20
7. Pseudocode for Host Message Control 29
8. Host Program Message Passing Code 32
9. Profile of Integration Program 38
10. Comparison of Three Computer Systems for a Benchmark.... 42

ABSTRACT

A study was made of some of the characteristics, capabilities, and limitations of the iPSC concurrent computer manufactured by the Intel Corporation. Initial experiments with test programs measured the large amount of time required to send and receive messages between nodes and between the cube manager and the nodes. Programs adapted to run concurrently will have the greatest speedup over the same program executed serially if the computational time is large relative to the time spent passing messages. A large-scale computational chemistry program (named ECEPP83) that calculates the global minimum energy of peptide structures (a peptide is a small protein) was ported and adapted to execute on the iPSC computer. The data entry and checking portion of the original code was ported to the 286/310 Intel computer that serves as a manager of the 32 to 128 CPU's (nodes) of the iPSC. The data for each structure is sent by the manager to a separate node which reports its results back to the host or system manager and then is assigned another structure. This adaptation is able to concurrently minimize the energy for 32 chemical structures a maximum of approximately 17 times faster than the same data can be utilized serially on a VAX 11-780 computer. A user manual was written to assist the user in assembling the input data file.

1. INTRODUCTION

As computer hardware becomes faster, larger and more complex computational problems are being addressed. Many computer scientists believe that present machines based on "Von Neumann" architecture are approaching a theoretical limit for the speed obtainable with computers having only one central processing unit (CPU). One approach to solving this problem is to build machines with multiple CPU's that can attack various aspects of a problem concurrently. Such parallel architectures offer the potential for achieving more computational operations per unit time, but also introduce new problems , such as determining which aspects of a serially executed algorithm are amenable to parallel computation, coordinating the results of the various CPU units, resolving race conditions, and reducing contention for resources [1]. The granularity of parallelism can vary all the way from fine-grained parallelism, such as a single executable statement, to an entire subroutine or several subroutines. Although it is possible for compilers to detect some opportunities to calculate "do loops" and array manipulations in parallel, in general, the potential for parallelism can be difficult to detect when utilizing most currently available programming languages. Thus, in order to move into parallel computations, either other languages more suited to concurrency must become popular [2], or existing code must be restructured [3,4]. For certain applications, modified

architectures and approaches may be required for optimal performance [5].

At the present time, the Oregon Graduate Center (OGC) has one of the first commercially available parallel computers, viz., an Intel Personal Super-Computer (iPSC)[1]. It consists of 32 microcomputers referred to as nodes. There is a total of 16 Mbytes of memory evenly divided but not shared between the 32 nodes. Each node has a copy of a small operating system, an Intel 80287 numeric coprocessor, and an Intel 80286 CPU. There are eight communication channels per node: seven internode Intel 82586 communication coprocessors and one "global ethernet" channel for communication with the cube manager discussed below. Each node can serially simulate the parallel execution of several independent processes (the actual number is limited by available memory). The process switching granularity is 50 msec. All nodes are identical and are connected by bidirectional links in a hypercube topology. In a 32 node basic hypercube unit of 4 nested cubes, each node is directly connected to 5 nearest neighbors. In larger configurations, in general, each node is directly connected to "d" nearest neighbors, where the hypercube has 2 to the power of d nodes. Although the OGC machine (d = 5) consists of a single unit of the specifications just described, the architecture allows expansion up to two or four units (64 or 128 nodes). Hypercube interconnections for a 32 node machine are physically implemented via backplane connections. Machines consisting of two or four 32 node machines are interconnected via external cables.

---

[1]Xenix is a trademark of Microsoft Corp., Unix of AT&T, and iPSC of Intel Scientific Computers.

Processes communicate with processes on the same or neighboring nodes by sending and receiving messages. Message passing can be "blocked" or "unblocked". A blocked message delays execution until the message is sent, the execution of statements is not delayed with an unblocked message. In the latter case, a check must be made to determine whether or not a message has been sent before attempting to utilize the same communication channel for sending another message. Although unblocked messages compared to blocked ones decrease execution time, a program may generate messages faster than the iPSC machine can receive them. The node operating system will "time out" unreceived messages and cause programs to halt execution. This point is discussed further in a later section.

The collection of nodes is controlled by a system cube manager (hereafter also referred to as the "host") which is an Intel 286/310 computer. This computer has the same processors as a node, but in addition, has a floppy disk reader, a Xenix operating system, and 2 Mbytes of memory in addition to a 40 Mbyte Winchester disk. The iPSC is a true "multiple instruction, multiple data" (MIMD) machine.

Programming the iPSC

At the start of this work, the following procedure was followed for programming the iPSC. Programs were compiled and the object modules for each process were bound together into one bound module. The node configuration utility (NCU) is an interface program that creates an executable system image

file which contains the user process, the node operating system, and the communication software. The user creates an NCU specification file which identifies the processes, specifies stack size and number of channels, and names the executable process. Invoking and running the NCU program is a time consuming process even for a small program. If it is necessary to recompile a program (e.g. after debugging), the NCU utility is re-invoked, but there are options which avoid repeating parts of the NCU process and thus successive invocations of the NCU are less time consuming.

Toward the end of this work, the static loading NCU process was replaced by a dynamic loading one. In this procedure, a dynamic loader installs the operating system and communication software on the nodes initially and this step need not normally be repeated. A new operating system command named "loadkill" removes any programs operating on the nodes and reinitializes them while leaving the installation of the operating system and communication software intact. The programs are compiled and bound as before, but then can be loaded on to the nodes directly or even dynamically by system calls from the program running on the cube manager. The total time required has been reduced to less than 50% of that formerly required.

2. RATIONALE AND EARLY STUDIES

Since the iPSC consists of 32 nodes the best speedup attainable for the time to solve a single problem would be 32 times the speed of a machine with only a single node. We will see that although such a speedup is approachable, the usual case is somewhat less than the maximum. For a program to take maximum advantage of the iPSC, it must require enough CPU time to justify the extra time spent in initializing and loading the nodes, and especially in coordinating the computation on the nodes via message passing. Messages between any two nodes in a hypercube topology pass through an average of d/2 internode connections, where "d" is the power of two that is equal to the total number of nodes (d is equal to 5 for a 32 node machine). Messages can be up to 16K (K=1024) bytes long, but are sent in 1 Kbyte packets and then reassembled after reception. Early test programs executed on the OGC's 32 node iPSC have shown that message passing between nodes is time consuming, and message passing from the nodes to the cube manager is even slower. Moreover, the time depends upon the number of internode connections a message must pass over to reach its destination. The results of a test program which sends a series of messages of various lengths between neighboring nodes (nodes 0 and 1) or between nodes that share a common neighbor, i.e. messages that must pass over two internode connections (node 0 and 3), are shown in figure 1.

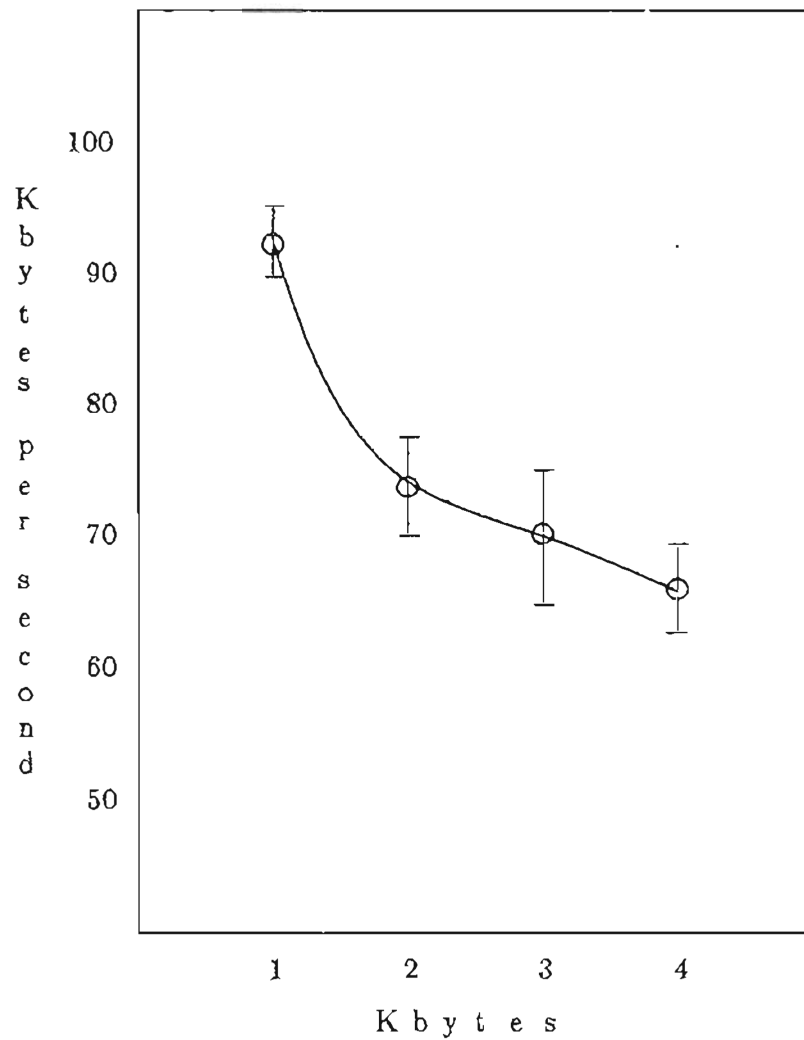

Figure 1.

Transmission Rate Between Nearest Neighbors Versus Message Length

The message transmission rate between nearest neighbors in Kbytes per second is plotted against the message length in Kbytes. The error bars in the figure represent one standard deviation. Clearly, messages to a neighbor are implemented by a different mechanism than those that must pass over more than one internode connection (figure 2).

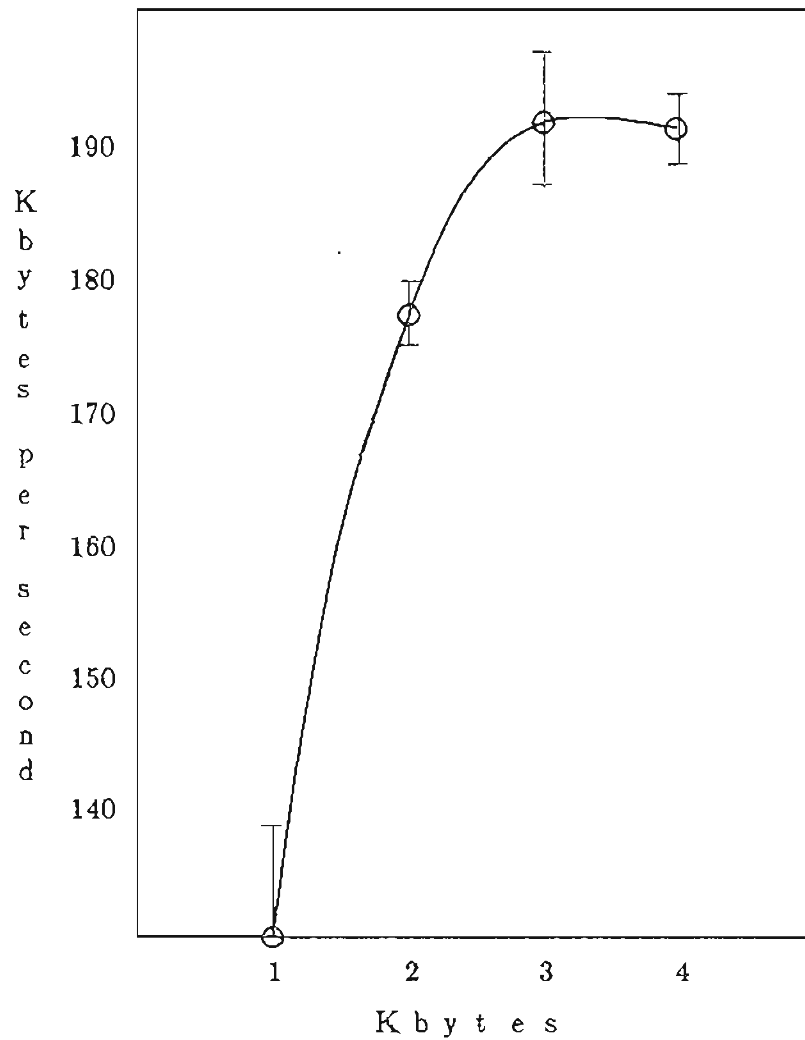

Figure 2.

Transmission Rate (Two Connections) Versus Message Length

As the need for forming 1Kbyte packets increases, the transmission *rate* for the former type decreases and the latter *rate* increases. In both cases, the time to send a message is relatively slow. The time for nearest neighbor messages of 1 Kbyte is 8.0 ± 0.5 msec per message and for a 4 Kbyte message is 23.0 ± 1.4 msec per message. These figures do not illustrate a transmission time anomaly

reported by the Intel corporation. Partially full multi-packet messages with a trailing packet of less than 500 bytes show longer transmission times than larger messages with the same number of packets, but the trailing packet is larger than about 500 bytes. This anomaly appears only for messages transmitted over at least two internode connections.

Internodes	Message Length(Kbytes)	Time(msec)
1	1	8.0 ± 0.5
1	2	11.5 ± 0.1
1	3	16.0 ± 0.5
1	4	23.0 ± 1.4
2	1	11.1 ± 0.3
2	2	27.3 ± 1.2
2	3	43.5 ± 3.1
2	4	61.3 ± 3.5

Figure 3. Time for Message Sending (Blocked)

For example, as shown in figure 3, a 1024 byte message from node 0 to node 3 (2 internode connections) required 11.1 msec to send. A message only 4 bytes longer (1028 bytes) required 64.2 ± 1.0 msec. As will be seen in the "Performance Measurements" section, the node to host messages are much slower than the node to node messages. Because of this high overhead of message passing, the creation of too many parallel process, (e.g. creation of slave nodes to attempt to ease the bottleneck of a rate-limiting subroutine), can actually increase the throughput time of the program relative to that of one with a smaller number of slave nodes.

Another test program was written which computes π by applying the rectangular rule on the integral....

$$\int_0^1 \frac{4}{(1+x^2)}$$

The user is asked to input the number of intervals, thereby fixing the size of each dx in the integration by the delta method.

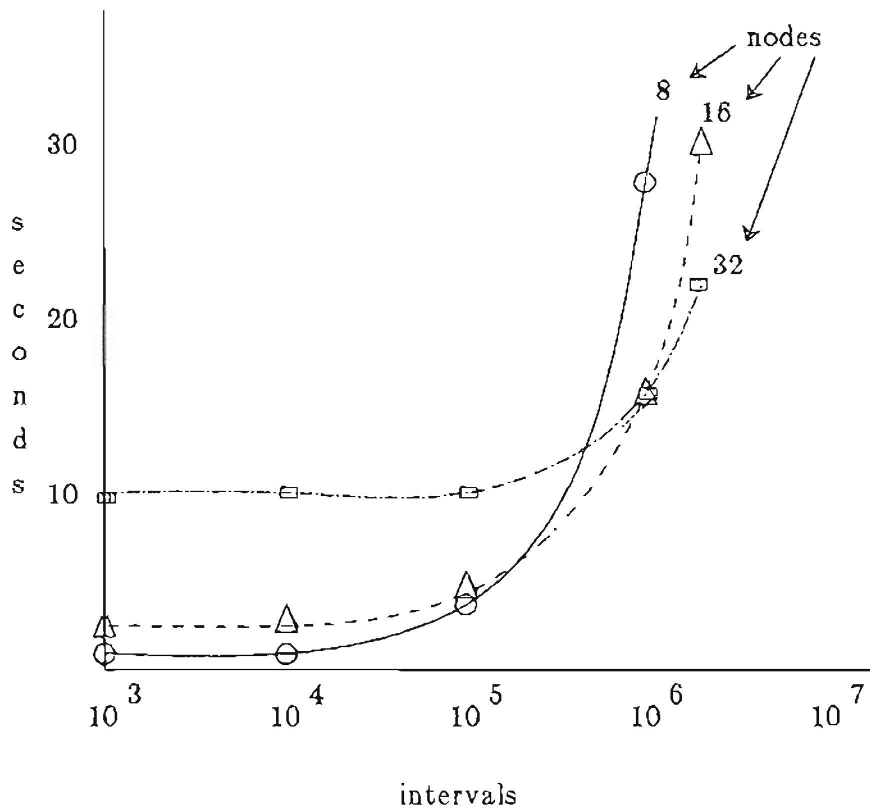

Figure 4. A Non-optimized Program Calculating Pi

The program divided up the task into as many sub-integrations as there are nodes available. The host then assigned a segment of the integration to each of the available nodes. As mentioned, these host to node messages are slower than node to node messages. Node 0 was a master node which received the

data from the other nodes (node to node messages) and after adding on its own contribution, sent the results back to the host in a single message. The program was run utilizing a total of 8, 16, and 32 nodes and the results are shown in figure 4. The most dramatic result shown in the figure is that for a low number of intervals, the *fewer* the nodes available, the *faster* the program executes. More nodes require more time consuming messages to be sent. This program sends 2n messages where n is the number of nodes available. Half of the messages are host to node and half are node to node. As the number of intervals increases, the ratio of computational time to time spent passing messages increases. The number of messages sent remains constant and as the number of intervals requested rises, the *greater* number of nodes available, the *faster* the program executes. At this point, the availability of nodes begins to pay dividends of decreased time required. When writing code for this program, the efficiency of message passing from the host to the nodes and vice versa was not optimized by the use of a minimum cost spanning tree arrangement of message passing. The host sent a message to each available node. In the next test program [6] the host only sends a single message with all the data to node 0. This node then assigns the intervals to the remaining nodes and collects the data as before. Moreover, the node to node communication between each node and the master node 0 is via a minimum cost spanning tree. When this was done and the timing measurements repeated, the results shown in figure 5 were found. At a low level of intervals, all three programs required less than 1.5 seconds to run. The curves cross over each other near 10000 intervals rather than over

100000 intervals as in figure 4. Apparently, for this algorithm, the efficient implementation of message passing and the elimination of all but two host-node messages can save up to about 8.5 seconds (10 seconds for the 8 node run of figure 4 minus 1.5).

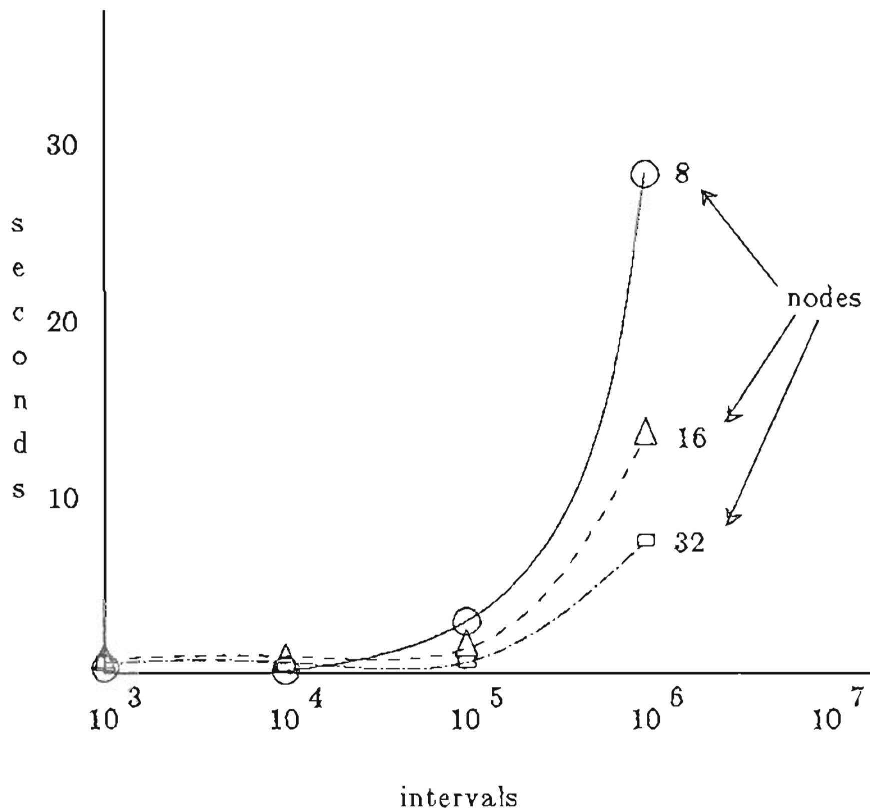

Figure 5. An Optimized Program Calculating Pi

A very important observation emerges from these experiments. If the number of messages is constant, the fraction of time devoted to message passing decreases as the computational load increases. Message passing optimization is not worth the effort unless the number of messages required is large or the time factor is dominant (or both).

Thus, for programs to be suitable for the iPSC, they must require relatively large amounts of processor time, must be intrinsically "parallelizable", and most important, must have a high ratio of computational operations to message passing. A key program design consideration for making effective use of the iPSC then is to separate out computational tasks that require much time, are parallelizable and also represent a significant fraction of the total effort of the program. Again, early test programs demonstrated that the ratio of executed instructions to messages sent (or received) must be large to justify the extra overhead involved in message passing. Just how large is a function of the length of the message, the number of node to node connections the message must pass through, and whether the messages are node to node or between the host and a node.

Additional test programs demonstrated that if messages were generated at a rate faster than they could be received, the system would "hang" as apparently the node buffers were filled to capacity. Error handler messages arising from the node operating system indicated that messages were being generated faster than the nodes could receive them. The explanation for this is that the node operating system (NOS version 2.0) does not do any flow control. Nodes are not able to alter running programs with the goal of letting the number of unreceived messages decrease. Similarly, many messages arriving at the host can cause the message reception apparatus to become costive and system messages indicate that messages may "time out" if they are not received within a certain time span. As will be seen, this flow control problem arose

during the implementation of the program for this work.

A major parameter in parallel processing is the size of the "granule" of computation that is executed in parallel. In traditional programs for Von Neumann computers, a granule corresponds to an entire application program, and little parallelism within an application program can be exploited. On the other hand, in most dataflow work to date, the grain size chosen for parallel scheduling has been at the level of a single arithmetic or logical operator [3,4]. If analysis of a program shows that one subroutine accounts for 50% of all the CPU time, the theoretical limit of improvement, if this subroutine can in fact become a "granule" and run in parallel, is two fold--ignoring the overhead of parallelization.

These early studies and others to be mentioned demonstrate that each node of the iPSC hypercube is inherently about 2 times slower than the VAX 11-780. The program chosen for the present work ran twice as slow on a single node of the iPSC compared to the time on the VAX. Thus, a 100% improvement of program execution would bring the performance of the hypercube and 11-780 close to the same value. Such an improvement would not justify the effort required to get it running on the iPSC--a program with much more speedup potential was required in order to maximize the advantages of the hypercube. Therefore, a search was made for a scientific application program that 1) is in current use, 2) is computationally intense rather than input-output intense, 3) requires enough execution time on a VAX class machine to justify an improvement effort, and 4) is amenable to parallel execution. Such a program

could then be utilized to study concurrent computations in general, and the advantages and disadvantages of a particular hardware approach to current computers, viz., the Intel iPSC hypercube.

After some searching among chemistry programs, an ideal candidate for meeting the above goals was found--a program that computes the energy of peptides (a peptide is a small protein) as a function of the three-dimensional conformation (i.e. structure of the molecule) utilizing rigid geometry calculations [7,8]. The program used for experiments with the iPSC is the property of the Polygen Corporation, Waltham, Massachusetts and is called ECEPP83 (Empirical Conformational Energy Program for Peptides-1983 edition). An earlier version, which does not have the minimizer, is available through the "Quantum Chemistry Program Exchange"[9]. This program has been used extensively to calculate the global energy minimum of peptide hormones and peptide effector molecules [7,8]. Chemists have made the case that the native structure or conformation of proteins and peptides is the one(s) with the lowest free energy. ECEPP83 and earlier versions enjoy relatively wide use since small peptides are usually not crystallizable and thus their structures are not determinable by Xray crystallography. Moreover, spectroscopic methods at present can only yield vague attributes of structures as complex and variable as peptides. In many cases, computational methods are the only means available to study this problem. ECEPP83 has been very successful for peptide structure-function studies and also for suggesting useful analog structures which will have a subset of the properties of the original molecule. Previously,

analogs have been synthesized by making substitutions without much knowledge of the mechanism of the functionality of the molecule. This approach has not been very successful. Rather, ECEPP83 assists the selection of substitutions based on three dimensional structure and the location of various chemical groups--choices made after the use of ECEPP83 has contributed to the understanding of the structure-function relationships of the molecule.

The search for a global minimum for one peptide can easily require many hours, even days, of VAX speed CPU time [7 and the references quoted there]. The order (O) of complexity for the minimum algorithm embodied in ECEPP83 is nr^2, where n is the number of conformers whose energy is sought and r is the length of the peptide, i.e. the number of amino acid residues (monomers) that make up the linear peptide chain (the polymer). The present version is approximately 5100 lines of Fortran77 code and was originally written for the IBM 360. The program has evolved through versions for the IBM 370, Xerox Sigma 9, the Univac 1100, and finally the Digital Equipment Corporation VAX 11-780. This history may partially account for the modest amount of documentation that accompanies the code.

Use of the program requires the user to define an initial structure for a particular peptide whose structure is to be calculated. This is done by defining an initial value for each dihedral angle of the molecule. The user then specifies the number of passes through the minimizer and what angles are to be allowed to change in the subsequent energy minimization algorithm. Lastly, the number of cycles through the program is chosen and from this point on the

program requires no I/O until a local minima is found. During each cycle the algorithm changes each dihedral angle at least once and calculates the energy interaction between each pair of atoms. If the energy rises, the angle is changed in the opposite direction (i.e. rotated opposite to the first change). If the energy decreases, rotation and calculation continues in the same direction until a minima is found. This process is repeated for each angle. The whole process is one cycle and the user may opt for the number of cycles to be calculated.

Initial studies on ECEPP83 (with gprof[10] on the 11/780) have showed that 33 to 40% of the CPU time was spent in calculating square roots! Moreover, there were many loop functions which had the potential for concurrency. The energy interactions between pairs of atoms are independent and therefore could possibly be calculated on separate processors. On the other hand, each such calculation would require a message to be sent back to the host, or sent to a node dedicated to data collection and task assignment. However, a consideration of how ECEPP83 is used by peptide chemists led to the possibility of an ideal solution to this problem. ECEPP83 is an iterative energy minimizer, but it is not able to escape from local minima. In other words, it travels down into an "energy valley"--but there is no assurance the the valley is the one of lowest global "elevation". Users must supply many starting conformer structures, minimize each one, and hope that they have hit the lowest valley or energy minima. This corresponds to feeding a series of structures to a "Von Neumann" computer. The approach taken here is to load the entire ECEPP83

program on every node of the iPSC concurrent computer and thus be able to calculate up to 32 structures concurrently.

The input and data checking part of ECEPP83 are run in the cube manager. The user creates and edits an input file which the cube manager reads serially. The necessary arrays are initialized with the information for a conformer. The cube manager maintains and updates, via messages from the nodes, an array which stores information as to the availability of nodes. The cube manager reads the array to find a node that is not presently assigned a conformer, sends the data for a conformer, and looks for a node for the next conformer data, etc. The starting conformers are generated by stochastic methods or by intelligent guesses and/or prior knowledge from other sources (e.g. from spectroscopy).

3. DEVELOPMENT OF THE PROGRAM

By implementing the major part of the ECEPP83 program on every node of the hypercube, the best match between the serial use of the program and the inherent advantages of the iPSC hypercube would be obtained. The effect of the slowness of the message passing mechanism would be minimized. Stated another way, since this algorithm is essentially used in a SPMD (Single Program Multiple Data) mode, implementation of the program on every node is close to the ultimate in large grain parallelism, i.e. a collection of computers under the control of another computer.

The initial task was to port the entire program onto the cube manager. The program as obtained from the Polygen Corporation ran correctly on the OGC VAX 11-780 with no modifications. Getting the code to run on the The Intel 286/310 computer that serves as the host for the 32 nodes of the iPSC was another matter. The available compiler (ftn286) lists as "ERRORS" several FORTRAN constructions for which the Unix 4.2 f77 compiler merely issues a "WARNING". Thus, the COMMON, EQUIVALENCE, and BLOCK-DATA statements were adjusted to conform to the more exacting standard of the ftn286 fortran compiler. All COMMON statements must be the same length, the dimensions of arrays in EQUIVALENCE statements must be explicitly stated (i.e., they are not 1 by default), and there can only be one

BLOCKDATA statement in the main program, not more then one. Some comments needed to be moved since the ftn286 compiler reads 132 space lines and the f77 compiler reads 72 space lines. Advantage was taken of the length limit of 31 for variable names allowed by ftn286 compared to the miserly 6 allowed by f77. Since the ftn286 compiler does not support multiple ENTRY points to a subroutine, this function was emulated by conditional GO TO statements. As a result of this, calls to these entry points in some cases had to be supplemented with dummy arguments since different ENTRY points do not require the same argument list or even that argument lists be the same length. Finally, there were calls to ENTRY points in two subroutines that entered the subroutine at the beginning and exited the subroutine before any executable statements. Comments found in the code suggested that such calls were based on a particular compiler on another machine and somehow optimized subsequent calls to the subroutine. Such unnecessary calls were eliminated from the code. The changes to COMMON and EQUIVALENCE statements comply with the ANSI 1978 standard [11] whereas the changes regarding the length of a line, ENTRY statements, and BLOCK DATA statements represent deviations from the ANSI standard.

The name of one subroutine had to be changed (DECIDE to DECIDE_IT) since it conflicted with a Xenix system library function. Similarly, an array named EPS was defined in a COMMON block and assigned a value in a BLOCK DATA statement. A real variable of the same name occurs in the main program, which does not has access to that COMMON block. The ftn286

compiler confuses these two notations whereas the f77 compiler does not. The variable name was changed to rEPS.

The final specifications and format for input are described in the user manual which can be found in appendix A. The input of amino acids is the standard three letter code used by biochemists [13] and is supplemented by additional three or four letter words for the various conformational forms of certain amino acids. The occurrence of the D isomers (the mirror image alternative position) of the amino acids is accomplished by placing a D in front of the standard three letter code. However, some of the existing amino acid codes were four letters long and thus adding a D to yield of 5 letter word would cause an input error. In addition, some different amino acid forms were represented by the same four letter code. These errors went unnoticed by previous users probably because of the scarcity of occurrence of the D isomers of the amino acids in question. New three and four letter assignments were introduced which avoid these ambiguities and are described in appendix A.

Other minor modifications were made, and two preprocessor-compiler errors were discovered and reported to Intel as part of this work: The preprocessor would only allow "include (file)" statements to be placed at the beginning of the main program. In addition, the compiler changed all upper case characters to lower case characters, even inside of a literal. The traditional name of the user defined data file was "ECPDAT" and the data file containing the bond angles and lengths for the amino acids, which is read by the ECEPP83 program initially before any user interaction, was named "TGENDATA". It was

necessary to change these file names to lower case characters. After loading the entire program, modified as described, onto the cube manager (not utilizing any of the nodes), a test program required 251 to 298 seconds to run (with no other users). The same program run on the VAX 11-780 (using gprof to yield meaningful times) required 84 to 89 seconds. Thus, for this example, the Intel 286/310 is slower than the VAX 11-780 by about a factor of three. If the nodes proved to be the same speed as the 286/310, we could expect a theoretical speedup of 32/3 or about 10 for a d = 5 iPSC hypercube (fully loaded) over that of the 11-780. This value can be compared to the actual speedup obtained after the program was implemented on the nodes.

In order to develop ECEPP83 for loading of the iterative subroutines onto each node, the program was separated into 5 separate programs for ease of editing and compiling. Programs named host.f and hostsubs.f handled user input, data checking, node tasking, and collection of the final data. These were loaded into the cube manager. Compiled and linked programs named node.f, nodesubs.f, and nodesubs2.f programs received data from the host and performed the iterative energy calculations and reported results back to the host. The final version of the host or cube manager program behaves as follows: ECEPP83 originally could obtain the data for a conformer either interactively or via an input file. The version resulting from this work is called ECEPP86 and is shown schematically in figure 6. The program obtains by DATA statements, the force field data necessary for the calculation of charge interactions. The data for amino acids angle data and bond length is read from an external

file named "tgendata". The only user input after the host program is initiated is to answer a request for the number of conformers to be input. The conformer data is then read from a user prepared input file named "ecpdat".

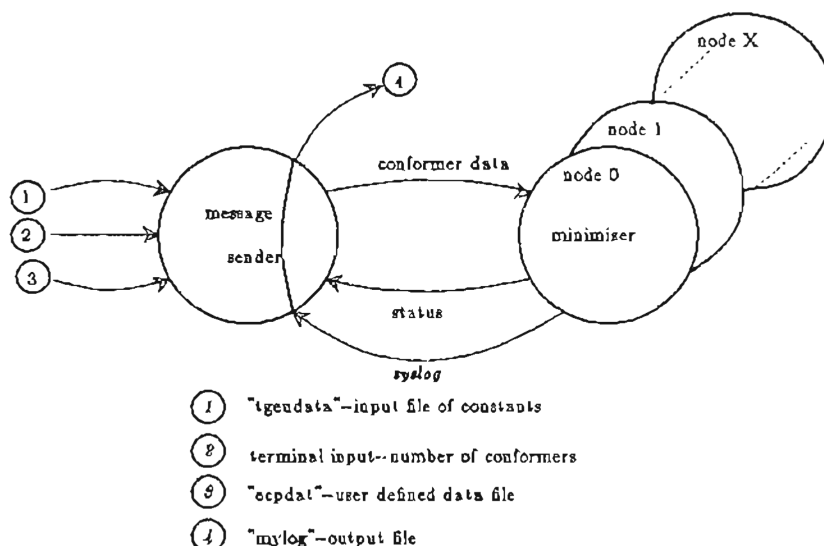

Figure 6. Data Flow View of ECEPP86 Loaded on the iPSC

If no input errors for a conformer are detected, the required data--most of which is contained in arrays and variables in COMMON statements--is sent to the lowest numbered node available. The free node is chosen after consultation of an array named "freelist" which contains the information as to the availability of nodes ready for assignment. Once a node is charged with the pertinent data for one conformer, successive iterative cycles of minimization are initiated and repeated as specified. When the nodes finish their calculations, the results are written into an output file named "mylog" via a node operating system utility named *syslog*. A call to *syslog* from a node program provides an implicit method to record data--routed through the host--into an output file. Alternatively, the initial data and the data for the minimization of

each conformer could have been managed by a master node. This node would be the only one corresponding with the host. Such an arrangement would have the advantage that 60% of the information required to define a conformer is the same for all conformers and thus this information could be sent in one message to the master node for dispersal to the nodes via a minimum cost spanning tree connection network. Alternatively, the host uses its slower message mechanisms to disperse this common information to each node separately. Aside from the loss of a "slave" node (a 1 out of 32 loss or about 3% of the computational power), there are several other problems with the master node concept in the present case. The resulting data from each node is several lines long and if the minimization data for two structures arrived at the master node concurrently, the resulting data would be interspersed. The master node could be loaded with software that separated the data from two or more incoming nodes based on the "type" variable that accompanys each message. However, the test programs mentioned alerted the possibility that such an arrangement could lead to message timeouts or program interruption if the number of messages unreceived became large. Messages incoming to the host cannot be received by type and as mentioned, the host message reception queue can overflow. It was decided not to implement a master node, rather to exit the final data via the preexisting *syslog* function which does not exhibit flow control problems. The data from two or more nodes written into the output file by *syslog* still can be interspersed and this problem was handled by processing the output file with the Xenix *sort* and *awk* functions to sort and convert

the final data to a more readable format.

The input data checking from the original program was retained. Any input error results in a diagnostic message on the screen which describes the error. The program then skips over the data for that conformer and reads the data for the next one. Therefore, input errors and errors occurring on one node can cause the loss of data for a conformer, but cannot cause the other node programs or the host program to halt. Each node performs a task that does not depend on the outcome of any other node. Rather than assign 32 jobs and ascertain which nodes have become free since the start, ECEPP86 upon detecting a node to be free, immediately reassigns that node and only assigns a job to a higher numbered node if no lower ones are now available. Since the program can be long running, it is very desirable to be able to examine the output at any time during the run. The data could be output via a message sent to the host, but this would have required an explicit receive command in the host.f code. Such an explicit send (node) and receive (host) was employed to notify the host that a node finished the calculations for a conformer and hence was ready to receive data for another. Similarly, if a node program encountered an irreconcilable error while running the iterative cycles, a *syslog* call outputs an error message and if possible, a message is sent to the manager (if the node program is still functional). This latter message is required for the manager to keep track of the number of nodes that can be expected to be producing answers versus those that are "hung" and not performing any executable statements. If the data for all conformers has been sent out, the host program must

remain "active" in order to receive the results as they are available. If the host program terminates, the node program is "timed out" after a few minutes and no more messages can be deposited in the output (mylog) file.

A script named "prettyprint" (Appendix B) was written utilizing the "sort" and "awk" tools [12] to convert the final output to a more pleasing format while removing unrelated operating system messages.

Testing showed that conformers whose energy calculations are caught in local minima, can sometimes be further minimized (jump out of local minima) by increasing the number of minimization "cycles" requested. At the start of each cycle the first derivative of the rate of change of the energy is reinitialized and is much larger than that calculated in the valley of a local minima. Thus, once caught in a local minima, the user can generate a new set of conformers to test, or alternatively, initially request more minimizer passes for each conformer (this latter, of course, involves hindsight).

The approach outlined here has several distinctive characteristics which are summarized below.

1) the ECEPP83 program depends on "brute force" calculations involving much CPU time. This is the state of the art of peptide conformer calculations though research is in progress [14,15] to construct expert systems for intelligent selections of starting conformers.

2) the simplified approach and the large grain parallelism suggested here afford nearly ideal load balancing.

3) Although the parallelization strategy suggested for ECEPP83 will not generalize to all scientific calculations, the approach presented here will have applicability to other problems, is relatively easy to understand, implement, and easy to use.

4) Most important, the speedup potential will be maximized.

The major problem in iPSC implementation of the ECEPP83 code was the management of the large amount of data that must be sent from the manager to the nodes. As mentioned above, there is a flow control problem in that messages cannot be allowed to be issued by the manager *ad libitum* without the possibility of "hanging" the system. When the message passing was coded and implemented, it was indeed found prudent to send a few messages to a node, have the node respond that it was ready for more data, then to proceed with more host to node messages. In order to send the data values in a COMMON block via a message, it is necessary to have that block declared identically both at the point of origin of the message and at the site of the reception of the message. A example of host to node message of the type employed here to send a COMMON block to a node is....

```
call sendmsg(ci1, 12, INUMRS, 2004, node, pid)
         (1)    (2) (3)     (4)     (5)  (6) (7)
```

where:

(1) "sendmsg" is a blocked send--i.e. the program proceeds beyond this point only after the message is sent. (This has no implication for whether the message has been received).

(2) ci1 is the communication channel identifier assigned by a previous "copen" call.

(3) 12 specifies the (integer) message type. It is the *only* qualifier that a node program can use to select a message to be received.

(4) INUMRS is the buffer containing the message to be sent. If it is desired to send a COMMON block, this would be the identifier of the first variable of that block.

(5) 2004 is the number of bytes of the buffer that is to be sent. If it is a COMMON block, it is the length of that block.

(6) node is the id of the node being sent the message.

(7) pid is the process id integer of the process being sent the message.

An example of a node call to receive the message given above is...

```
call recvw(ci1, 12, INUMRS, 2004, cnt, node, pid)
      (1)   (2) (3)   (4)      (5)  (6)   (7)  (8)
```

where: (2) and (3) have the same significance as for "sendmsg" and...

(1) "recvw" is a blocked receive--the calling process is blocked until the message has been received.

(4) INUMRS is the name of the buffer where the received message will be stored.

(5) 2004 is the number of bytes in the buffer named "INUMRS"

(6) cnt is the number of bytes received--if cnt is greater than 2004, the excess part of the message is truncated.

(7) node is the id of the node that sent the message

(8) pid is the process id of the process that sent the message

Figure 7 shows a segment (pseudo code) of the host program that handles the message passing to the nodes.

```
                    .
                    .
     do 25 k = 0, conformers
C
C       search for an available node
C          if freelist(i) is 0-node is free
C          if freelist(i) is 1-node is busy.
C
  50    for (i=0 to number_of_nodes) if (freelist(i).eq.0) go to 40
C
C       if every node is busy, then a recvmsg must be provided so that
C       program can receive a message that a node is free.
C
         call recvmsg(cil, type, status, 2, cnt, node, pid)
         if (status.eq.0) freelist(node) = 0
         if (status.eq.1) hung_nodes = hung_nodes + 1
         go to 50
C
  40    update freelist() and starting sending messages to free node
                    .
                    .
C
C            messages containing data sent to nodes
C            illegal input increments a variable...
C
         bad_structures = bad_structures + 1
C
C            rest of data for that conformer is passed over
C
                    .
                    .
C
C       test to see if all conformers sent out
C
       if ((structures).eq.conformers) go to 900
 25   continue
C
```

```
C     determine immediately the number of jobs left on the hypercube
C
 900  do 23 j = 0, (number_nodes - 1)
         if (freelist(j).eq.1) left_on_nodes = left_on_nodes + 1
 23   continue
      left_on_nodes = left_on_nodes - bad_structures - hung_nodes
C
C     provide a means to receive messages from remaining jobs on nodes
C     unreceived messages will eventually time out and syslog crashes
C
 27   call recvmsg(ci1, type, status, 2, cnt, node, pid)
C
C     whatever the message type or value of status,
C     there is one less job left_on_nodes
C
      left_on_nodes = left_on_nodes - 1
      if (left_on_nodes.eq.0) go to 1000
      go to 27
1000  end
```

Figure 7. Pseudocode for Host Message Control

Every effort was made to send only those COMMON blocks and variables that were actually needed. A possible problem was noticed in regards to COMMON blocks on the nodes. It was found that if an array variable was reassigned in a subroutine two calls below the node program, that change was known to the COMMON block locally, but not necessarily in the node program. In other words, the COMMON block was only updated locally and not necessarily updated two levels up in the main program. This behavior has not been further studied and verified. Presently the workaround involves sending messages out of the nodes from locations where the most recent values are known to be present.

Unfortunately, character strings can not be sent in messages. In order to send a message buffer containing the variable KV (see appendix A) which has

the type "CHARACTER*4", it is necessary to declare KV in an EQUIVALENCE statement as equivalent to a variable declared as INTEGER*4. This latter variable is able to be sent in a message as an integer to the nodes where it is again equivalenced to KV.

In the present version of the code (ECEPP86) 36 messages are sent to a node for each conformer. The information in these 36 messages totals 85532 bytes. In addition, 12 flow control (i.e. node to host) messages were utilized. In the current implementation it takes approximately 12 seconds to send the data for one conformer to any one node. After the host has sent a maximum of 6 messages or a maximum of 2 to 10 K bytes of data, the host waits for a message from the node signaling the successful reception of those messages. After the host receives such a flow control message, another segment of data is sent out. During the development of these flow control messages, it was possible, with fewer such controlling messages, to create situations where the ECEPP86 program would terminate ("hang") about every other run. In such a critical race condition, the insertion of a simple write statement in the host code could increase the percent of normal program termination. The number of flow control messages employed in the final version of the code is robust and could be cut back and a faster node loading than 12 seconds could be achieved. On the other hand, the program has not failed due to this cause since the present number of flow control messages (12) was implemented. In total 48 messages are sent (36 + 12) or about 4 messages per second. The test program mentioned earlier was able to sent about 10 per second, however, in the present

case the messages all involve the host, average almost 2000 bytes long (and hence had to be broken up into several 1K packets), and pass over more inter-node connections.

During the time of node initialization, a message may arrive from another node signaling that the node is finished with calculations on a particular conformer and ready for another assignment. Such a message must wait until the next flow control message arrives. During each pause generated by the 12 additional flow control messages from the node to the host, the host is able to receive pending messages of any type. The type accompanying the message allows the software to determine whether the message is a flow control message, one concerning an available node, or a message signaling that a node has an interrupt from which it cannot recover. If the message is the first type, the program continues to send conformer data. If it is the second type, the freelist array is updated in the appropriate position, the program loops back to receive another message. This cycle continues until a flow control message is received. If it is of the last type, the program loops back to receive another message, the freelist is not updated, and the host program will not send any new data sets to this now non-functional node. A small segment of the the host program for the above message handling is...

```
                .
                .
      call sendmsg(ci1, 34, CHIANG, 3380, node, pid)
C
C           next message could be a flow control (type 100) or
C           a notification that a node is finished and ready for
C           another task (type 200) or one signaling an interrupt
```

```
C          from which the program cannot recover or be re-
C          initializeed (type 300).
C
 705     call recvmsg(ci2, type, condition, 2, cnt, node, pid)
         if (type.eq.100) go to 710
         if (type.eq.200) then
              freelist(node) = 0
            go to 705
         endif
         if (type.eq.300) then
            hung_nodes = hung_nodes + 1
            go to 705
         endif
C
 710     call sendmsg(ci1, 35, ENOORD, 10240, node, pid)
C
                    .
                    .
```

Figure 8. Host Program Message Passing Code.

In the original program, the user could be advised of several things that have been removed from the implementation on the hypercube. The user had the option of seeing the results of each iteration, which would include, if so requested, the Cartesian coordinates of each atom, the breakdown of the total energy of the molecule into the contribution from each type of atomic interaction calculated (electrostatic, non-bonded, torsional, cystine torsional, and loop closing), and this information could be given before or just after each iteration, or both. In the present case, the user is just informed of the final optimized angle of each bond, the total energy, and the energy due to proline residues. If needed, these angles can be fed to the original program to obtain (in minimal time since the results are already optimized) the atomic coordinates.

4. RESULTS

At the present time, ECEPP83--the original program, and ECEPP86--the original program modified for the iPSC, yield similar but not absolutely identical results. A number of conformer data sets have been processed by each machine. The greatest difference between the two machines each processing the same data involved two passes through the minimizer of 100 cycles each. In one cycle, each angle of the peptide (i.e. the angles the user wishes varied) is varied at least once and then varied again continuously until the energy value for the peptide no longer decreases. The VAX 11-780 afforded a value of 68402 Kcal and the iPSC a value of 68406 Kcal. This difference is attributed to the effects of accumulated roundoff and is acceptable. Similarly, it was noticed in one part of one calculation, that after only 30 cycles, the two machines value for one angle differed by 0.30 degrees. In this case, the energy values were identical as apparently this amount of rotation in a side chain bond did not affect the free energy.

In one case, initially terrifying when first seen, the two machines produced totally different answers starting from the same data set. However, if each final set of angles was fed to the other machine, the same answer was obtained. In other words, although the two machines obtained different minimized structures from the same data, each one appeared to have traveled down

to a different energy valley. There are many local minima but only one lowest energy structure. Each machine, when fed the final set of angles calculated by the other, verified the energy values. A study of each pass through the minimizer indicated that initially the two machines were minimizing along the same path. The structure of the peptide undergoing minimization was GLY SER CYS PRO MET CYS (see Appendix A for the symbols used). The program was varying the GLY omega angle, which is the C to N planar and trans peptide bond [13], when the minimized structures started to diverge. An attempt was made to determine the exact location of the divergence and the values calculated by each machine at each point. The minimizer function utilized by the program is very complex[16]. Coupled with the fact that the minimizer increments are many and minute, the exact point of diversion could not be determined easily. The investigation was sufficient to show that the divergence occurred at a point where the calculated energy as a function of the omega angle in question was constant on one machine and had a very slight downward slope on the other. Apparently, one machine (the 11-780) persisted in changing this same angle while the iPSC switched to varying another angle. Thus, each proceeded down a different solution path.

In summary, the local minima found on one machine could be verified on the other and this phenomenon has occurred only once. Such a result does not constitute a "flaw" in parallel computation, but certainly is a consequence that must be considered with algorithms that deal with problems that have many solutions. One could consider an algorithm that calculates chess moves as

analogous to the one discussed here. There may be several minima for a particular peptide that differ only by a few Kcal and thus may be thermally interconvertible. Similarly, there may be several chess moves of similar desirability. In both cases, the algorithms presently known are incomplete as far as finding the perfect answer.

The ECEPP83 program, as originally obtained from Polygen Corporation, was ostensibly able to minimize cyclic peptides. These structures have no free end, analogous to a bicycle chain. Although the program would minimize a cyclic input structure, upon reaching the junction between the two ends, the program exited without being able to calculate the relative energy of the molecule. Since cyclic peptides are uncommon, this part of the code has not been updated to be able to complete such calculations. Although Polygen Corporation has accomplished this in their latest version of ECEPP83 called ECEPP85.

As mentioned above, the node program outputs data to a system logfile named "mylog" with a call to a library function named *syslog* which sends the data through the cube manager and into "mylog". As with user specified messages, *syslog* write commands are quite slow and the data may appear in the mylog file several seconds after initiation of the call to *syslog*. Moreover, since the data output is several lines long, it is relatively common for the output data from two nodes to be interspersed in the mylog file. Finally, the mylog file also contains all system notifications concerning not only *syslog* calls, but also loading and unloading commands etc. The mylog file may be viewed while

the ECEPP86 program is running and thus provides a way of viewing the data as it is produced. After all conformers are calculated, the data was converted by a program, called "prettyprint" (Appendix B), which utilizes the "awk" and "sort" Xenix tools to remove unwanted lines, sort any interspersed output, and present the final answers in a more easily readable format.

5. PERFORMANCE MEASUREMENTS

In section 2: reference was made to a test program which integrated a function that computed π. The point was made that this program was slower when 32 nodes were utilized than when only a single node was involved in calculations. In that case, the computational time is short relative to the time required for message communication. Messages can be blocked or unblocked. With a blocked message, no more statements are executed until the message is sent, and with unblocked messages the program continues to execute and the message is sent when the operating system finds it convenient to do so. The node operating system supports a routine named *status* which allows a process to determine the state of the buffer associated with an unblocked message. A call to *status* with the message channel identifier as an argument returns a 0 if the message has been sent (no promises with respect to reception) and returns a 1 if the message buffer is not available because the message has not yet been sent. In all the test cases utilized in this study and for ECEPP86, blocked messages were used exclusively. For the test cases, programs with unblocked messages did not execute faster than the same program with blocked messages. Apparently, the time required to send the messages in these cases was the same order of magnitude as that required to execute a *status* check to determine if the communication channel could be reused. Because of the flow control

problem mentioned, no attempt was made to use unblocked messages in the ECEPP86 program code. In order to study blocked message time requirements more closely, a polynomial integration program (integrating polynomials up to a degree of nine) was run with varying numbers of slave nodes. The entire univariate polynomial was passed to node 0. The limits of integration were divided up by node 0 into as many equal intervals as there are nodes available (total nodes minus one). Node 0 sent an integration interval to each remaining node which in turn reported its results back to node 0. After receiving all intervals, node 0 summed the results and sent a single message back to the host. The host Fortran code does not have a real-time clock function available. However, the Xenix operating system *times* procedure can be called from a Fortran program on the host. Figure 9 shows the data obtained for a degree five polynomial. "Clock Cycles" refers just to the computational time and does not include the portion of the code that is interactive data entry.

Slave Nodes	Clock Cycles (50 msec)	Host Messages (in or out)	Node Messages Node-Node
31	44	2	62
15	23	2	30
1	4	2	2

Figure 9. Profile of Integration Program

In this program node 0 is the recipient of the data from the other nodes above it. The results are accumulated here and the final results sent to the host in a single message. Clearly the computational time is very short relative to the time required to send and receive messages. The host and node programs were

then modified such that a maximum of 32 nodes were available and each of these sent their results directly back to the host for summation. With 32 nodes working and no node to node messages, the program required 551 clock cycles to run. The program sent 32 host to node messages and 32 node to host messages or about 2.3 messages per second. In the present case shown in tabular form above, a program where almost all the messages are node to node, more than 25 messages were sent per second. Node to node messages are much faster than those that enter or leave the host. As seen in an earlier discussion, the speed of node to node messages varys as a function of message length and the number of internode connections.

With ECEPP86, there is no node to node communication and therefore, all communication is host to node or node to host. The "Development of the Program" section discussed the 12 seconds required to send 85.5 Kbytes of initial data required for one conformer to the assigned node. Occasionally, ECEPP86 would run much faster, requiring only about 2.5 seconds to send the same data. This behavior occurred during the time that the program required to load all the nodes. Similar behavior could be made to occur if the dynamic loader is installed with "contention" mode rather than "polling" mode. These two options refer to the method used to pass messages. On the one hand, nodes are "polled" to see if they have any messages to send as opposed to "contention" mode where messages contend for their destination. At the time of this writing, poll mode is the standard usage and the contention option has not been officially released by the Intel company. Whether the occasional faster

behavior seen with ECEPP86 can be explained by an inadvertent change to contention mode is not known at this time. In contention mode, the program crashes randomly, when the above fast behavior occurs spontaneously, it does not! Rather, this aberrant behavior could represent some very favorable "phase" occurrence. The phenomenon has been reported to Intel and is under investigation.

The starting data for three test conformers was fed to ECEPP83 running on the 11-780 execution profiled with gprof [10]. It is important when using gprof not to leave the terminal waiting for I/O from the user. If the gprof must be used in a situation where input is required from the terminal, then it is important to type the correct input before it is requested by the program, to keep the input buffer of the terminal filled. Waiting at the terminal does affect the profiler, even though the converse might have been expected. The CPU time measured for the three test conformers was 31, 88, and 220 seconds for a total of 339 seconds. The cube could calculate the same three serially on one node in 535 seconds. The 11-780 is approximately 1.58 times faster with these three calculations than a single iPSC node. In spite of the fact that the ECEPP83 version running on the 11-780 has slightly more printout than the hypercube version, then the hypercube has the potential to calculate peptide conformers roughly 20 times faster(32/1.58) than the 11-780. In another test case, 64 sets of the above data which took 339 seconds to run on the 11-780, was run on the iPSC concurrently. Ignoring the fact that the cube was only fully loaded less than half of the time, the speedup for the iPSC over the 11-780

was 11.6. It is expected that more detailed measurements (i.e. the fully loaded time is increased relative to the partially loaded time) could show that the iPSC hypercube can achieve close to a 20 fold speedup with some SPMD applications such as the ECEPP86 program studied here.

Since these measurements were taken, the f77 library function *frexp()* has been improved. The f77 *sqrt()* function uses this function and currently the program mentioned above that required 220 seconds, now runs in 184 to 188 seconds. For the ECEPP83 program running on the VAX 11-780, this represents a 15% improvement. A recalculation of the possible speedup of the iPSC hypercube over the VAX 11-780 yields a new value of 17 (32/1.86). It is expected that each new version of either machine will continue to move this figure up or down. As long as such perturbations are relatively small, the results reported here will have the same relevance.

Finally, the laboratory which originally developed the ECEPP83 program compared the performance of a IBM 370/168 mainframe computer versus a Prime 350 minicomputer fitted with a Floating Point Systems AP-120B array processor [17]. The comparison was made with one cycle of minimization with the data for bovine pancreatic trypsin inhibitor [18]. The code used for the Prime computer was carefully programmed in assembly language for those parts of the ECEPP83 code known to be "bottlenecks" in terms of time required. The original ECEPP83 code at the start of the present work is not identical to that used in reference [17]. Thus, for the computations shown below in figure 10, the same data is utilized for three different machines each

employing different coded implementations of the same algorithm.

	AP-120B/Prime 350	IBM 370/168	VAX 11-780
Time (sec)	300	652	1030

Figure 10. Comparison of Three Computer Systems for a Benchmark

The authors making the initial comparison estimated that more of the improvement of the Prime 350 over the IBM 370/168 was due to hardware rather than to the hand coding. In the present case, the iPSC was not able to run this benchmark since it required more space than is available on a single node. As mentioned, the ECEPP83 and ECEPP86 programs have an order of complexity (O) equal to the square of the length of the peptide. The benchmark here is the data for an unusually long peptide, the starting conformer was that previously known from X-ray crystallographic data. The calculation of such a large molecule, without the X-ray data (such data is not usually available) would be unthinkable with the present state of the art. Because of its large size, certain arrays of the ECEPP86 program (viz. VAR, INDXV, and W) had to be re-dimensioned to handle the benchmark and the resulting requirements were too large to be loaded on to one node of the iPSC. This restriction is not a serious limitation of the ECEPP86 variation developed for the iPSC--as mentioned, the benchmark is an example of a somewhat unusual, but admittedly useful, application. In addition, we have already seen that even if the program could have been loaded onto a node, it would be approximately 1.86 times slower than the 11-780. So, in most cases, usage of ECEPP83

involves the serial input of many hundreds of conformers to Von Neumann machines. The iPSC can perform such calculations (32 or more conformers) concurrently and therefore a fully loaded 32 node hypercube has the potential for computation almost three times faster than even the Prime 350 ((300/((1030 * 1.86)/17))) using an array processor with hand written code. As mentioned, the Intel company plans the commercial availability of a vector processor in 1986. For a basic unit of 32 nodes, this will involved replacing every other node with a iPSC-VX (vector extension) board. It will be very interesting to test whether this configuration (16 nodes instead of 32) will realize a gain in speedup of more than the 2 fold required to "break even".

6. CONCLUSIONS

For any person considering writing original code for parallel execution or converting existing code from serial to parallel execution, several considerations can be suggested. The following suggestions are made on the assumption that it is more difficult to construct parallel computations than serial ones.

(1) What is the potential for speedup when a comparison is made between the parallel machine in question and the existing serial machine?

(2) Whether writing new or porting existing code, what debugging tools are available?

(3) What is the stage of development of the parallel machine, and what is the reliability of the software and its documentation?

(4) On parallel machines, the results of one processor must be communicated to other processors and/or to a computer controlling the processors either by message passing or shared memory. What is the overhead involved or contention problems associated with processor cooperation?

(5) For existing code, how well documented is the code, and what depth of understanding is required to modify the serial code for parallel execution?

(6) How difficult will be the thorough testing of the parallel implementation for correct output and normal termination?

(7) What is the granularity of parallelization that can be applied to the algorithm? What is the ratio of computation effort to the effort mentioned in item 4 above?

(8) Is expertise available to deal with the problems that seem to arise in any effort to improve an existing application code?

The present study has pointed out that for the case of the Intel iPSC computer, items (2) to (4) above had negative aspects which increased the implementation time or tended to minimize the gains achievable. In the case of item (5) above, it was usually the case that it was sufficient to know what the various subroutines accomplished, rather than how they were designed, in order to construct the parallel version of ECEPP83. On the other hand, the in program documentation was not updated each time the code was improved. The formal documentation was written in 1975 for the first implementation on an IBM 370 computer. In the case of the present study, positive effects of readily exploitable large grain parallelism and help from the code's developers served to mask these difficulties and those associated with items (1) to (6). The large grain parallelism implemented here matches well the single program multiple data use of the ECEPP86 program. The number of messages required did not grow with the size of the problem, and although slow, the message passing overhead is a small fraction of the computation time.

Consideration should be given to all of the above items when implementing any algorithm on a parallel machine. The completed parallel implementation must be tested extensively to help assure that the output will be

compatible with the program executed serially.

A final consideration has to do with the dynamic nature of both hardware and software of any new machine. In the case of the Intel iPSC machine, hardware improvements were made to the message passing systems and several software updates were implemented during the course of this work.

Parallel computation has a bright future. However, short of a new "parallel language", useful implementation of existing code and algorithms will require considerable effort, even under favorable circumstances. The Intel iPSC represents one hardware approach affordable to concurrent computation that is presently available. Clearly, the development and utilization of parallel hardware and software is as yet at a formative stage, and developments are proceeding rapidly. For example, in the late summer of 1986, a new vector processor is promised that should improve both vector and scalar operations. It will be interesting to see how this new hardware will affect the performance of the ECEPP86 code.

REFERENCES

(1) McGraw, J.R., and Axelrod, T.S., "Exploiting Multiprocessors: Issues and Options", Preprint from the Lawrence Livermore National Laboratory, Livermore, CA, 1984.

(2) Shapiro, E., Takeuchi, A., "Object Oriented Programming in Concurrent Prolog", New Generation Computing, vol.1, p.25, 1983.

(3) Babb, R.G. and Storc, L., "Parallel Processing on the Denelcor HEP with Large Grain Data Flow Techniques", Technical Report CS/E 85-010, Oregon Graduate Center, 1985.

(4) Babb, R. G., "Programming the HEP with Large-Grain Data Flow Techniques", in *MIMD Computation· HEP Supercomputer and Its Applications*, (ed. by J. S. Kowalik). Cambridge, MA: The MIT Press, 1985.

(5) Hammerstrom, D., Maier, D., and Thakkar, S., "The Cognitive Architecture Project", *Computer Architecture News* vol. 14, No. 1, pp. 9-21, 1986.

(6) This program was kindly provided by Cleve Moler of Intel Scientific Computers, Beaverton, OR.

(7) Chuman, H., Momany, F.A., and Schafer, L., "Backbone Conformation, Bend Structures, Helix Structures and Other Tests of an Improved Conformational Energy Program for Peptides: ECEPP83", *Int. J. Peptide Protein Res.*, vol. 24, pp. 233-248, 1984.

(8) Hagler, A.T., "Theoretical Simulation of Conformation Energies and Dynamics of Peptides", Preprint from the Agouron Institute, La Jolla, CA.

(9) "Quantum Chemistry Program Exchange", Program No. QCPE 286. Chemistry Department, Indiana University, Bloomington, IL 47401. phone: 812-337-4784.

(10) "Unix Programmer's Manual 4.2 Berkeley Software Distribution", Computer Science Division, Department of Electrical Engineering and Computer Science. University of California, Berkeley, CA 94720, 1979.

(11) "American National Standard Programming Language--FORTRAN", American National Standards Institute, Inc., 1430 Broadway, New York, N.Y. 10018, 1979.

(12) "Xenix 286 Reference Manual", Intel Corporation, 3065 Bowers Ave., Santa Clara, CA. 95051, 1984.

(13) Stryer, L., "Biochemistry", Second Edition. W. H. Freeman and Co., San Francisco, CA., 1981, or any beginning biochemistry textbook.

(14) *Biopolymers,* in press.

(15) Conversations with Polygen Corporation, Waltham, MA.

(16) Fletcher, R., and Powell, M.J.D., "A Rapid Convergent Descent Method for Minimization", *Computer Journal* , vol. 6, pp. 163-168, 1963.

(17) Pottle, C., Pottle, M.S., Tuttle, R.W., Kinch R.J., Scheraga, H.A., "Conformational Analysis of Proteins: Algorithms and Data Structures for Array Processing", *J. Computational Chem.* , vol. 1, pp. 46-58, 1980.

(18) Swenson, M.K., Burgess, A.W., Scheraga, H.A., "Conformational Analysis of Polypeptides: Application to Homologous Proteins", in *Frontiers in Physicochemical Biology* , (ed. by B. Pullman). New York, NY: Academic Press, 1978.

APPENDIX A: ECEPP86 USER'S GUIDE

INTRODUCTION

This manual assumes that the reader has some prior knowledge of protein chemistry or is willing to learn.

ECEPP86 calculates the lowest energy secondary and tertiary structure for peptides up to 100 amino acids long. The primary structure and an initial conformation must be user defined. The program employs rigid geometry without consideration of entropy or surrounding water molecules. The program does not vary bond angles, rather it varies bond rotational (dihedral) angles. For example, the distance between 1,3 atoms is not varied, but 1,4 atoms, 1,5 atoms, and higher are varied. Thus, the final energy value obtained is a relative energy and is only comparable to other conformations of the same molecule and not to the energy of another molecule--however similar the molecules may be. Minimized structures are defined by a set of dihedral angles along with a value for the relative energies. The energies for the proline (PRO) residues are output separately and thus the total energy for a conformer will be the value for the variable ETOT (total energy) plus that for the variable EPRO (proline energy).

DATA FORMAT

Any number of sets of starting conformers are listed serially in an input file named "ecpdat". Numbers in parenthesis in the following discussion refer to input line numbers in the sample input file listed below. The set of data for each conformer (except the last) is terminated by a "&" character (28,62, but not 75) and this symbol as well as the sample number *must* be in space 1 at the left margin. The standard three letter code for amino acids is augmented to include many amino acid derivatives.

Amino Acid Abbreviations

Alanine	ALA	Hydroxyproline	HYP
Aspartic acid	ASP	Serine	SER
Cystine	CYS	Histidine	HIS
Glutamic acid	GLU	H-N epsilon HIS	HIS2
Isoleucine	ILE	Proline up (phi=-67)	PRO
Leucine	LEU	Proline down (phi=-75)	PROD
Asparagine	ASN	Lysine	LYS
Glutamine	GLN	Methionine	MET
Valine	VAL	Arginine	ARG
Tyrosine	TYR	Tryptophan	TRP
Threonine	THR	Cysteine	CYSH
Norleucine	NLE	Phenylalanine	PHE

Place a "D" in front of any 3 letter code to get D-isomers. In order to avoid 5 letter codes, the following exceptions for D-isomers are used.

L-isomer	D-isomer
PROD	DPRD
CYSH	DCYH
HIS2	DHS2

The codes for the end groups have also been augmented to include derivatized N- and C- terminals.

C-terminals

Carboxyl	COOH	Methyl-carbonyl	COME
Amide	CONH	N-methyl amide	CONM
Dimethyl amide	NME2	Methyl ester	COOM
Ethyl ester	COOE	Cyclic peptide	CO-

N-terminals

Amine	NH2	Protonated amine	NH3
N-methyl	MENH	N-acetyl	ACNH
N-formyl	FONH	Cis proline	PROC
Trans proline	PROT	Charged proline	PRO+
Pyroglutamic acid	PGLU	Cyclic peptide	NH-

Although the title of a conformer can be any set of alphanumerics, the sort and awk script called "prettyprint" requires that the conformer set name be an integer (1,29,63). All the peptide output from ECEPP86 is deposited in a user defined system logfile named "mylog". "Prettyprint" takes "mylog" as input and removes extraneous system calls and sorts and reformates the mylog file to an easier to read datafile named "answers".

After the title the next line of the "ecpdat" file shows the number of times the data for a conformer is to be passed through the minimizer (2,30,64). The next line is the number of residues in the peptide, this number is always 2 more than the number of amino acids since the end groups are considered residues (3,31,65). This is still true for a cyclic peptide (i.e. residues + 2). The next three input lines are the N-terminus, the amino acid sequence, and the C-

terminus (4-6,32-34,66-68). Since the input format for the amino acids is "a4", the width of each amino acid designation must be 4 characters wide. If the amino acid designation is already 4 characters long, then it is necessary to omit the space after such a designation (33,67). If the peptide contains CYS residues (as opposed to CYSH) the next line must be a "-1" followed by x pairs of disulfide pairs (7). Be sure to allow the correct spacing demanded by the 2i5 format. Subsequently, the next x line(s) designate the residue numbers of the x pairs participating in disulfide bonds. In the present case, there is only one such line (8). If there are no CYS residues, then lines (7-8) are omitted.

The starting dihedral angles *of the backbone atoms only* can be designated by two different methods. The first is designated by lines (9,35) as GNI or "general input". Following this option, there are as many lines as there are residues (amino acids + 2). The first and last of these lines (10,16 and 36,43) are for the N-terminus and C-terminus, respectively. Not every N or C terminus has three angles, but these lines *must* have three angles designated. If the actual terminus chosen has less than three angles to be designated, the excess is ignored--but is required to be present. In between the lines mentioned, there are as many lines as there are amino acids--i.e. not counting the two termini lines (11,15 and 37,42). These lines designate the PHI, PSI, and OMEGA angles, respectively.

The second method involves the user designating a particular predetermined structure for a particular sequence of residues (69-71). The possible

choices for the predetermined structures are...

structure		input abbreviation	residues designated
R handed helix	PHI=-75,PSI=-33	HLXR	inclusive (2 4 is 2,3,4)
L handed helix	PHI=54,PSI=50	HLXL	inclusive
Beta Sheet (extend)	PHI=-154,PSI=153	BTC5	inclusive
C7 Equatorial	PHI=-84, PSI=78	C7EQ	inclusive
C7 Axial	PHI=78, PSI=-63	C7AX	inclusive
Helix type 2	PHI=-158, PSI=-58	HLX2	inclusive
Beta turn type 1	(57,127;125,-51)	TRN1	adjacent residues
Beta turn type 2	(-42,100;125,36)	TRN2	adjacent
Beta 1 region	PHI=-151, PSI=44	BT1	inclusive

This input is ended with the word "DONE" (69-71).

The above efforts are necessary to designate the starting backbone dihedral angles. The next effort is the designation of the starting amino acid side chain angles (CHI angles).

Each CHI angle of each side chain has a separate input line. The first integer is the residue number (remember that the nth amino acid has a residue number of $n + 1$). The second integer is the CHI angle number (1,2,3...etc) and the third entry is the angle in degrees. This data is ended by a line of three zeros (25,52,72, but not 60, see below). If it is desired to use default values for the side chain CHI angles, the above input is replaced by one line of three zeros (as in line 72).

At this juncture the starting conformer is completely defined. However, the user has the task of specifying which of the above angles are to be varied and which are to remain fixed. Similar to the above, there are two methods for

doing this. The easiest way is to stipulate that the variable angles are either "ALL" (73), "BACK"(backbone) (26), "SIDE" (sidechain), "NONE", or "SPEC" (53). The first three of these are self explanatory, the last option, viz. "SPEC" constitutes the second method for "specifying" (SPEC) the angles to be varied. If the peptide is cyclic, then the "SPEC" option must be utilized. Referring to lines (54-60), the first integer is the residue number, the second is the number of angles to be varied and this is followed by a list of the numbers of those angles (PHI = 1, PSI = 2, OMEGA = 3, CHI1 = 4, CHI2, = 5, CHI3 = 6, etc). The total number of entries following the second integer should be equal to the second integer. As before, these entries are ended by a line of 3 zeros (60, not 25,52).

Finally, the last entry for any conformer is the number of cycles of minimization desired (27,61,74). Shown below is a segment from a correctly formated input "ecpdat" file listing the starting conformer for three different molecules.

SAMPLE INPUT

Input line	Entry in "ecpdat" file	Format	Comments
1	1	70a1	title must be an integer
2	10	i2	minimizer cycles
3	7	i2	residues + 2 end groups
4	PGLU	a4	N end--4 characters
5	SER CYS PRO MET CYS	19a4	3 letter code words
6	COO-	a4	C end--4 characters
7	-1 x	2i5	-1 flags x pairs of -S--S-
8	y z	2i5	y disulfide bond to z

```
9     GNI 0 0                                    a4,2i2   desire for general input
10     180.    180.    180.                        *      N end--3 angles
11    -155.    157.   -180.0                              a set for each residue
12    -120.    94.00   180.0                              including end groups
13    -20.0   -79.     180.
14    -110.    117.   -98.
15    -115.    109.   -180.
16     180.    180.    180.                               C end--3 angles
17    2  1   171.2                                 *      residue number then...
18    2  2   178.                                         CHI number then...
19    3  1   67.                                          angle in degrees
20    5  1   165.
21    5  2   177.
22    5  3   179.
23    5  4   180.
24    6  1   130.
25    0  0   0.                                           end of CHI angles
26    BACK                                        a4      variable angles (KV)
27    100                                                 number of cycles
28    &                                           a1      flag to end a data set
29    2                                                   conformer number 2
30     2
31     8
32    NH3
33    GLY SER CYSHPRO MET CYSH                            no disulfide bonds
34    COO-
35    GNI  0 0
36       180.00   180.00   180.
37      -179.46   56.82   -175.14
38       65.730  -51.19   -152.0
39      -60.62   -60.63    180.0
40       67.581  -56.82    180.
41      -65.73    51.19   -98.
42       60.620   60.63   -180.
43       180.00   180.     180.
44     3  1  -162.76                                      "3" is SER--"1" is CHI1
45     3  2  -178.                                        SER--"2" is CHI2
46     4  1  -67.                                         "4" is CYSH, etc.
47     6  1  -165.                                        MET has 4 CHI angles
48     6  2   177.
49     6  3   179.
50     6  4   180.
51     7  1   130.                                        CYSH has 1 CHI angle
```

```
52     0 0  0.
53    SPEC                                a4      specify variable angles
54    2 3 1 2 3                           20i2    residue number then...
55    3 4 1 2 3 4                                 number of angles then...
56    4 4 1 2 3 4                                 sequential number: PHI=1,
57    5 3 1 2 3                                   PSI=2,OMEGA=3,CHI1=4,
58    6 6 1 2 3 4 5 6                             CHI2=5,etc.
59    7 4 1 2 3 4                                 PHI,PSI,OMEGA,CHI1
60    0 0 0                                       end SPEC with 3 zeros
61    100
62    &
63    3                                           conformer number 3
64     5
65     9
66    ACNH                                        N-acetyl
67    GLY SER SER DMETLYS PRO TYR                 D-methionine isomer
68    COO-
69    HLXR 2 6                            a4, 2i2 starting structure
70    TRN2 6 7                                    TRN1 and TRN2 must
71    DONE                                        specify adjacents
72     0 0 0.                                     use default values
73    ALL                                         vary all angles
74    100
```

APPENDIX B: PRETTYPRINT

The shell script "prettyprint" takes as input the mylog file which contains the *syslog* output of ECEPP86. Prettyprint calls the scripts pretty1 and pretty2 which utilize the Xenix functions "awk" and "sort" to convert the mylog file into a more readable form. System calls unrelated to the chemistry calculation at hand are also removed and the final output is written into a file named "answers".

prettyprint is.....

```
awk -f pretty1 mylog > slop
sort -nd +0 -1 +1 -2 slop > slop2
awk -f pretty2 slop2 > answers
rm slop slop2
```

pretty1 is.....

```
BEGIN {i = 1}
{
  if
(($11!="ACI")&&($3=="NODE:")&&($8!="Loader:")&&($8!="Cube")
  &&($8!="Cubeload")&&($8!="Load:")&&($10!="Exception")
  &&($8!="Warning--Ill"))
    {
      print $4,i,$8,$9,$10,$11,$12,$13,$14,$15,$16,$17,$18,$19,$20
    }
  else if ($11=="ACI")
    {
      print $4,i,$8,$9,$10,$12,$13,$14,$15,$16,$17,$18,$19,$20,$21
    }
  i++
}
```

Lastly, pretty2 is.....

```
BEGIN {
  x = "ETOT"
  y = "Junction"
  z = "."
    }
{
  if (($3 != x) && ($3 != y) && ($3 != z))
    {
        printf("%3d %-15s",$3,$5)
      for(i = 6; i <= NF; i++)
        {
            printf("%9.3f", $i)
        }
      printf("\n")
    }
  else
    {
        print $3,$4,$5,$6,$7,$8,$9
        printf("\n")
    }
}
```

APPENDIX C: A PEPTIDE PRIMER FOR NON-CHEMISTS

INTRODUCTION

This short appendix will introduced the subject of protein structure-function studies and the relevance to the program developed in this work.

It is safe to say that the *raison d'etre* of the genetic material of all known organisms (including viruses) is the specification of the collection of proteins that the organism is capable of synthesizing and also to specify the chronological time and place of that synthesis. The genetic material serves only that purpose and the collection of proteins so specified is a necessary and sufficient definition of the organism in question. Since all known organisms use the same genetic code, it follows that the proteins of all known organisms have the same structural basis. Proteins are all synthesized as linear polymers composed of a unique sequence of twenty different subunits--called amino acids. Proteins which are not linear and/or contain other amino acids or other molecules result from reactions that occur after synthesis as linear polymers. Once formed, proteins subsequently rapidly fold up into a characteristic three dimensional shape. Although all protein molecules and all the molecules of one protein can exist as "random coils" in three dimensional space, all biologically

active protein molecules exist in a configuration in space that is the same for each type of protein molecule. In other words, each unique protein polymer has its own unique three dimensional structure (termed its own "conformation"). Protein chemists believe that the three dimensional conformation of a protein is the thermodynamically most stable one at the temperature in question. As a corollary to this hypothesis, two or more conformers may have structures differing by only a few kilocalories in their thermodynamic stability and thus a molecule may have several populations in different conformers. Experiments now considered classics [13] have shown that *the information needed to specify the complex three-dimensional structure of proteins is contained solely in its amino acid sequence.* If a native protein is unfolded into a straight chain molecule (this can be accomplished with heat or certain reagents), the molecule can spontaneously re-fold to its native configuration. Straight chain and/or unfolded native proteins seldom exhibit their biological activity or properties as enzymes, hormones, toxins, allergens, etc. Thus, the biological activity of proteins is a function of their conformation. In order to understand the mechanism of their function, the conformation must first be determined. Only then can the interaction of proteins with cell receptors, with chemical reactions that proteins catalyze in the form of enzymes, with each other in the case of antigen-antibody reactions--be analyzed and understood at the molecular level. Structure determines function. The arrangement of the charged groups of amino acids and their electrons in space dictates the interactions with similar

groups on the target of the protein's function. Once the conformation of a protein is known, the detailed understanding of the mechanism of its function is possible--without the conformation, the mechanism of action is merely conjectural.

How then is the conformation determined?

There are a number of spectroscopic methods which yield information on the conformation of proteins and peptides (a peptide is short protein)--but not enough information can be obtained to define the entire structure. If the protein or peptide can be crystallized, X-ray crystallography can provide the total global minimum structure. This can be a long process depending on the size the protein whose structure is desired. Unfortunately, many peptides in the range of more than two and less than 100 amino acids cannot be crystallized. For these, computational methods at present are the only means available. As mentioned in the earlier sections of this work, the ECEPP83 program calculates a three dimensional structure of a peptide after the user specifies a starting conformer. The starting conformer defines the dihedral angles of both the backbone structure and those of the amino acid sidechains. The program has access to an input file which contains the bond lengths, the bond angles, and other constants for the twenty amino acids and their derivatives which are found in native and most synthetic proteins. The algorithm is based on rigid geometry, i.e. the bond angles and bond lengths are held constant, and the dihedral angles are varied, the energy of each generated conformer is

calculated, and the results evaluated by a minimizer function which acts iteratively to obtain the local minimum. The difficulty with this approach arises from the existence of many minima in the multidimensional conformational energy surface of the protein [17]. The algorithm employed for the ECEPP83 code is not able to escape from energy minima (to minimize to deeper ones) unless the minima are relatively shallow. Therefore, to reach the deepest "energy well" requires the calculation of many different starting structures. If enough starting conformers are tested and they are sufficiently spread out over the multidimensional conformational energy surface of the protein, eventually one (or more) of these test starts will minimize to the "deepest energy wells".

On a serial computer, starting conformers must be processed one at a time. The implementation described in this work allows the conformers to be tested concurrently and this represents very large grain parallelism.

BIOGRAPHICAL NOTE

The author was born February 24, 1935 in Flushing, NY and graduated from the Great Neck, NY public schools in 1953. In 1957 he received a Bachelor of Science degree in Biology and Chemistry (*cum Laude*) from Bucknell University, Lewisburg, PA. The Massachusetts Institute of Technology, Cambridge, MA. awarded him the PhD degree in Biochemistry with a minor in Organic Chemistry in 1962. After two years in the Medical Service Corps, US Army, he did post doctoral work at the National Institutes of Health, Bethesda, MD. In 1966 he became an Assistant Professor of Chemistry at the University of Oregon, Eugene, OR. and in 1972 an Associate Professor at Memphis State University, Memphis, TN (full Professor in 1978). At this time his research interests were in the area of the role of vitamins as coenzymes and protein turnover in bacterial and mammalian systems.

The author moved back to Oregon in 1981 while still remaining associated with Memphis State University as a traveling consultant in a program to train nuclear power plant operators. This program was spawned as a result of the Three-mile Island nuclear accident in 1979. He enrolled as a part-time Master's degree candidate in 1983 at the Oregon Graduate Center while continuing some part-time work.

The author has four children and enjoys various outdoor pastimes including aviation as a pilot.