PERFORMANCE ANALYSIS OF A PARALLEL MOLECULAR DYNAMICS PROGRAM

Amitabh B. Sinha
Department of Computer Science
University of Illinois
Urbana, IL 61801

Helmut Heller* **Klaus Schulten**
Beckman Institute and Department of Physics
University Of Illinois
Urbana, IL 61801

July 13, 1992

Abstract

EGO is a parallel molecular dynamics program running on Transputers. We conducted a performance analysis of the EGO program in order to determine whether it was effectively using the computational resources of Transputers. In particular, we were concerned with whether communication was overlapped with computation, so that the overheads due to communication not overlapped with computation were less. With the assistance of performance tools such as UPSHOT, and with instrumentation of the EGO program itself, we were able to determine that only 8% of the execution time of the EGO program was spent in non-overlapping communication. Our next concern was that the MFLOPS rating of the EGO program was 0.25 MFLOPS, while the Transputers have a sustained rating of 1.5 MFLOPS. We measured MFLOPS rating of small blocks of OCCAM code and determined that they matched the performance of the EGO code.

1 Introduction

A main aim of molecular biology is to understand the structure-function relationship of biological polymers, mainly proteins and nucleic acids. In the past decade it has become

*To whom correspondance should be sent

evident that the function of a biopolymer is dependent on properties other than its static structure, such as thermal mobilities of atoms, activated motions of constituent groups, local electric fields and dielectric relaxation [1, 2, 3, 4, 5, 6]. These properties are often very difficult to measure experimentally even for small subsections of biopolymers, let alone for the whole polymer. It appears that the required information can be obtained only by computer simulations of biopolymers. The prospects of computer simulations in molecular biology rest on the availability of suitable computer resources. In fact, simulations until now are limited to rather small biopolymers (of a few thousand atoms) and short simulation periods (of a few nanoseconds). Furthermore, the cardinal issue of a faithful representation of biopolymers by computer simulation is closely linked to the availability of computational resources: realistic descriptions of forces acting between the constituents of biopolymers, e.g., a proper description of Coulomb forces without 'cut-off', require enormous computer time; simulations must also represent enough of the surrounding medium, e.g. lipids of biological membranes and water, in order to achieve realistic descriptions.

Computer simulations are faced with a serious computational barrier which can be illustrated by the following estimate of the requirements on computer time: In order to determine the forces between all atoms of a protein with 12 600 atoms, i.e., of the photosynthetic reaction center of *Rhodopseudomonas viridis*, without 'cut-off', about 98 s on a Cray X-MP are needed [7]. Since the forces have to be re-evaluated at each integration step, the size of which has to be chosen 1 fs or shorter, a simulation describing a period of 1 ns requires at least one million steps, i.e., more than 1,000 days of Cray time. A 'cut-off' of pair interactions to 10 Å reduces this time to about 19 days, but one may question the soundness of such cut-off. The numbers illustrate a well-known point, namely, that computational requirements for molecular dynamics simulations are prohibitive and, for many problems, exceed all available means. This situation can be improved by employing parallel computers to simulate biopolymers. Parallel machines have speeds similar to those of much larger conventional vector machines at a small fraction of their cost. The parallel strategy has made the method of computer simulations accessible to many researchers, and allows simulations of larger systems as well as of more realistic (and computer time consuming) force models.

At present, the computational speed, even of parallel computers, is far from what is needed to solve many typical problems in structural biology, e.g., the description of biochemical reactions lasting longer than 1 μs, or the protein folding problem [8]. More powerful parallel machines, like the Intel Paragon [9] or Thinking Machines' CM-5 [10] promise to achieve Tera-FLOPS performance and allow one to carry out simulations currently beyond reach. It is essential, however, not only to demand faster hardware, but also to optimize the software for optimal performance. This optimization is two-fold: on the one hand, new algorithms [11, 12, 13, 14] can bring considerable performance improvements, while on the other hand a careful analysis and tuning of the code at hand avoids a wastage of the available processing resources. This motivated us to conduct a performance analysis of the communication-computation structure of the parallel implementation of a molecular dynamics program to determine whether further improvements in performance could be achieved by tuning the program. This is especially important for *parallel* programs. Due to the nature of parallel programs which almost always involve communication, in some form or the other, we have not only to look out for the common pitfalls of sequential programs, e.g., evaluating constant

expressions, but in addition we have to check for proper parallelization, e.g., load balancing, deadlocks, minimization of communication time.
This report on how such performance analysis was carried out should be useful to other researchers in the area of parallel molecular dynamics simulations and in other areas solving difficult computational problems on parallel computers in as much as it would give them directions in conducting a performance analysis of their code to determine the optimality of the code.
In this paper we summarize our performance study of the molecular dynamics program, EGO, running on Transputers [15, 7, 16, 17][1]. In Section 2 we present an overview of the computation involved in the EGO program and the block structure of the EGO program. In Section 3.1 we carry out a performance analysis of EGO, and in Section 4 we present our conclusions.

2 The Program EGO

2.1 Computational Task Involved

In molecular dynamics simulations a large fraction of the computing time is spent in the evaluation of Coulomb forces. This part of the code therefore presents the best opportunities for algorithmic speed-ups. Hence, we are mainly concerned with the computation of Coulomb and van der Waals forces, both involving $O(N^2)$ floating point operations, where N is the number of atoms in the molecule[2]. Since the van der Waals forces are computationally very similar to the Coulomb forces, we examine only the evaluation of Coulomb forces.

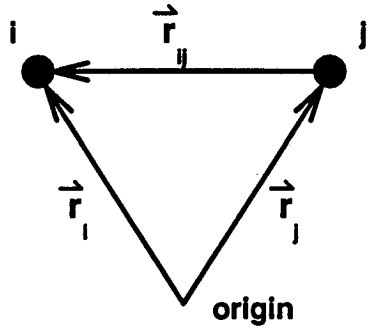


Figure 1: **The vectors $\vec{r}_i$ and $\vec{r}_j$ are with reference to the origin; $\vec{r}_{ij}$ is the difference vector.**

The Coulomb forces, which describe the electrostatic interactions in a homogeneous dielectric environment depend on the charges q_i and q_j of pairs (i,j) of atoms and on the corresponding vector $\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$ joining the atoms at positions $\vec{r}_i$ and $\vec{r}_j$ (see Figure 1). The force between atoms i and j acting on atom i is

$$\vec{F}_{ij} = \frac{q_i q_j \vec{r}_{ij}}{4\pi\varepsilon r_{ij}^3} \quad . \tag{1}$$

[1]The Transputer system is an MIMD machine, similar to an Intel hypercube or a network of workstations.
[2]$f(n) = O(g(n))$, iff there exist two positive constants c and n_0 such that $\mid f(n) \mid \leq c \mid g(n) \mid$ for all $n \geq n_0$ [18]

The force between atoms i and j acting on atom j is

$$\vec{F}_{ji} = -\vec{F}_{ij} \quad . \tag{2}$$

On a given atom the Coulomb forces of **all** other atoms act, i.e., the total Coulomb force acting on an atoms i is $\vec{F}_i = \sum_{i \neq j} \vec{F}_{ij}$. It is, therefore, necessary to determine $\vec{F}_{ij}$ for all pairs of atoms. There are $N(N-1)/2$ pairs for a total number N of atoms. Using Newton's law, Equation 2, it is not necessary to compute both $\vec{F}_{ij}$ and $\vec{F}_{ji}$; therefore it is possible to cut down the computation time to about one half by avoiding redundant computations.

2.2 Parallelization of Computational Task

The interactions between any pair of atoms are mutually independent. On a parallel computer that supports fine grained tasks each one of these interactions could be dealt with as a computational sub-task. In this investigation we want to study the performance on a Transputer-based MIMD type machine which is not efficiently used in the limit of very fine computational granularity. Rather, the Transputer is more efficient for medium grained sub-tasks. Hence, we distribute the atoms of a biopolymer over the available set of processors, each processor computing the interactions corresponding to the set of atoms local to it, typically about a thousand atoms.
The atoms assigned to a specific Transputer will be referred to as the 'own' atoms of that Transputer, all other atoms are the 'external' atoms. For a discussion of the computational task of a processor we separate the Coulomb forces into two contributions

$$\vec{F}_i = \sum_{\substack{\text{'own'} \\ \text{atoms } j}} \frac{q_i q_j \vec{r}_{ij}}{4\pi\varepsilon r_{ij}^3} + \sum_{\substack{\text{'external'} \\ \text{atoms } k}} \frac{q_i q_k \vec{r}_{ik}}{4\pi\varepsilon r_{ik}^3} \quad . \tag{3}$$

$\vec{F}_i$ is the force which acts on atom i 'owned' by the processor. In order to evaluate the first contribution the processor needs to know only the coordinates of its 'own' atoms j. The second contribution, however, requires knowledge of the coordinates of all 'external' atoms k. Each individual term of the second sum is referred to as a partial force. These coordinates are passed around the processors in such a way that any time coordinates pass by, a processor uses them to complete computation of the total force $\vec{F}_i$ that acts on its 'own' atoms. The machine topology optimal for the required communication is the systolic loop. The topology is discussed in the next section.

2.3 Topology of the Transputer System

The systolic ring topology [19, 20, 21, 7], employed in our machine is depicted in Figure 2, provides the communication channels needed for the ring of processors. The sytolic loop consists of two rings, such that each processor is connected twice to its left and right neighbors.
In Figure 2 the first ring is used to circulate the coordinates of the atoms in a clockwise fashion. By applying Equation 2 it is possible to cut down the computation time to about one half by avoiding redundant computations. This leads to the necessity of communicating partial forces along the second ring in a counter-clockwise fashion.

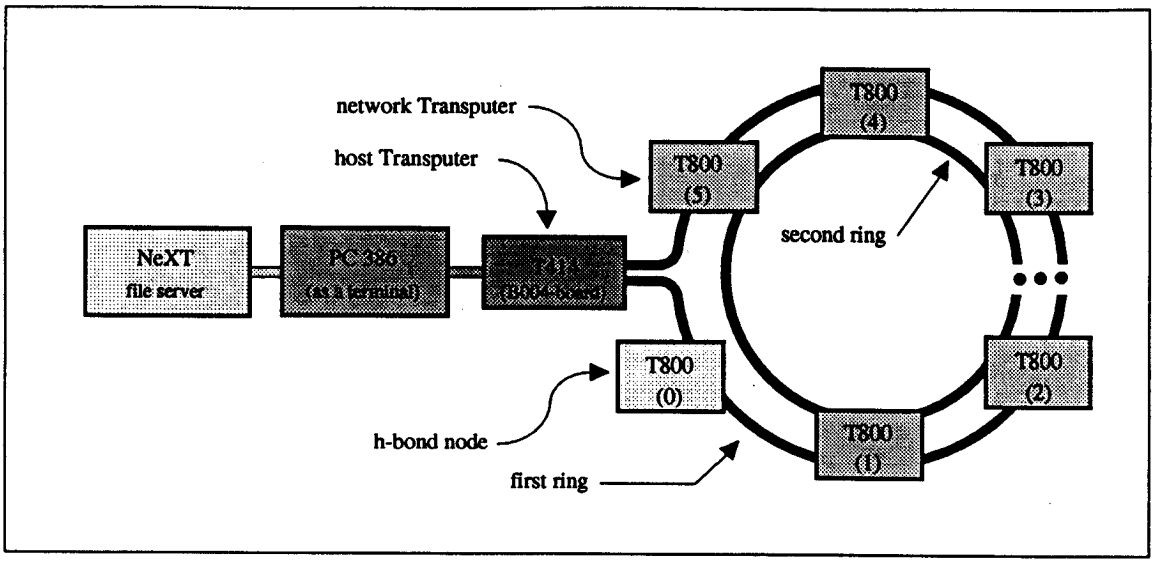


Figure 2: **Schematic diagram of the systolic ring topology of the Transputer system used in the EGO program.**

2.4 Program Flow in EGO

Figure 3 shows the program flow in EGO. The overall structure of the simulation program is a loop over the computational cycle (*initialize* → *integration*, Fig. 3). A cycle starts at the moment when the program has just completed an integration step, yielding a new set of coordinates. The computational cycle then uses these coordinates to determine the forces acting on the atoms. When these forces are determined the Newtonian equations are integrated for a time step Δt and a new computational cycle is started.

In a first step (7 → 8, Fig. 3) each node starts computation of the forces $\vec{F}_i = \sum_j \vec{F}_{ij}$ acting on its 'own' atoms i beginning with the contributions of pair interactions $\vec{F}_{ij}$ originating from all of its 'own' atoms j. In parallel with this computation[3], each node hands the coordinates of its 'own' atoms to its next neighbor in clockwise direction and receives the coordinates of atoms 'owned' by its other neighbor (5 → 6, Fig. 3). Each node keeps its 'own' coordinates in store.

The simulation program then carries out repeatedly the following steps (9 → 10, 11 → 12, 13 → 14 and 15 → 16, Fig. 3). On the basis of the coordinates just received each node evaluates pair interactions $\vec{F}_{ij}$ for the corresponding set of 'external' atoms (11 → 12, Fig. 3), and adds the results to the forces $\vec{F}_{ij}$(15 → 16, Fig. 3). As explained above, at this point the nodes may avoid evaluating interactions of pairs (i, j) which previously have been evaluated for the opposite ordering of atoms, namely (j, i), by a node which 'owns' atom j. For this purpose forces, too, need to be communicated along the ring of computational nodes. After adding all possible contributions to $\vec{F}_i$ the 'external' atoms are passed by each node to its neighbor next in clock-wise direction (13 → 14, Fig. 3), and new 'external' coordinates are received from its neighbor next in counterclock-wise direction (9 → 10, Fig. 3).

These steps are repeated until each node receives its 'own' coordinates again, at which point

[3]Communication and computation can be overlapped on the same node in a Transputer system, because communication is carried out by a DMA-engine, while computation is carried out by CPU/FPU unit. The CPU/FPU unit and the DMA-engine are independent units.

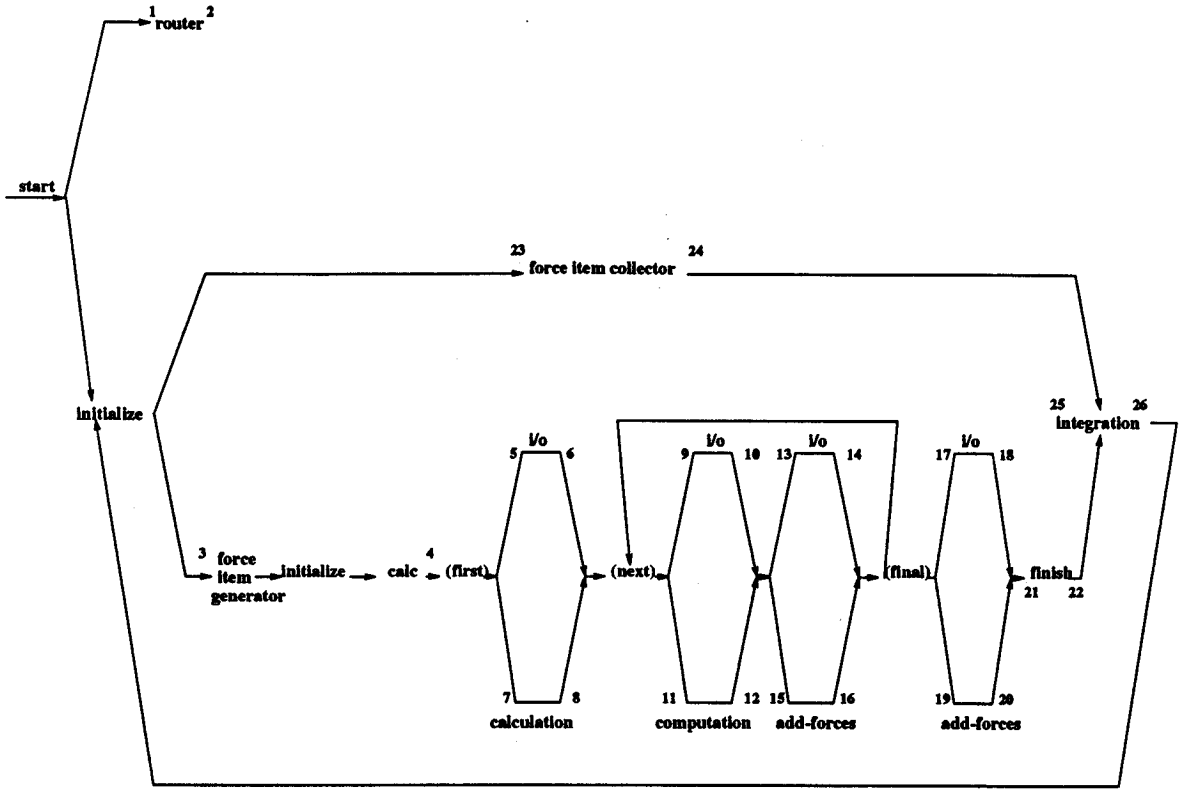


Figure 3: **This figure shows the program flow in EGO. The code segments performing communication are executed in parallel with some computational code segments. Various initializations are carried out in each integration step in stages 3 → 4. In stage 7 → 8 forces between 'own' atoms are computed with the simultaneous transfer of 'own' atoms to the neighbor, step 5 → 6. In the *next* stage loop, each processor computes the interactions between its 'own' atoms and each set of 'external' atoms (11 → 12), and simultaneously communicates its current set of 'external' atoms to its neighbors (9 → 10). Also, in the *next* stage there occurs the transmission of partial forces and the addition to the forces maintained by each processor (13 → 14 and 15 → 16).**

this loop is terminated. The strategy outlined keeps all computational nodes busy at all times except for periods when some set of nodes may have to wait for another set of nodes to complete their computations. However, these periods can be kept very short as long as all nodes 'own' nearly the same number of atoms.

3 Performance Analysis of EGO

3.1 Distribution of Execution-Time Among Program Segments

The first step in analyzing performance is to determine the parts of the program, which account for most of the execution time. Once that is done these parts can be examined more carefully to detect and eliminate performance bottlenecks.
The first few integration steps in the EGO program are disk i/o bound (this consists of

reading in and distributing the atoms of the molecule onto the processing elements of the Transputer system), and are therefore not representative of the execution pattern of the later integration steps. The results from other, later, integration steps, presumably not affected by the i/o in the first few steps, were very uniform. We, therefore, arbitrarily chose the thirty-first integration step as a representative integration step for our analysis. Table 1 shows the break-up of the execution time for the thirty-first integration step for the protein BPTI (Bovine Pancreatic Trypsin Inhibitor) with 568 atoms on a Transputer system with 25 processors. The third column is the average of the number of timer ticks (1 timer tick = 64 microseconds) spent executing that code segment on all the processors, and the fourth column shows the standard deviation from this average.

Program Phase	Code Segment	Average Timer Ticks	Standard Deviation	Percent Deviation
initial	$3 \rightarrow 4$	736.9	11.8	1.6
first	$5 \rightarrow 6$	247.0	28.4	11.5
	$7 \rightarrow 8$	504.0	24.4	4.8
next	$9 \rightarrow 10$	6735.2	565.5	8.4
	$11 \rightarrow 12$	5082.0	1000.1	19.7
	$13 \rightarrow 14$	253.0	16.5	6.5
	$15 \rightarrow 16$	277.6	6.5	2.3
final	$17 \rightarrow 18$	236.5	150.8	63.8
	$19 \rightarrow 20$	1802.7	449.8	24.9
finish	$21 \rightarrow 22$	20.1	5.0	24.8
integration	$25 \rightarrow 26$	70.0	7.0	10.0

Table 1: **This table shows the split-up of time among the various code segments (and their corresponding phases in the execution of the EGO program) for the thirty-first integration step. The first two columns show the program phases and the code segments, respectively. The phase names and the code segment numbers are with reference to Figure 3. The third column shows the number of timer ticks (averaged over 25 processors) spent in executing that code segment, the fourth column shows the standard deviation from the average time in timer ticks, and the fifth column shows the deviation as a percentage of the average time.**

Table 1 shows that the *next* phase (the loop over the stage $9 \rightarrow 16$) contributes the most to the execution time. The reason for the pre-dominance of the *next* stage is that it is in this stage that each processor computes pair-wise interactions between its 'own' set of atoms and **each** of the other 'external' sets of atoms.

3.2 The *next* Phase

In the previous section we have determined that a majority of the execution time was spent in the *next* stage. In the *next* stage there is a significant amount of communication — the

sending and the receiving of coordinates and forces from each processor to and from its right and left neighbor, respectively, in the ring topology. In this section we take a more detailed look at the computation and communication in the *next* phase.

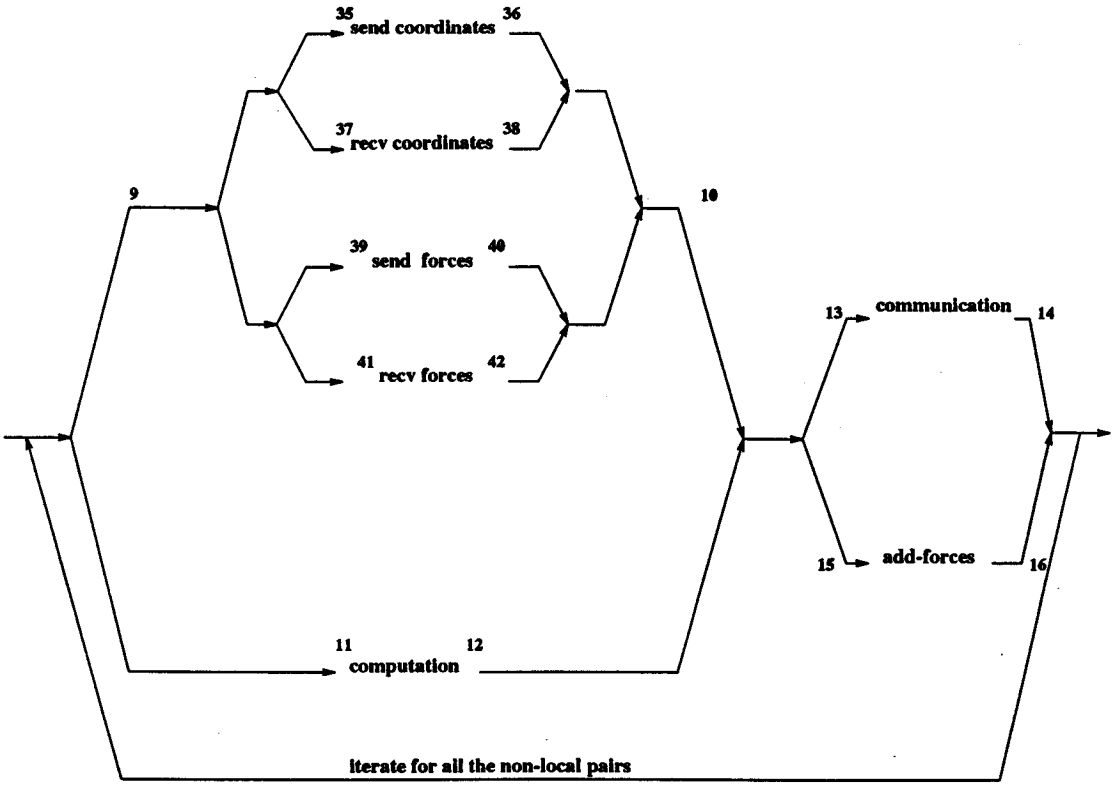


Figure 4: **In the *next* phase loop, each processor computes interactions between its 'own' set of atoms and each of the 'external' sets of atoms(11 → 12). Simultaneous with the computation of interactions each processor is carrying out a significant amount of communication — sending its current set of 'external' atoms to the next neighbour in the ring (35 → 36), receiving its next set of 'external' atoms from its predecessor (37 → 38), sending partial forces of the previous set of 'external' atoms to next neighbour (39 → 40), and receiving partial forces on its current set of 'external' atoms from its predecessor (41 → 42).**

Figure 4 provides a detailed view of the *next* stage. The *next* phase is repeated for all 'own' and 'external' pair combinations, avoiding duplication of efforts in the computation of the interaction forces between atoms i and j. Stages 35 → 36 and 37 → 38 represent the sending and receiving of coordinates between a processor and its successor and predecessor, respectively. Similarly, stages 39 → 40 and 41 → 42 represent the sending and receipt of forces between a processor and its successor and predecessor, respectively. The computation of the pair-wise interaction between a processor's 'own' set of atoms and the currently residing 'external' set is done in stage 11 → 12. Stages 13 → 14 and 15 → 16 do not take much time, and are thus not important for this analysis (the execution times for these stages appear in Table 1).

Table 2 shows the time spent by EGO in executing the various sub-segments of the *next* phase.

It can be determined from the table that both the computation and communication times are significant. What is not apparent from this table is to what degree the communication time is overlapped with the computation time. In the following section we look at some performance tools that provide a visual representation of the *next* phase.

Code Segment	Average Timer Ticks	Standard Deviation
11 → 12	5082.0	1000.5
35 → 36	3508.6	935.6
37 → 38	3270.9	812.7
39 → 40	4872.0	775.6
41 → 42	4853.9	1136.2

Table 2: **This table shows the split-up of time among up the various code sub-segments of the *next* phase of computation. Some of the sub-segments have been left out, because their corresponding timings appear in Table 1. The first column shows the code segments, the numbers are with reference to Figure 4. The second column shows the number of timer ticks (averaged over 25 processors) spent in executing that code segment, and the third column shows the standard deviation from the average time.**

3.3 Communication Overhead Analysis

We had two choices for a visual tool to look at the execution of the EGO program — ParaGraph [22], and Upshot [23]. Both ParaGraph and Upshot run under X-Window and are post-execution tools. They provide displays based on a logfile containing the trace of the program execution. We describe both these tools briefly and explain our choice for a computational performance tool.

ParaGraph, developed by Heath and Etheridge at Oakridge National Laboratories, offers a wide range of analyses — critical path computation, lengths of message queues, concurrency profile, spacetime diagrams etc. Its primary drawback is that many of the analyses are dependent on the assumption of synchrony in message passing. Message passing synchrony essentially means that the time for a send on one processor should precede the time for the corresponding receive on another processor. Since most non-shared memory machines, including the Transputer, do not have a common clock this poses some problems. For this purpose an additional tool, PICL (Portable Instrumentation Communication Library) [24], is provided. PICL implements communication primitives on some non-shared memory machines in a distributed fashion, and the PICL implementation takes care of synchrony issues. However PICL is not implemented for Transputers. Therefore, even though ParaGraph offered enviable information about the execution of the program, we opted against it because without the assistance of PICL it turned out to be very difficult to produce the proper logfiles for post-execution data analysis.

Upshot was developed by Herrarte and Lusk at the Argonne National Laboratory. It does not provide the analyses supported by ParaGraph, but its requirements for the logfiles are much

less stringent. Our main concern was to be able to display the execution of the EGO program, and analyze it for communication overheads. Upshot displays the logfile information either (or both) as an event trace of the program over all processors over time or as a trace of states of the program on all processors over time, where a state has a beginning and a closing event. We chose Upshot as the performance tool, because its logfiles could be easily generated by checkpointing critical points in the EGO program.

The local clocks of the processors of the Transputer system are reset at the beginning of every program execution. This provides sufficient synchrony between the clocks local to each Transputer to properly order events during data analysis, since the drift of the clocks over our performance runs (10 minutes wall clock time) was small enough — less than 2–3 timer ticks. This approximate synchrony is necessary if we are to make any sort of relevant analysis of overlaps of communication and computation on a processor and the overlap of the iterations of the *next* phase over all the processors.

The trace information was generated by clocking the events marked in the flow chart of Figure 3 and the diagram for the *next* stage in Figure 4. Information was stored in local buffers while an integration step was being performed. At the end of the integration step data was written out to the host processor. This way no additional communication during an integration step was introduced which would have disturbed the execution pattern of EGO. The generality of the UPSHOT program allowed a very simple clocking mechanism to generate the desired trace for runs of the EGO program.

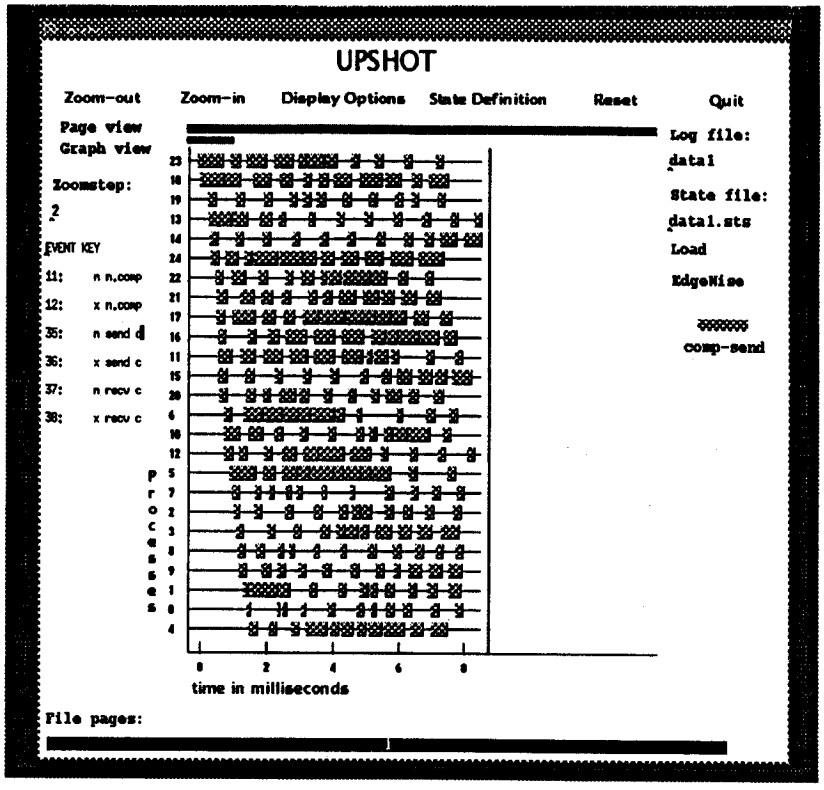
UPSHOT displays only one event for each processor. To monitor concurrency of execution of two events on the same processor, one can adopt one of the following methods:

- The first approach is to simultaneously display the communication and computation in the *next* phase in two separate UPSHOT runs. This can be done by having two separate log-files, one containing just communication events and the other computation events (or more trivially by having the same log-file, and displaying only the appropriate states).

- The other approach is to split the events occurring on one processor into communication and computation events, and displaying them on different virtual processors in the same UPSHOT run.

The UPSHOT views for communication and computation are displayed in Figure 5, the upper picture showing the communication and the lower picture showing the computation in the *next* phase. A qualitative idea about the overlap of computation and communication can be obtained from these figures. We also determined the percentage of time spent exclusively in communication, i.e., the amount of wasted time. We describe below the corresponding procedure to determine this time.

Each period of communication has a starting event occurring at time, S_m, and a finishing event occurring at time, F_m. Similarly, each period of computation has a starting event occurring at time, S_p, and a finishing event occurring at time, F_p. We will denote the periods when communication occurred alone, C, by pairs of these states, e.g., $\{S_m, S_p\}$. The magnitude of non-overlapping communication can be estimated as $S_p - S_m$. There are six possible ways in which a pair of communication and computation periods can interleave (the commas separate the events in the order they occur):



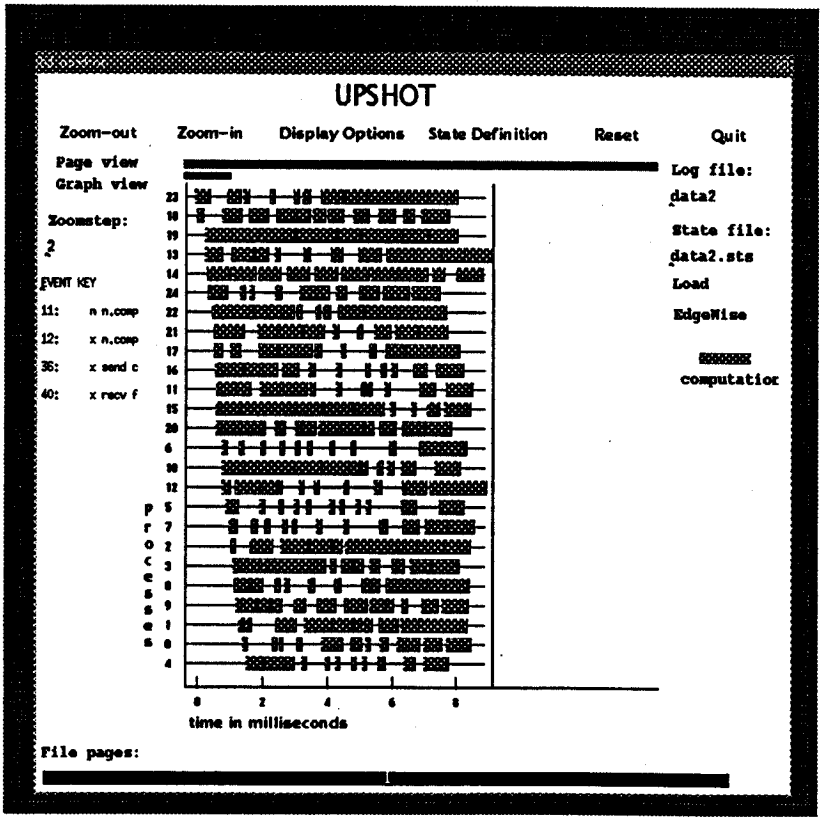


Figure 5: **These figures present the UPSHOT views of communication and computation in the *next* phase of the EGO program.**

1. (S_m, F_m, S_p, F_p): $C = \{S_m, F_m\}$.

2. (S_p, F_p, S_m, F_m): $C = \{S_m, F_m\}$.

3. (S_p, S_m, F_m, F_p): $C = \emptyset$.

4. (S_m, S_p, F_m, F_p): $C = \{S_m, S_p\}$

5. (S_p, S_m, F_p, F_m): $C = \{F_p, Fm\}$.

6. (S_m, S_p, F_p, F_m): $C = \{S_m, F_m\}$.

A record of communication and computation periods and a knowledge of the above interleavings allows one to estimate the overall non-overlapping communication time, C. We were able to determine that in the *next* phase 8% of the execution time (averaged over all the processors) was spent in non-overlapping communication.

3.4 MFLOPS Achieved

The small fraction of time spent in non-overlapping communication didn't leave much room for improvement in the communication overheads of the EGO program. The only issue that we had not addressed so far were possible improvements of blocks of sequential code. Our first step in that direction involved measuring MFLOPS rates for segments of code and comparing them with the rated performance of the Transputers — 1.5 MFLOPS (sustained, for a 20 MHz part [25]).

The Transputer manuals [25, 26, 27, 28, 29, 30, 31] provides data for number of cycles taken to execute various floating point operations. The corresponding figures for addition, subtraction, multiplication, divison and square root of double floating operations are 7, 7, 20, 32 and 80 cycles, respectively. An addition takes 350 nanoseconds[4], hence 2.86 million additions can be done per second. Since the floating point rating of the Transputer is 1.5 MFLOPS, an addition can be taken to correspond to 0.5 floating point operations. Similarly subtraction, multiplication, divison and square roots are equivalent to 0.5, 1.5, 2.4, and 6.3 floating point operations, respectively.

Armed with the above figures we evaluated the inner loop of the EGO code for its floating point performance. The MFLOPS rate for this code segment turned out to be a dismal 0.27 MFLOPS. Since there were conditional statements and loops in that program segment, we conducted tests to determine MFLOPS rates for some other small demonstration programs. The demonstration programs are shown in Figure 6.

The execution time for program segments A, B, C, D, and E in Figure 6 were 432, 180, 442, 500, and 2370 timer ticks[5], respectively. Using the floating point conversion data determined earlier on, program segments A, B, and C had MFLOPS ratings of 0.36, 0.87, and 0.35, respectively. The corresponding MFLOPS rating for E would be 0.57 if we considered the overheads of the conditional statement, and 0.75 otherwise. We did not compute MFLOPS

[4] assuming a 50 ns cycle time on a 20MHz Transputer

[5] 1 timer tick = $64\mu s$

```
REAL64 new, old:
new = 1.0(REAL64)
old = 0.5(REAL64)
```

```
SEQ i=0 FOR 10000
  SEQ
    new := new + old
    new := new - old
```

(A)

```
new := new + old
new := new - old
new := new + old
new := new - old
...repeat last two steps 9998 times
```

(B)

```
SEQ i=0 FOR 1000
  SEQ j=0 FOR 10
    SEQ
      new := new + old
      new := new - old
```

(C)

```
SEQ i=0 FOR 10000
  IF
    new=1.0
      new := 0.5 (REAL64)
    new=0.5
      new := 1.0 (REAL64)
```

(D)

```
SEQ i=0 FOR 10000
  IF
    new>old
      new := new*new
    new<old
      new := DSQRT(new)
```

(E)

Figure 6: **Code blocks used to measure the MFLOPS performance.**

rating for program segment D, but it can be seen from the timings obtained that a conditional statement can take a considerable amount of time.
The poor MFLOPS performance of these very basic program segments was an indication that the MFLOPS performance of EGO (the basic structure of EGO being very similar to program segment C) was not unusual, and would probably need assembly coding to achieve better performance.

4 Conclusions

In this paper we have traced out the efforts in conducting a performance analysis of EGO, a parallel molecular dynamics program running on Transputers. The performance analysis was carried out by generating trace data for a run of the EGO program, and then examining the data with performance tools such as Upshot, and some of our own routines to measure communication overheads. We were able to determine that the EGO program did not suffer from one of the main drawbacks of parallel programs — communication overheads. We were also able to determine that the poor MFLOPS rating of the EGO program was not unusual by measuring MFLOPS rates for some small sample program segments. The ratings for these

program segments differed considerably from the expected sustained rating of 1.5 MFLOPS for the Transputers. The MFLOPS rate of these code segments could be improved with re-coding many of these program segments into assembly language. The drawback of such codes would be their non-portability.

5 Acknowledgements

This work was supported by the National Institutes of Health Resource *Concurrent Biological Computing* under grant number 1 P41 RR05969*01. We thank Rusty Lusk at Argonne National Laboratory for assistance with the Upshot performance tool. We also thank L. V. Kale at the University of Illinois for insights into performance bottlenecks and techniques for analysis of parallel programs.

References

[1] M. Karplus, in: Studies in Physical and Theoretical Chemistry, ed. J. L. Rivail (Elsevier Science Publishers, Amsterdam, 1990) p. 427.

[2] M. Karplus and J. A. McCammon, Scientific American 4 (1986) 30.

[3] M. Karplus and J. A. McCammon, Ann. Rev. Biochem. 53 (1983) 263.

[4] C. L. Brooks III, M. Karplus, and B. M. Pettitt, Proteins: A Theoretical Perspective of Dynamics, Structure and Thermodynamics (John Wiley & Sons, New York, 1988).

[5] J. A. McCammon, B. R. Gelin, and M. Karplus, Nature 267 (1977) 585.

[6] J. A. McCammon and S. C. Harvey, Dynamics of proteins and nucleic acids (Cambridge University Press, Cambridge, 1987).

[7] H. Heller, H. Grubmüller, and K. Schulten, Molecular Simulation 5 (1990) 133.

[8] F. M. Richards, Scient. Am. (1988) 54.

[9] Intel Corporation (1991).

[10] The Connection Machine CM-5 Technical Summary (Thinking Machines Corporation, 1991).

[11] L. Greengard and V. Rohklin, J. Comp. Phys. 73 (1987) 325.

[12] L. Greengard, The Rapid Evaluation of Potential Fields in Particle Systems (MIT Press, Cambridge, 1988).

[13] L. Greengard and W. D. Gropp, A parallel version of the fast multipole method (1988).

[14] L. Greengard and V. Rokhlin, On the efficient implementation of the fast multipole algorithm (1988).

[15] K. Boehncke, H. Heller, H. Grubmüller, and K. Schulten, in: NATUG 3: Transputer Research and Applications 3, ed. A. S. Wagner (IOS Press, Amsterdam, 1990) p. 83.

[16] H. Grubmüller, H. Heller, A. Windemuth, and K. Schulten, Molecular Simulation 6 (1991) 121.

[17] B. Banko and H. Heller, User Manual for EGO (405 N. Mathews Ave., Urbana, IL 61801, U.S.A., 1991), [Beckman Institute Technical Report UIUC-BI-TB-92-07].

[18] E. Horowitz and S. Sahni, Fundamentals of Computer Algorithms (Computer Science Press, Rockville, Maryland, 1989).

[19] N. S. Ostlund and R. A. Whiteside, in: Macromolecular Structure and Specificity: Computer-Assisted Modeling and Applications, eds. B. Venkataraghavan and R. J. Feldman (Annals of the N. Y. Acad. of Sciences 439, New York, 1985) p. 195.

[20] W. D. Hillis and J. Barnes, Nature 326 (1987) 27.

[21] A. R. C. Raine, D. Fincham, and W. Smith, Comput. Phys. Commun. 55 (1989) 13.

[22] M. T. Heath and J. A. Etheridge, ParaGraph: A tool for visualizing performance of parallel programs.

[23] V. Herrarte and R. Lusk, User Manual for Upshot.

[24] G. A. Geist, M. T. Heath, B. W. Peyton, and P. H. Worley, PICL: a portable instrumented communication library, C reference manual (1990).

[25] Reference Manual: Transputer Architecture (1987), #72TRN04803.

[26] The Transputer Databook (1989).

[27] IMS T800 Architecture (1987), Technical note 6, #72TCH00601.

[28] IMS T800 Transputer (1987), Preliminary Data, #42108200.

[29] The Transputer Family 1987 (1987), product information.

[30] Reference Manual: Transputer (1985).

[31] IMS T414 Transputer (1987), preliminary Data, #42107801.